A COMPILER FOR A FUNCTIONAL PROGRAMMING SYSTEM

Shih-Lien Lu

November 1984
CSD-840045

UNIVERSITY OF CALIFORNIA

Los Angeles

A Compiler for A Functional Programming System

A thesis submitted in partial satisfaction of the

requirement for the degree Master of Science

in Computer Science

by

Shih-Lien Lu

1984

ACKNOWLEDGEMENT

I would like to thank Professor Milos Ercegovac, my advisor, for his generosity, patience, encouragement, and valuable help during the preparation of this thesis. I also would like to thank my parents for their support. Last, but not the least, I would like to thank God who has been my strength in the years of my study.

This research is supported in part by a grant from NASA Lewis Research Center NAG 3-132.

ABSTRACT OF THE THESIS

A Compiler for a Functional Programming System

by

Shih-Lien Lu

Master of Science in Computer Science

University of California, Los Angeles, 1984

Professor Milos D. Ercegovac, Chair

The functional style of programming provides an alternative in problem solving to the sequential style of languages commonly used. Given this different style of programming, a new approach to executing programs written in functional language is desired. With conventional computing modules in mind, we proposed to translate programs written in functional languages to an intermediate language. This intermediate form Complete Decomposed Form, bridges the gap between the functional language and the conventional hardware.

Programs written in functional languages are first translated into CDF and then into the C programming language to be executed on an existing machine - VAX-11/780. Some issues concerning code optimization are discussed. With an application in mind such as real time simulation of continuous systems, we also explore a methodology of solving problems in the functional programming style.

Table of Contents

page

List of Figures

page

CHAPTER 1
INTRODUCTION

1.1 Background

It is widely believed that a computer architecture should be based on a high-level language in order to achieve ease of use (programmability), performance and effectiveness in use of hardware resources [KELL-79]. Most high-level languages in use today are not supported by the underlying architecture of conventional machines in this sense, resulting in a large semantic gap. It is also evident that these languages have difficulty in managing the complexity of the underlying software needed. Some of the problems require excessively expensive solutions in these languages which are unreliable, inflexible, and hard to maintain. These problems can also be attributed to the style and the nature of the programming languages used [BACK-78].

1.2 The Problem

The "Software Crisis" resulted from the fact, as Backus points out, that conventional programming languages are "fat" and "flabby". As languages evolved they grew bigger and more complex in order to incorporate new features while retaining old ones. The payoff rate decreases dramatically as languages become very complicated and difficult to learn. The small advantages gained in programmability by adding new features is offset by the problems caused by the

added complexity. The second problem of conventional languages is their inflexibility in combining existing programs and procedures to construct more complex programs. Practically most programs in conventional languages are written individually. There is little sharing in existing routines that perform similar tasks. Side effects resulting from state changes and variables are the main cause of these problems. Together with the complexity and inflexibility, there is also the problem of "correctness." Because conventional programming languages lack useful mathematical properties, programs are hard to reason about and debug. Denotational and axiomatic semantics are elegant and powerful concepts, yet in using them to describe conventional languages one can only "talk" about programs.

1.3 The Approach

A conceptually different approach, based on the concept of functions, to high-level languages, programming and architecture has been investigated by a number of researchers [ARVI-82, BERK-75, ARVI-81, DENN-79, KELL-79, PATE-81, PLAS-76, TREL-80, TURN-79, WATS-82]. Functional or applicative languages offer a nontraditional approach to programming. They are easy to learn because of their small sets of primitives and simple semantics. Yet, a functional language can be made powerful enough to handle with complex algorithms. It also has an important advantage of being very close to human reasoning in problem solving. Moreover, programs written in functional languages have the algebraic properties which simplify correctness proofs [WILL-81]. Since programs have neither states, nor variables, a program has no side effects. However, there are difficulties with the functional style of languages in achieving cost-effectiveness in execution. The problem of cost-effectiveness in executing

functional programs is strongly related to the representation of the program at run time and compatability of this representation with the execution-level resources (eg., processors and communication) [ERCE-in preparation]. The main objective of this work is to develop an intermediate level representation for functional programs and a corresponding translator. This intermediate form could be executed on a special architecture, or be compiled for execution on conventional computers.

1.4 Thesis Overview

With a particular class of applications in mind, we adapted an intermediate representation for functional programs, called Complete Decomposed Form (CDF) [FELL-81]. This intermediate form narrows the semantic gap between functional languages and conventional hardware used to execute them.

In Chapter 2 we discuss basic concepts of functional languages, their semantics and syntax, largely based on Backus [BACK-78]. In Chapter 3 the intermediate level language (CDF) is introduced. Motivations for DF as well as advantages and disadvantages are discussed. In Chapter 4 some transformations useful for optimization are described. In Chapter 5 we discuss the implementation of translation of CDF into C on VAX-11/780. Finally in Chapter 6 we illustrate the approach with an example, solving a real-time simulation problem.

CHAPTER 2

THE HIGH-LEVEL LANGUAGE

2.1 Introduction

High-level languages are necessary when complex computations to be performed on computers. There are several criteria we look for in choosing a high-level programming language:

(1) Clarity and simplicity;

(2) Conceptual consistency;

(3) Suitability for applications;

(4) Expandability;

(5) Efficiency in execution, translation, program creation and program testing.

Conventional high-level languages rarely possess a majority of the qualities mentioned above. An alternative to the conventional languages is a functional-style of programming language [BACK-78]. Functional (applicative) programming languages are based on the concept of mathematical functions. Programs written in a functional language map objects to objects. There are neither states, nor variables. A functional language satisfies most of the criteria mentioned above. It is simple and clear. Its semantics are consistent and easy to understand. It acquires complexity as required by different applications. Expandability

is its strength too. The only disadvantage it exhibits is in the efficiency of execution on a conventional machine.

2.2 Functional Language

The functional programming system is based on the use of a fixed set of combining forms called functional forms. These, plus simple definitions and primitives, are the only means to build new functions from existing ones. With a given input, a functional language program always maps the input into the same output. Backus' functional language consists of:

(1) A set of objects: O.

(2) A set of predefined primitive functions: F.

(3) An operation, application; denoted ":".

(4) A set of functional forms: FF.

(5) A set of definitions: D.

Throughout the thesis we will be using FP to refer to this functional language defined by Backus.

2.2.1 Objects, Operation and Definitions

Objects can be "bottom", which is "undefined", or null, which is the empty set, or a single atom, which can be a string of characters or numbers, or a list of atoms, or a list of lists. A list is denoted as (a,b,c....), where a, b, c.... can be atoms or lists. A null list is represented as "()".

The operation refers to the applying of functions on objects. The action of this application is termed an execution of a function with the object as its argument.

User can define functions. These definitions are the partial expressions of the algorithm used to solve a problem. Programs are built from primitives, defined functions and functional forms.

2.2.2 Primitives

There are several types of primitives, arithmetic, logical, conditional, and data list manipulation. Arithmetic primitives are addition, subtraction, multiplication, division, and perhaps other functions such as SIN, COS, LOG and EXP. Logical primitives are OR, AND, XOR and NOT. Both of these two type of primitives accept objects of type (a, b), where a and b are atoms. For example, apply + on (1, 2) will produce 1+2=3. Conditional primitives are EQ, GT(greater than), LT(less than), AT(atom), NL(null), and LS(list). The first three primitives act on pairs (a, b), where a and b are atoms, and return values True or False (T or F). The last three primitives check the type of objects. There are primitives which are used to manipulate lists, SLn(select(n)), PK(pick), FR(front), TL(tail), RL(rotate left), RR(rotate right), AD(append), PR(pair), SP(split), IX(index), CT(concat), CL(circle left), CR(circle right), TR(transpose) etc. These primitives and their semantics are described in more detail in [BACK-78, LAHI-81].

2.2.3 Functional Forms

Primitive functions can be combined using several functional forms. The functional forms used are:

(1) **Compose** *: denoted* CM(f, g):x;
f and g are primitive functions or definitions, and x is an object. It returns the result of applying f on the result of applying g on x.

(2) **Construct** *: denoted* [f,g,h,...]:x;
if returns (f:x,g:x,h:x...). That is, it constructs a list of results from applying functions f, g, h... on x.

(3) **Condition** *: denoted* IF(p,f,g):x;
if p applied on x has result true (T) then it returns f:x, else it returns g:x.

(4) **Apply to all** *: denoted* AP f:x;
if x=" ?" (bottom) it returns " ?",
else if x=() it returns (),
else if x=(x1,x2,x3,x4,x5,....xn),
it returns (f:x1,f:x2,f:x3,..., f:xn).
(n>=1)

(5) **Insertion** *: denoted* IN f:x;
if x=" ?" it returns " ?",
else if x=() it returns (),
else if x=(x) it returns x,
else it returns
(IN f:(x1,x2,...xn-2),f:(xn-1,xn)).
(n>=2)

(6) **Constant** *: denoted* K y:x;
if x=" ?" it returns " ?"
else it returns y.

The notation for primitives and functional forms can be easily modified to improve readability as discussed in the following section.

2.3 Syntactic Variation

It has been pointed out that FP can be used to implement algorithms [LAHTI-81]. Its ability to match the problem and to map the solution algorithm closely to program coding has been recognized. Many examples are given to show how FP helps to evaluate and guide programmers to tackle a particular problem differently in order to achieve better performance. Unfortunately, the prefix form is not conducive to understand programs written in FP. This is clearly seen when a numerical problem is solved with prefix notation. For example, a simple evaluation of the two roots of quadratic polynomial:

$$p(x) = ax^2 + bx + c \tag{2.1}$$

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{2.2}$$

$$x = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{2.3}$$

will create a quite unreadable version as illustrated in Figure 2.1.

Besides the readability, prefix form also requires user to specify explicitly the parallelisms inherent in the expression. For example: $x=a+b+c+d$ can be expressed in two ways in the prefix form: Clearly we observe that the first expression implies parallelism when executed.

```
/* evaluate two roots of quadratic equation
/* INPUT: (a, b, c)

posx = CM(/,[CM(-, [sqrtofb4ac, negb|, twoa|));
negx = CM(/,[CM(+, [negsqrtofb4ac, negb|, twoa|));

   negb          = CM(*, [K-1,SL2|);
   twoa          = CM(*, [K2,SL1|);
   negsqrtofb4ac = CM(*, [K-1,sqrttofb4ac|);
   sqrtofb4ac    = CM(SQRT, CM(-,[bsq,4ac|));

      bsq        = CM(*, [SL2, SL2|);
      4ac        = CM(*, [K4, CM(*, [SL1, SL3|)|);

;
```

Figure 2.1 Solution of Quadratic Equation in FP

```
x=CM(+,[CM(+,[a,b]),CM(+,[c,d])]);
or
x=CM(+,[a,CM(+,[b,CM(+,[c,d])])]);
```

One solution is to allow expressing functions in both the prefix and infix forms. We prefer to use the algebraic style of expressing problems, without losing the ability of using functional forms to construct and manipulate programs.

Basically, we allow users to define frequently used algebraic expression with infix notation. A pre-processor can be used to convert the infix notation into prefix form.

For example we can express one root of quadratic equation as follows:

```
x=((-1)*b+sqrt(b^ 2-4*a*c))/2*a
```

which if necessary is translated into prefix form:

```
x=CM(#*,[CM(#/,[CM(#+,[CM(#*,[K-1,b]),
    CM(sqrt, CM(#-,[CM(POW,[b,K2]),
```

```
CM(#*,[CM(#*,[K4,a]),c])])]),K2]),a]);
```

Several other examples illustrate the use of this preprocessor is given as follows:

```
/* ************************************** */
/*  FUNCTIONS                             */
/*  notice the arguments of function      */
/*  f,rad,g imply a given input type      */
/*  and they are automatically mapped     */
/*  input selection primitives            */
/* ************************************** */

f(x,y,z)=g(x)+h(y)+func(x+y+z,x*y*z);
rad(x,y)=sin(x^2+y^2)+cos(x^2+y^2);
g(x,y,z)=f(x+deltax,y+deltay,z+deltaz);

/* ************************************** */
/* EXPRESSIONS                            */
/* ************************************** */

area=1/3*pi*radius^2;
vol=4/3*pi*radius^3;
radius=x+y+z+w+s+d*f;
pi=3.14159;
;
```

After the preprocessing only prefix form is showed:

```
/* FUNCTIONS */

f=CM(#+,[CM(#+,[CM(g, SL1),CM(h, SL2)]),
  CM(func, [CM(#+,[CM(#+,[SL1,SL2]),SL3]),
  CM(#*,[CM(#*,[SL1,SL2]),SL3])])]);
rad=CM(#+,[CM(sin, CM(#+,[CM(POW,[SL1,K2]),
    CM(POW,[SL2,K2])])),CM(cos, CM(#+,[CM(POW,[SL1,K2]),
    CM(POW,[SL2,K2])]))]);
g=CM(f, [CM(#+,[SL1,deltax]), CM(#+,[SL2,deltay]),
    CM(#+,[SL3,deltaz])]);

/* EXPRESSIONS */

area=CM(#*,[CM(#*,[CM(#/,[K1,K3]),pi]),CM(POW,[radius,K2])]);
vol=CM(#*,[CM(#*,[CM(#/,[K4,K3]),pi]),CM(POW,[radius,K3])]);
radius=CM(#+,[CM(#+,[CM(#+,[CM(#+,[CM(#+,[x,y]),z]),w]),s]),
     CM(#*,[d,f])]);
pi=K3.14159;
```

We observe that allowing programs to be expressed in infix can improve readability. Functions defined in infix form still retain their functional properties.

Discussion of Functional Programming Approach

We now illustrate some advantages of a functional language approach to programming over a conventional approach, by considering the following example:

EXAMPLE

Newton's method for finding the root of a function:

This iterative procedure consists of:

* An initial guess *x0*,

* Approximation of derivative as :

$$\frac{df}{dx} = \lim \frac{\Delta f}{\Delta x}$$

$$= lim \frac{(f(x+delta)-f(x))}{\delta}$$

(as delta -> 0)

* A new value *newx:*

newx <-- x-f(x)/f'(x)

Substitute *newx* into the function to find

the new value of *f(newx).*

Repeat this process until

$|f(x)-f(newx)| < limit.$

In this example we assume:

$f(x)=x^3 - 2x^2 - 2x + 4$

A FORTRAN solution to this problem might be:

```
C
C    NEWTON'S METHOD
C
     PROGRAM NR(INPUT,OUTPUT)
C
  03 FORMAT(1P,2E14.5)
C
     PRINT, *X*
     READ, X
     PRINT, *       X          F(X)*
     FX=F(X)
     PRINT 3, X, FX
C
C    FIND DERIVATIVE
C
  22 DELTA=0.00001*X
     DIFF=(F(X+DELTA)-FX)
     DERIV=DIFF/DELTA
C    FIND NEW X
     X1=X-FX/DERIV
     FX1=F(X1)
     IF (ABS(X1-X).LT.0.00001) GOTO 99
  59 CONTINUE
C
C    GET NEW X THEN REPEAT
C
  81 X=X1
     FX=FX1
     PRINT 3, X, FX
     GOTO 22
  99 STOP
     END
C
C    FUNCTION DEFINITIONS
     FUNCTION F(X)
     F=X**3-2.0*X**2-2.0**X+4.0
     RETURN
     END
```

Using infix FP to solve the same problem:

```
/* newton's method */

limit    = 0.00001;
delta(x) = limit*x;

/* f(x) */

fx(x)    = x^3-2.0*x^2-2.0*x+4.0;
fofdx(x) = fx(x+delta);
dfdx     = (fofdx-fx)/delta;
```

```
newx(x)  = x-fx/dfdx;

/* main */

main         = newtonsmethod;
newtonsmethod = CM (repeat, initial);
initial      = [newx,oldx];
repeat       = IF( test, SL1, continue);
continue     = CM( repeat,[CM(newx,SL1),SL1]);
test         = CM(GT, [limit,CM(#FABS,#-)]);
oldx         = ID;
;
```

However the infix form is transformed into prefix form as follows:

```
/* newton's method */

limit=K0.00001;
delta=CM(#*,[limit,SL1]);

/* f(x) */

fx=CM(#+,[CM(#-,[CM(#-,[CM(POW,[SL1,K3]),
    CM(#*,[K2.0,CM(POW,[SL1,K2])])]),
    CM(#*,[K2.0,SL1])]),K4.0]);
fofdx=CM(fx, CM(#+,[SL1,delta]));
dfdx=CM(#/,[CM(#-,[fofdx,fx]),delta]);
newx=CM(#-,[SL1,CM(#/,[fx,dfdx])]);

/* main */

main         = newtonsmethod;
newtonsmethod = CM (repeat, initial);
initial      = [newx,x];
repeat       = IF( test, SL1, continue);
continue     = CM( repeat,[CM(newx,SL1),SL1]);
test         = CM(GT, [limit,CM(#FABS,#-)]);
;
```

From the above example we can see several advantages of the FP approach to programming. First, the program is hierarchically structured. The main routine, "newtonsmethod", first, sets up an initial pair of *(newx,oldx)*. Then it tests if *newx-oldx* is less than the limit (test). If not, it

continues to get another *newx* from the old *newx*. This testing and acquiring *newx* is repeated until the desired accuracy is achieved. Instead of jumping around with "goto" statements, the program is well structured and easy to read.

The second advantage is related to concurrency. This is observed from the definition of function f(x). An inherent parallelism is demonstrated as we try to evaluate f(x)=x**3-2*x**2-2*x+4.

The third advantage lies in the fact that FP programs are easier to debug than conventional language programs. FP tells the user exactly where the problem is when a function generates an illegal object. Thus, the user can easily pinpoint the mistake. Another advantage has to do with the language itself. This particular functional language has less than fifty primitive functions, instead of all the complex structures of FORTRAN statements, DO, GOTO etc. As a result, this language can be learned in a short time.

In summary, we mention the following advantages of a functional programming language approach:

(1) Concise, clear and easy to learn language definitions;

(2) Well-defined, extensible and simple semantics;

(3) Well-structured programs, easy to compose;

(4) No variables or control statements; easy to debug;

(5) Natural expression of parallelism; and

(6) Possibly very close to the mathematical model of a problem.

One of the problems of the functional language approach is the lack of suitable architectures on which functional programs can be efficiently executed. There are several proposals for functional language machines [ARVI-82, DARL-81, DENN-79, MAGO-81, KELL-79]. These proposals emphasize novel aspects of machine representation and require special resources.

In the next chapter we will describe an intermediate language which, in our opinion, is more suitable for execution on conventional architectures.

CHAPTER 3

THE INTERMEDIATE LANGUAGE - CDF

The main objective of this thesis, as mentioned in the first chapter, is to develop a translator for programs written in the Backus' functional language into object code which is "close" to the conventional processors. We have selected an intermediate language (CDF [FELL-81]) as the target of our translator. From CDF, we can either interpret programs directly on a suitable machine, or we can further translate CDF into a conventional language to be executed by a conventional machine (Figure 3.1).

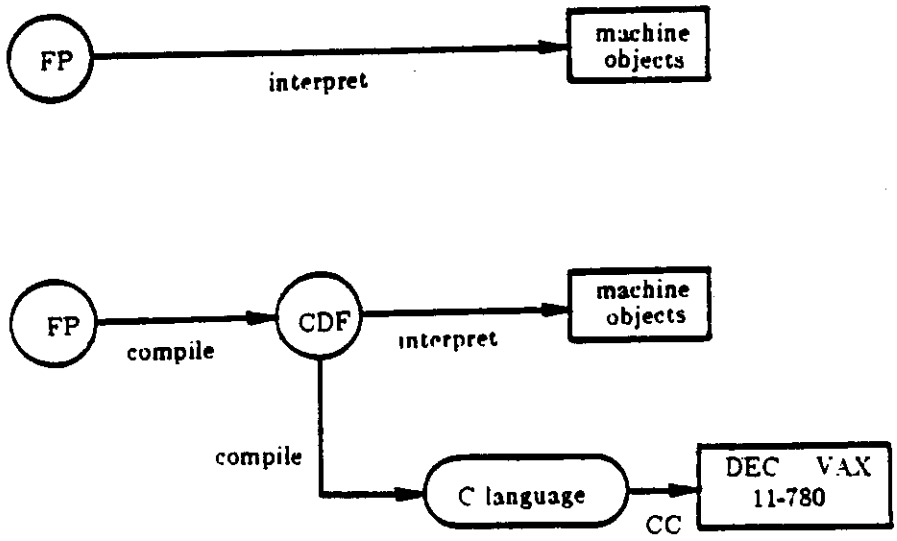


Figure 3.1 Illustration of the Methodology

Several advantages can be gained by using an intermediate form. First, the semantic gap between functional programming languages and machine can be narrowed. As the CDF is "closer" to the hardware it may make interpretation more efficient than interpreting functional language programs directly. Second, the intermediate language gives the flexibility of not confining the target language to a particular hardware organization. The intermediate form can be translated into many machine languages. In this study we chose the C language and DEC VAX 11/780 as the execution environment. This is presently being studied in [RAVI-83]. Third, the intermediate form simplifies task allocation in a multiprocessor environment.

3.1 Definition of CDF

The functional form COMPOSE is one of the main constructs in functional programs. It corresponds to the "sequential" flow of computation during execution. For example, a definition of "D = CM(f,g):x" can be treated as obtaining first D1 = g:x, and then applying f on D1 to get D = f:D1. The intermediate language is defined to represent the functional program in a decomposed form with respect to its sequential parts as specified by the functional form COMPOSE. This form of the intermediate language has been suggested in [FELL-81]. A complete decomposition form (CDF) of a functional program takes the advantage of the sequential aspect of composition, and breaks a long program into statements. Abstractly, a program in CDF is a list of statements representing the traversal of the parsed program tree, each consisting of a label, a ":" and a body instruction. The label enables another instruction to refer to it and to request the value needed. Each label is unique within a program. In the case of a recursive definition,

two labels may be equated to express that a definition has already been given. In this case the statement looks like "fn := fm".

An instruction consists of an "operator" and "arguments lists". An operator can be either a primitive function or a functional form. Argument lists can be lists of labels whose value need to be evaluated or a "." which indicates input data object or the data object produced in the program. The length of the argument list is not fixed. The functional form CONSTRUCT has no definite length, while the length of functional form IF is three. For functional forms AP, AI, IN, it can either be one or two. There is a unique syntactic name for each functional form and primitive with the exception of COMPOSE which, due to the decomposition is eliminated.

3.1.1 A Formal Description

Statements in CDF can have three different forms as defined in the following:

(1) <LABEL> : <PRIMITIVE> <LABEL>
(2) <LABEL> : <FUNCTIONAL FORM> <LABELS>
(3) <LABEL> : "=" <LABELS>

where:

<LABELS> : <LABELS> <LABEL>
<FUNCTIONAL FORM> : "CN" (construction)
| "AP" (apply to all)
| "AI" (associative insert)
| "IN" (insert)
| "IF" (condition)
| "K" (constant)

<PRIMITIVE> : "AR" (append right)
| "AL" (append left)
| "AT" (atom)
| .....

```
                .
                .
<LABEL> : "." (input)
        "f"<DIGITS>
<DIGITS> : <DIGITS> <DIGIT>
<DIGIT>  : "0"
         | "1"
         ...
         .
         .
         | "9"
```

This syntactic specification implies a demand-driven model [TREL-81]. A syntax for data-driven model can be specified as shown in the following:

(1) <LABEL> : <PRIMITIVE> <LABEL>
(2) <LABEL> : <FUNCTIONAL FORM> <LABELS>
(3) <LABEL> : "=" <LABELS>
(4) "*" <FUNCTIONAL FORM>
(5) "*END-" <FUNCTIONAL FORM>

where

<LABEL> <LABELS> <PRIMITIVES> <FUNCTIONAL FORM> are unchanged from the specification given above.

Demand-driven and data-driven specification are pre-order and post-order traversals of the execution tree, respectively.

3.1.2 Graphical representation:

The semantics of CDF is rather simple. For primitives, the label on the left of ":" contains the address of the result after applying that primitive on the object pointed by the label on the right of ":". For functional forms, there are different meanings according to the individual functional forms.

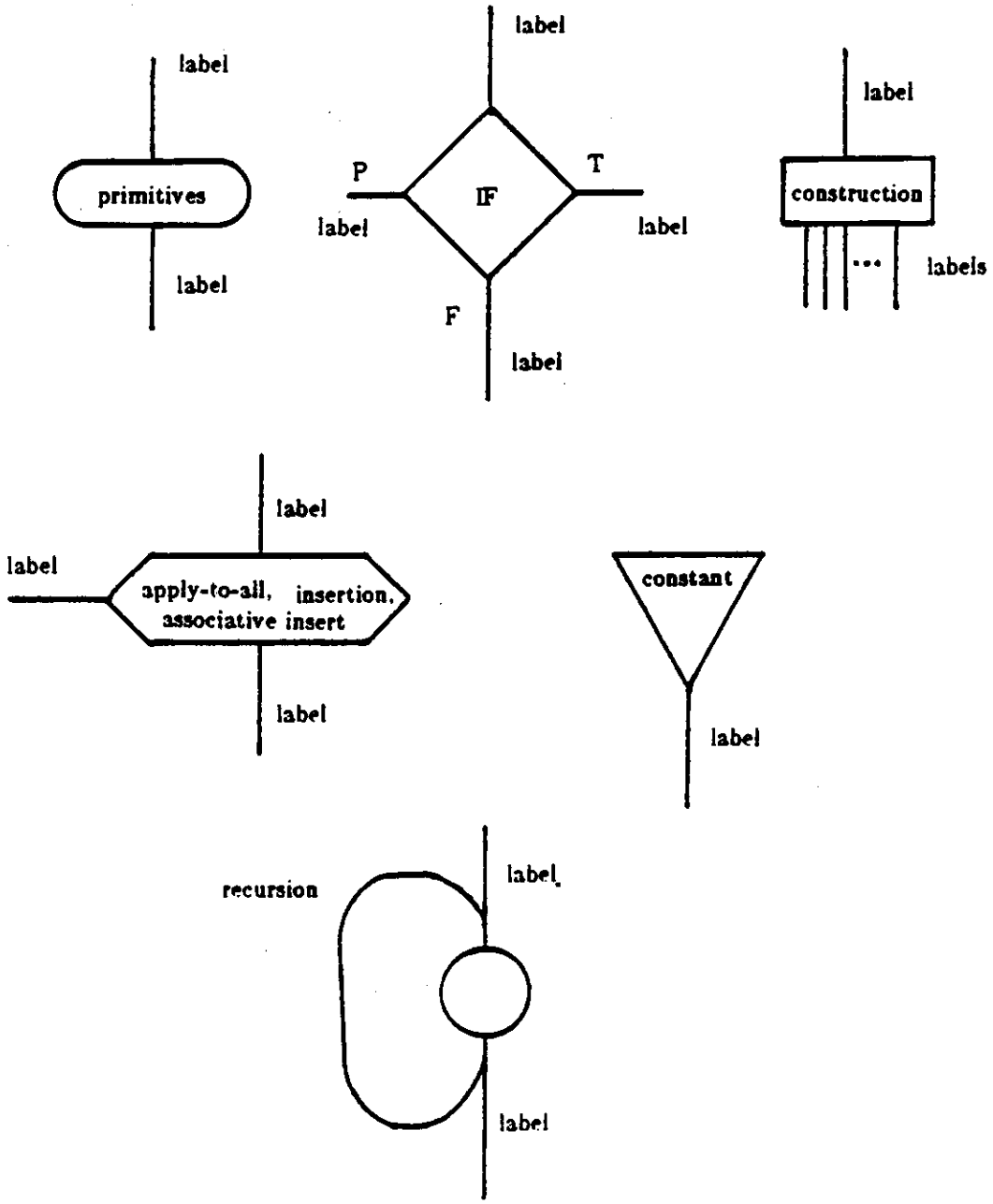


Figure 3.2 Graphical Representation of CDF

Functional forms IN, AI, and AP take two labels, the first one is the function definition to be applied or to be inserted, the second is an object. Graphically, they can be represented as in figure 3.2.

3.2 An Example

To have a clearer view of the CDF we give an example in a functional language and translate it into both data-driven and demand-driven forms of CDF. Refer to the following FP specification of the program we will parse. This program produces indices of a list of elements matched to a given key.

```
/* This program takes a key and a list and returns all indices of
/* the list which match the key.
/*
/* INPUT : (key,(array of elements))
/*       key - an atom
/*       array of elements - the list we are searching
/* OUTPUT: 0 - if no element matches the key
/*       <list of indices> - indicates the indices of elements
/*                           that match the key.
/* SUBPROGRAMS:
/*  equal : check each elements with the key
/*  index : get all indices
/*  pairs : pair the result of equal with indices
/*  search: checks, and returns all indices that match
/* EXAMPLE:
/*  equal : (a,(a,b,c,d,e,f)) = (T,F,F,F,F,F)
/*  index : (a,(a,b,c,d,e,f)) = (1,2,3,4,5,6)
/*  pairs : ((T,1),(F,2),(F,3),(F,4),(F,5),(F,6))
/*  search : pairs : (input) = (1)
/*  main  =  (1)

   equal = CM (AP EQ, DL);
   index = CM (IX, LN, SL2);
   pairs = CM (TR, [equal, index|);
   search= CM (CT, AP IF(SL1,[SL2|,K[]), pairs);
   main  = CM (IF (NL,K0,ID),search);
   ;
```

Parsing an FP program into this syntactic type of the CDF implies the

demand driven model of execution. It is shown as follows:

```
/* The CDF statements are :

    f0 : IF f2 f3 f4
    f2 : NL f1
    f3 : K0
    f4 : ID f1
    f1 : CT f5
    f5 : AP f7 f6
    f7 : IF f8 f9 f10
    f8 : SL1 .
    f9 : CN f11
    f11 : SL2 .
    f10 : K
    f6 : TR f12
    f12 : CN f13 f14
    f13 : AP f16 f15
    f16 : EQ .
    f15 : DL .
    f14 : IX f17
    f17 : LN f18
    f18 : SL2 .
```

This same example can also be translated into a syntactic form which implies a data-driven model of execution as illustrated in the following:

```
f7 : DL .
 *AP
f8 : EQ .
f5 : AP f8 f7
 *END-AP
f10 : SL2 .
f9 : LN f10
f6 : IX f9
f4 : CN f5 f6
f3 : TR f4
 *AP
f12 : SL1 .
f15 : SL2 .
f13 : CN f15
f14 : K[]
f11 : IF f12 f13 f14
f2 : AP f11 f3
 *END-AP
f1 : CT f2
f16 : NL f1
```

```
f17 : K0
f18 : ID f1
f0 : IF f16 f17 f18
```

The codes between corresponding "*<FUNCTIONAL FORM>" and "*END-<FUNCTIONAL FORM>" are treated as macros.

Graphical representations for both the demand and the data-driven style are shown in Figure 3.3 and Figure 3.4, respectively.

3.3 The Translation from FP to CDF

Since all functional forms and primitives are in prefix form, it is natural to write the parser recursively. First, the program text written in FP is read into an input buffer in which all comments and tags for tracing ("@") are deleted. Then a user definition name table is established to check for recursion. The translator starts by requesting the definition named "main". The main part of the translator is the parser, which checks the text to see if a name is a user defined name, a primitive function, or a functional form, in this order. If it is a primitive, the appropriate label for the operand is assigned and printed. If it is a user defined name, a procedure is evoked to check if there is recursion. If it is not a recursive call, the parser continues to parse; otherwise the statement number which is repeated is equated to the current one to produce an "fn := fm" type statement. If it is the functional form CM (composition), it is removed. If it is an another functional form the parser continue to parse the appropriate field supplied to this form. If a name is not identified, an error message is printed: "undefined term or function." This is repeated until all of the text has been parsed or an error is detected. Tags are kept and incremented according to different functional

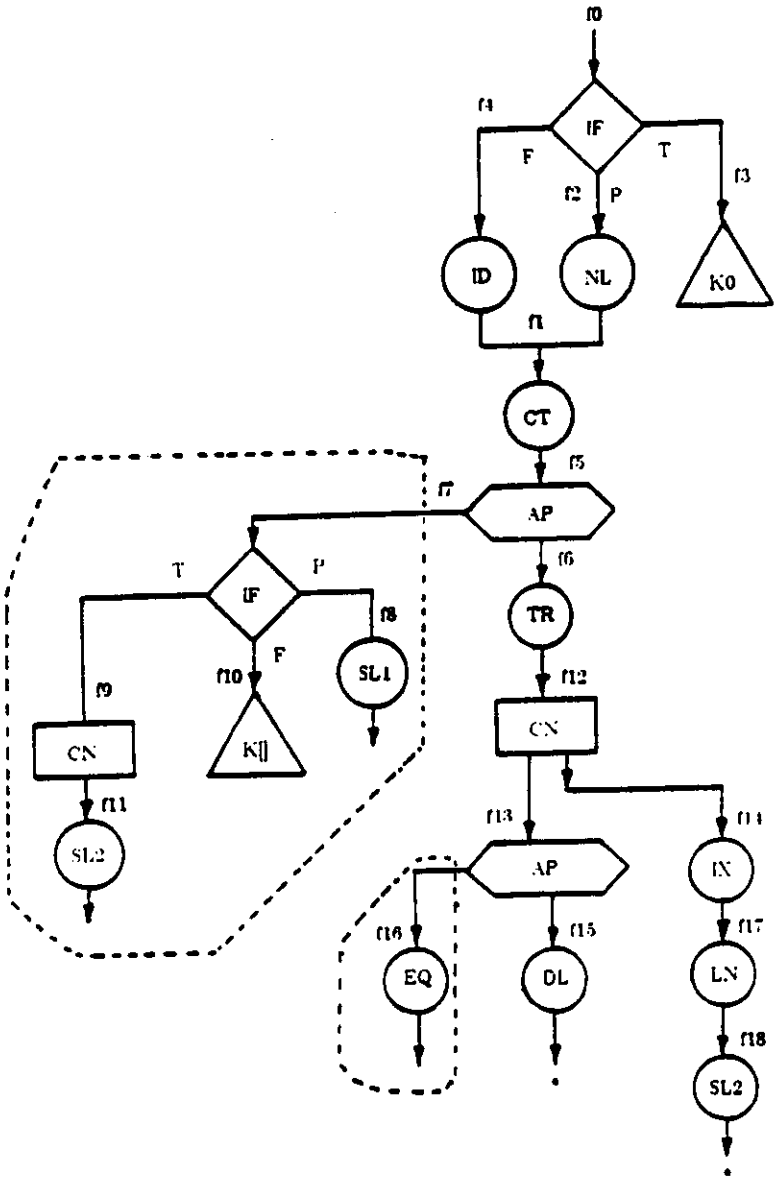


Figure 3.3 Demand Driven Version

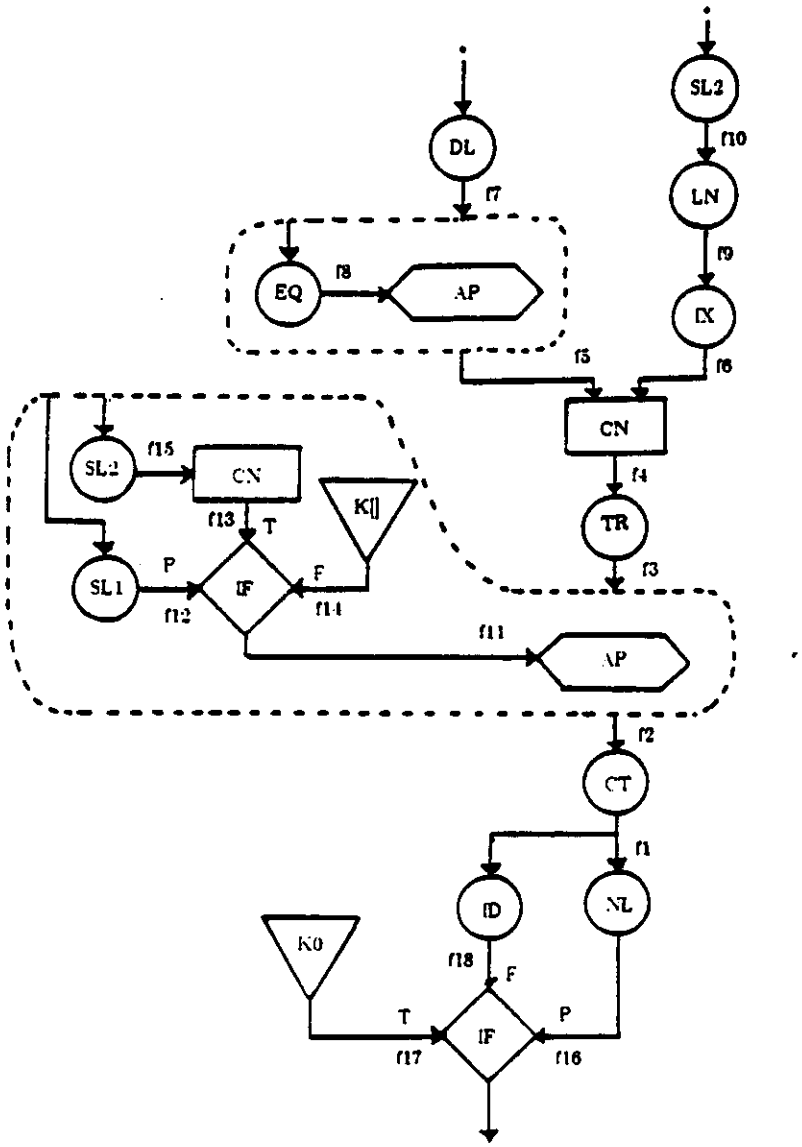


Figure 3.4 Data Driven Version

forms, primitives and recursion. In the following we give an algorithm for parsing FP programs into CDF form:

```
procedure parse(string,label,comp,ff)

check(string) :

 case 1: user defined name :

     if (not recursion) then
        parse(userdef,label,compo,ff);
     else print (label,":","=",previouslabel,ff);

 case 2: PRIMITIVES :

     print (label,":",primitives,ff);

 case 3: FUNCTIONAL FORMS :

     check(functional-form):

      case 1: CONDITION

            print(label,":",tag,tag+1,tag+2);
            get-the-operands;
            for each operands do
            begin
              parse(operand,tag,compo,ff);
              tag = tag +1;
            end;

      case 2: CONSTRUCTION

            count-the-number of elements;
            i = number of elements
            print(label,":");
            for index = 1 to i do begin
              tag = tag + 1;
              print(tag);
            end;
            for each elements do begin
              parse(element,tag,compo,ff);
              tag = tag + 1;
            end;

      case 3: APPLY TO ALL
                | INSERT
                | ASSOC. INSERT

            get the rest of the string after the functional form;
```

```
            temp = the rest;
            if (compo = true) then
              print(label,":",string<1:2>,tag++);
            else print(label,":",string);
            parse(temp,tag,0,ff);

        case 4: CONSTANT

            print(label,": K", string);

        case 5: COMPOSITION

            temptag = 0;
            for each elements in COMPOSITION do
            begin
              tag = tag + 1;
              temptag = temptag + 1;
            end;
            for i = 1 to temptag do
              parse(element,i,compo,ff);

    otherwise: ERROR;

end-proc.
```

3.4 Discussion

There are several reasons why it is useful to convert FP representations into CDF. In CDF we can perform preliminary code optimization suitable for conventional machine organization. This can be accomplished because functional languages have no side effects. A function definition applied on the same data always produces the same result. If two sections of code are the same, only one copy is needed. This is particularly useful because it allows us to take advantage of FP modularity. It is often to the user's advantage to build complex programs from existing software. However, this flexibility and modularity may cause repetition and inefficiency. Code repetition in the CDF can be detected and removed. For example:

A polynomial

*f(x)=x**4+x**2*

can be defined using FP as:

```
f=CM(+,[x-forth,x-square]);
x-forth=CM(*,[x-square,x-square]);
x-square=CM(*,[ID,ID]);
```

The corresponding CDF is:

```
f0 : + f1
f1 : CN f2 f3
f2 : * f4
f4 : CN f5 f6
f5 : * f7
f7 : CN f8 f9
f8 : ID .
f9 : ID .
f6 : * f10
f10 : CN f11 f12
f11 : ID .
f12 : ID .
f3 : * f13
f13 : CN f14 f15
f14 : ID .
f15 : ID .
```

Notice that since f8, f9, f11, f12, f14, f15 are the same

This program can be optimized by replacing f9,f11,f12,f14,f15

with f8 to obtain:

```
f0 : + f1
f1 : CN f2 f3
f2 : * f4
f4 : CN f5 f6
f5 : * f7
f7 : CN f8 f8
f8 : ID .
f6 : * f10
f10 : CN f8 f8
f3 : * f13
f13 : CN f8 f8
```

By repeating the optimization, the CDF program is
further reduced to:

```
f0 : + f1
f1 : CN f2 f3
f2 : * f4
f4 : CN f5 f6
f5 : * f7
f7 : CN f8 f8
f8 : ID .
f6 : * f7
f3 : * f7
```

By eliminating f6, f3 we obtain the final form:

```
f0 : + f1;
f1 : CN f2 f5
f2 : * f4
f4 : CN f5 f5
f5 : * f7
f7 : CN f8 f8
f8 : ID .
```

CDF also provides sequencing order for execution on a conventional machine. A limited number of primitives and functional forms can be coded to form an instruction field. For primitives, each statement takes a single operand referenced by a pointer (an address). Functional forms require two operand fields, with one being the address of a function to be applied on the second operand, specifying an object reference. A subprogram is generated with appropriate data assigned.

Lastly, statements are completely tagged so a multiple processors environment could be used to achieve high performance. This is achieved by allocating blocks of the CDF statements into different processor and executing them independently. A CDF program can be activated in demand-driven or data-driven manner. Since CDF programs represent the execution tree, task allocation for multiple processors environment would be straightforward.

CHAPTER 4

TRANSLATION FROM *CDF* TO *C*

We discuss in this Chapter an implementation of a translator to convert programs from CDF into C. Here C is assumed as the target language because it can be efficiently executed by the available machine (DEC VAX 11/780), and because UNIX [THOM-78, KERN-78a ,KERNb] supports many software tools for C. The translation approach is based on macro substitution. The translation contains a procedure for each of the pre-defined primitives. A statement in CDF is equivalent to a procedure call in C. The data object needed for that primitive function is referenced by a pointer to a character string. This implies that all data objects are stored as strings of characters. The primitive function returns a pointer to a string of characters which is the result of the application of that particular function. Functional forms are treated differently from primitives as discussed in the following section.

4.1 Functional Forms

There are six functional forms allowed in our functional language. They are CM (*composition*), [] (*construction*), K (*constant*), IF (*condition*), IN (*insert*), AI (*associative insert*) and AP (*apply-to-all*). The functional form *composition* is removed as we translate FP into CDF. The remaining functional forms have different semantics. Yet *insert*, *associate insert*, and *apply-to-all* are very similar to each other in that all of these take two

operands, one being the function to be inserted or applied, the other being the data object. For *apply-to-all*, data object is divided into N elements, with N being the length of the data. Each of the N elements is submitted to the same function. Results from these function evaluations are put together to form the final result. We illustrate this with a description as follows:

AP f:x

case 1 : x = ? --> ?
case 2 : x = atom --> ?
case 3 : x = (x1,x2,...xN) (N>=2)

```
AP f:x =
  (1) for i:=1 to N do begin
        y[i] <-- f: x[i]
      endfor
  (2) collect all results and do construction.
        result <-- [y1, y2,... yN]
```

In this case it is not necessary to produce multiple copies of the function being applied. For a multi-processor system it is possible to broadcast the data elements to different processors, apply the same function, then collect each of the results to form the final result. For *insertion*, a sequential process is required:

IN f:x

case 1 : x = ? --> ?
case 2 : x = x1 --> x1
case 3 : x = (x1, x2, x3,.... xN) (N>=2)

```
IN f:x =
  (1) y <-- xN
  (2) for i:= N-1 downto 1 do begin
        y <-- f : [ x[i], y ]
      endfor
  (3) result <-- y
```

For *associative insert*, a function is defined recursively on the object split from the original data.

AI f:x

case 1 : x = ? --> ?
case 2 : x = x1 --> x1
case 3 : x = (x1, x2, x3, ... xN) (N>=2)

AI f:x =
(1) **if** *N is odd* **then**
f:(AI f:[x1,x2,...xN-1/2],
AI f:[xN+1/2,...xN])
(2) **else**
f:(AI f:[x1,x2,...xN/2],
AI f:[xN/2,...xN])
endif

In a multi-processor environment all these recursive calls can be thought of as parallel function evaluations. For *construction*, all of the functions within the *construct* are evaluated simultaneously. Their results are passed back to the calling procedure. The calling procedure returns the concatenation of these results upon completion.

The conditional statement takes three operands. These operands can not be evaluated simultaneously since they might cause an infinite loop. The first operand of the form is evaluated first in order to determine which of the other two functions is to be executed to obtain the result.

IF(f,g,h)

If *(f:x equal True)* **then return***(g:x)*
else return*(h:x)*

Constants may be handled at the compilation time. Constants are part of the program statements.

4.2 Memory Allocation Scheme

A global stack is employed to perform the memory simulation. For primitive function calls, the referenced data object is copied first to a local memory. Then the data is erased from the global stack. After application of that particular function, the result is again stored in the global stack with a pointer passed back by the function. With this approach a simple program, consisting of only compositions of primitives, needs only a stack size equal to the maximum size of the intermediate result. For functional forms ***apply-to-all***, ***insert***, and ***associative insert*** the data object is copied to a local memory first, then erased from the global stack. For ***apply-to-all***, each of its element is put into global stack and passed to the function, which is the first operand of these functional forms for execution. After all has been executed, results are constructed to form the final result. For ***insertion***, a similar thing action is performed by copying the original to local memory, and erasing it from the stack. According to their semantic definitions, functions are evaluated and results put into the global stack. The functional forms ***construction*** and ***condition*** can be defined simply: *Construction* evokes its functions and collects their reference pointers. Results pointed to, are erased after they have been copied into the global stack. *Conditional* statement evaluates the predicate first. Its result is tested and erased from the global stack. The proper function is selected and evoked as composition. *Constant* requires only copying from the program store to the global stack.

4.3 Recursion

One might raise doubts on how the global stack can handle recursion. In the execution graph, recursion is a re-entry which forms a cycle. In CDF representation it is written as "fn:=fm <object label>". First the current working object pointer is saved in a stack. New object is obtained by calling the "object label", which is the second operand of the statement. This reference to the label is assigned as the object of the function "fm", which is evoked recursively. If this new object terminates the procedure, it returns the value and pops the stack to get back to the original object pointer and continues the program from there. If the new object does not terminate the recursive call, it repeats the previous steps by pushing the newly obtained object pointer into the stack and obtains another result by calling the object label with the second result as the object reference. In doing so the result of "object label" is not fixed as we might have thought, but changes its value every time a new call is issued. This is also the reason why *conditional* statement can not execute the two possible functions simultaneously with the predicate, this might result in infinite recursive calls.

Since FP has no global variables, there are no side effects. The global stack is sufficient to handle the recursion. During the run time this stack may grow very large, yet we do not have to retrieve any value in the stack. Thus the size of the stack has no effect on the efficiency of the algorithm used. That is, we do not need to search in the stack for a particular element.

CHAPTER 5

OPTIMIZATION CONSIDERATIONS

It is claimed that the functional style of programming improves programmability. It is also the general consensus that programs written in this style can be very inefficient ([WADL-81] [ISLA-81]). In order to make FP object codes efficient, code optimization is necessary. The environment which codes will be executed in governs the optimization consideration. In a multiprocessor environment, optimization considerations differ from the conventional compiler optimization which has a sequential environment. This environment described by finite state automata is quite close to the procedural language used to describe the solution of a problem. Since FP is different from the procedural language used, we need a different approach to the problem of optimization. It again depends on the environment which FP programs will be executed in.

In general there are two extremes in code optimization. At one end is true algorithm optimization. The compiler is to "understand" what a function defined by user is suppose to do and to produce a transformation of the user's program to solve the same problem on a target machine. This is difficult to accomplish with conventional procedural languages. Because of the side effects caused by the states and variables, it is difficult to break down a big program into smaller modules to be "understood" and "transformed".

One a
braic transf
equivalent t
ginal progr
these transf
the origina
allows the
tions to th
vided by t
hardware
hardware
generate
paramete

O
which ar
simpler
will not
local sul

Instead
into ob
object
differe
code t
cessor

"efficient",. An example of this approach is illustrated in the data flow language LAU [COMT-77]. In this language the construct "EXPAND" explores parallelism. When the compiler sees this construct it reads from the program a number provided by the user specifying how much parallelism is to be abstracted from that section of the code. Accordingly, manyp copies of this object code is generated. Although this expansion creates much more code, the code generated is faster for a multiple processor environment.

Since CDF represents the execution tree of a program written in functional programming language, it is natural to treat each statement or a block of statements as a task.

There are three possible levels of optimization in the FP environment. And they are:

(1) Functional Programming Level: transformation

(2) CDF Intermediate Form Level: elimination redundancy

(3) Object Code Level: code improvement

5.1 Functional Programming Level

Several people have worked on the transformation of FP programs using algebraic properties. ([WADL-81] [ISLA-81]) Here are some transformation rules which can be implemented:

(1) AP CM(f,g) = CM(AP f , AP g)

(2) AP[f1,f2,f3,...fn] = CM(TR,[APf1,APf2,APf3....APfn])

(3) AP IF(f,g,h) = CM(AP IF(SL1,SL2,SL3),TR,[APf,APg,APh])

Proofs are as follows:

(1) AP CM(f,g):x =? CM(APf , Apg) :x

case 1: x = "?" both sides get "?"

case 2: x = (x1, x2, x3, ... xN)

LHS = AP CM(f,g):x
= (CM(f,g):x1, CM(f,g):x2,....
CM(f,g:xN))
RHS = CM(APf,APg):x
= AP f:(APg:x)
= APf :(g:x1, g:x2, ... g:xN)
= (f:(g:x1),f:(g:x2), ... f:(g:xN))
= (CM(f,g):x1, CM(f,g):x2,
.... CM(f,g):xN)
= AP CM(f,g):x
= LHS

(2) AP[f1,f2,f3,...fn]:x =? CM(TR,[APf1,APf2,APf3 ... APfn]:x

case 1: x = "?" *OK*

case 2: x = (x1, x2, x3 ..... xN)

LHS = AP [f1,f2,....fn]:x
= ([f1,f2,...fn]:x1,....[f1,f2,...fn]:xN)
= ([f1:x1,f2:x1,...fn:x1],
.... [f1:xN,f2:xN,...fn:xN])

by taking the transpose we obtain:

([f1:x1,f1:x2,f1:x3,....fn:xN],...
[fn:x1,fn:x2,...fn:xN])

RHS = [APf1,APf2,....APfn]:x
= [APf1:x, APf2:x,....APfn:x]
= [(f1:x1,f1:x2,....),(f2:x1,f2:x2,...),...
(fn:x1,...fn:xN)]
= LHS

(3) AP IF(f,g,h):x =? CM(AP IF(1,2,3),TR,[APf,APg,APh]):x

case 1: x = "?" *OK*

case 2: x = (x1,x2,x3,....xN)

LHS = AP IF(f,g,h) :x
= (IF(f,g,h):x1,
IF(f,g,h):x2,....IF(f,g,h):xN)

and [APf,APg,APh] :x
= [APf:x,APg:x,APh:x]
= [(f:x1,f:x2,....f:xN),(g:x1,...g:xN),
(h:x1,...h:xN)]

take the transpose we get:

[(f:x1,g:x1,h:x1),(f:x2,g:x2,h:x2),...
(f:xN,g:xN,h:xN)]

RHS = AP IF(1,2,3):[(f:x1,..),
(f:x2,..),(f:xN,..)]
= [IF(f:x1,g:x1,h:x1),IF(f:x2,g:x2,h:x2),
IF(f:x3,g:x3,h:x3)]
= [IF(f,g,h):x1),IF(f,g,h):x2),
IF(f,g,h):x3)]
= LHS

The purpose of performing these transformations is to simplify the allocation task. We see that the *apply-to-all* functional form takes a function and applies it to every element of the input list. This form constitutes one of the most frequently used parallel construct of FP. It is rather difficult to allocate resources for a program with an *apply to all* of a complex function. Many subtasks have to be generated in order to be executed evenly by the hardware. It also has the problem of uneven execution time which may cause inefficiency. If we can find transformations to map complicated functions-which is being *applied-to-all*, to simple primitives, we would achieve a better run time efficiency. It also facilitates the optimization on the CDF level. This will be discussed in more detail in the next section.

5.2 CDF level

We have demonstrated in Chapter 3 that repeated statements can be eliminated without side effects. On first thought, this elimination process may seems to require exponential time, however the following intuitive algorithm is polynomial time.

```
While there are repeated statments do

Begin
   For i:=1 to n do begin
      For j:=i to n do begin
         if (statement[i]=statement[j]) then
           begin

             eliminate ith statement;
             perform necessary change on label;

             n <- n-1

           endif;
      endfor;
   endfor;

endwhile.
```

In the worst case the inner most loop requires O(n) time, since relabeling takes n comparisons to find out if there are any statements which refer to the eliminated one. The checking of repeated statements takes $O(n^2)$ worst case. Thus the total complexity is $O(n^4)$. A better solution is to sort the statements according to their primitives or functional forms first. This would take O(nlogn) time. After sorting, repeated statements could be clustered together and eliminated, any necessary relabeling is performed at the same time. The time complexity for relabeling is still $O(n^2)$. However

the average case takes only O(cn) time, (c being a constant). The value of "c" depends on the percentage of repeated statements. Elimination and relabeling would not change the order of the sorted list. As a result further elimination can be repeatedly performed until there are no repeated statements. Then the program is re-sorted according to statement numbers. To summarize we give a simple description as follows:

sort the list according to their function types;

while *there is repeated statements* **do**

  for *each type of functions*
    eliminate repeated statement;
    relabel all necessary references

endwhile

sort the list again according the the statement number.

We would like to obtain the total time complexity of this algorithm. First we need 2*nlogn to sort the list twice. If we assume the maximum portion of repeated statements is one half of the original program, for each repeated statement eliminated we need to examine n-1 statements to relabel all statements which might refer to this eliminated statement. The number of different functional forms and primitives is constant. Therefore we are bounded by a constant. Relabeling, then takes O(n) time in its worst case. So it takes a maximum of $O(cn^2)$ time to do elimination and relabeling. Adding these two type of requirements, the time bound is now $O(n^2)$. We see that this is not a high price to pay for the improvement we may obtain, since we estimate that about thirty percent of the CDF code may be repeated. We observe also that, as we perform transformations on the

functional language level, the number of repeated statements may increase. This is a result of breaking down functional forms into primitives. As the number of statements increases the repetitions also tend to increase.

5.3 Code Level

After an FP program is parsed into CDF we can translate it into the C language and run it on a VAX 11/780. This is done simply by macro substitution. Let us look at a simple example of how this translation from CDF to C is accomplished.

For example:

```
f0 : SL1 f1
f1 : SL2 .
```

can be easily thought as :

```
*char f0()
{
extern *char f1();
  return(select(1,f1()));
}
*char f1()
{
extern *char input;
  return(select(2,input));
}
```

The C compiler does its optimization on the C code. Depending on the target language the final code can be further optimized. If microprogramming code is employed to perform the primitive calls, it can be fine tuned through careful programming. This would only be a one-time job. Because primitives are used over and over again, the savings may be great.

CHAPTER 6

SOLVING ORDINARY DIFFERENTIAL EQUATIONS USING FP

The field of numerical problem solving can be divided into many classes. Some examples are linear and non-linear programming, ordinary differential equations, partial differential equations. In this chapter we exhibit the applicability of FP on a particular class: a continuous system simulation requiring the solution of a system of ordinary differential equations.

6.1 Generalized Model

A large class of engineering problems when expressed mathematically take on the form of systems of simultaneous ordinary differential equations [KARP-81].

$$
\begin{aligned}
a_{1,n}\frac{d^n y_1}{dt^n} + a_{1,n-1}\frac{d^{(n-1)} y_1}{dt^{(n-1)}} + \ldots + a_{1,1}\frac{dy_1}{dt} + a_{1,0}y_1 &= F_1(t, y_1, \ldots, y_m) \\
a_{2,n}\frac{d^n y_2}{dt^n} + a_{2,n-1}\frac{d^{(n-1)} y_2}{dt^{(n-1)}} + \ldots + a_{2,1}\frac{dy_2}{dt} + a_{2,0}y_2 &= F_2(t, y_1, \ldots, y_m) \\
&\vdots \\
a_{m,n}\frac{d^n y_m}{dt^n} + a_{m,n-1}\frac{d^{(n-1)} y_m}{dt^{(n-1)}} + \ldots + a_{m,1}\frac{dy_m}{dt} + a_{m,0}y_m &= F_m(t, y_1, \ldots, y_m)
\end{aligned}
\tag{6.1}
$$

6.2 General Approach in Solving the System of Equations

A common approach to solving these equations numerically is to transform these equations into a set of first order equations. This is sometimes called the "state variable" approach, in that a new set of variables is created. For example, the following equation :

$$a_n\frac{d^n y}{dx^n} + a_{n-1}\frac{d^{(n-1)}y}{dx^{(n-1)}} + ... + a_1\frac{dy}{dx} + a_0 y_1 = F(x,y) \tag{6.2}$$

can be rearranged into a set of first-order equations:

$$\begin{aligned}
\frac{dy_{(1)}}{dx} &= F_1(x, y_{(1)}, y_{(2)}, \cdots, y_{(n)}) \\
\frac{dy_{(2)}}{dx} &= F_2(x, y_{(1)}, y_{(2)}, \cdots, y_{(n)}) \\
&\vdots \\
\frac{dy_{(n)}}{dx} &= F_n(x, y_{(1)}, y_{(2)}, \cdots, y_{(n)})
\end{aligned} \tag{6.3}$$

where $y_{(2)}, y_{(3)}, \cdots\cdots y_{(n)}$ are a new set of state variables given as:

$$\begin{aligned}
y_{(1)} &\equiv y \\
y_{(2)} &\equiv \frac{dy_{(1)}}{dx} \\
y_{(3)} &\equiv \frac{dy_{(2)}}{dx} \\
&\vdots
\end{aligned} \tag{6.4}$$

$$y_{(n)} = \frac{dy_{(n-1)}}{dx}$$

(6.4)

By substituting these new state variables into (6.2) and rearranging (6.2) we obtain equation (6.5).

$$dy_{(n)} = \frac{1}{a_n} F\left(x, y_{(1)}, y_{(2)}, \ldots\ldots, y_{(n)} \right)$$

$$- \left(a_{n-1} y_{(n)} + a_{n-2} y_{(n-1)} + \ldots\ldots + a_0 y_{(1)} \right) \qquad (6.5)$$

Each of the remaining state variables is integrated numerically in term, using the solutions for the previously considered state variables. (6.5) we may obtain a equation for $dy_{(n-1)}$, and $dy_{(n-2)}$ from $dy_{(n-1)}$ and on. Applying this method to equation (6.1), we obtain:

$$\frac{dy_1^n}{dt^n} = \frac{1}{a_{1,n}} \left[F_1(t, y_1, \ldots, y_m) - a_{1,n-1} \left(\frac{d^{n-1} y_1}{dt^{n-1}} \right) \cdots - a_{1,0} y_1 \right]$$

$$\frac{dy_2^n}{dt^n} = \frac{1}{a_{2,n}} \left[F_2(t, y_1, \ldots, y_m) - a_{2,n-1} \left(\frac{d^{n-1} y_2}{dt^{n-1}} \right) \cdots - a_{2,0} y_2 \right] \qquad (6.6)$$

$$\vdots$$

$$\frac{dy_m^n}{dt^n} = \frac{1}{a_{m,n}} \left[F_m(t, y_m, \ldots, y_m) - a_{m,n-1} \left(\frac{d^{n-1} y_m}{dt^{n-1}} \right) \cdots - a_{m,0} y_m \right]$$

Integrating these we obtain:

$$\frac{dy_1^{n-1}}{dt^{n-1}} = \int_0^t \frac{1}{a_{1,n}}\left[F_1(t,y_1,\ldots,y_m) - a_{1,n-1}\left(\frac{d^{n-1}y_1}{dt^{n-1}}\right) \cdots - a_{1,0}y_1\right] + IC_1$$

$$\frac{dy_2^{n-1}}{dt^{n-1}} = \int_0^t \frac{1}{a_{2,n}}\left[F_2(t,y_1,\ldots,y_m) - a_{2,n-1}\left(\frac{d^{n-1}y_2}{dt^{n-1}}\right) \cdots - a_{2,0}y_2\right] + IC_2 \qquad (6.7)$$

$$\vdots$$

$$\frac{dy_m^{n-1}}{dt^{n-1}} = \int_0^t \frac{1}{a_{m,n}}\left[F_m(t,y_1,\ldots,y_m) - a_{m,n-1}\left(\frac{d^{n-1}y_m}{dt^{n-1}}\right) \cdots - a_{m,0}y_m\right] + IC_m$$

This integration process is performed until all state variables are obtained.

6.3 Using Functional Language

The need for hierarchical representation at the user definition level has become clear to us as well as to others [NILS-74] [BEKE-68] [KARP-77]. This hierarchical representation is achieved by separating the program into four blocks: a function block, a constant block, a flow block, and an input block. In general the program for solving a set of O.D.Es would look like:

```
/* FUNCTIONS */
    f1(x1, x2, .... xn) = .....
    f2( ... .
    .
    .
    .
    fn
    sum(w,x,y,z) = w + x + y + z
    .
    .
    .
```

```
/* CONSTANTS */

    const =
    .
    .
    .
    table1  = ( ( 0.1, ...), (...).. )
    table2  = ( ....
    .
    .
    .

/* FLOW */

    y1   = sum(F1, ..... );
    y2   = INT(y1);
    y3   = INT(y2);
    .
    .
    .
    next = [y1, y2, y3, ....];

/* INPUT */

    input := (IC1, IC2, ..... , h)
```

Each of these blocks is discussed in one of the following sections.

6.3.1 Function Block

This block contains the definitions given by the user. In our example, in equation (6.1) we need to evaluate:

$$\begin{aligned} &F_1(t, y_1, y_2, \cdots y_m) \\ &F_2(t, y_1, y_2, \cdots y_m) \\ &\quad . \\ &\quad . \\ &\quad . \end{aligned} \tag{6.8}$$

$$F_m(t, y_1, y_2, \cdots y_m)$$

The user can specify all of these functions in mathematical terms and the notation for the necessary function definitions in infix form. Any other functions which the user requires, such as summation or product can also be expressed here. The format of function definitions can be hierarchical. That is, a user is able to define a function, using other functions. These expressions can be transformed into prefix form. Optimization can be performed during this conversion from infix to prefix of the execution trees defined by these definitions of functions in order to obtain a minimum height tree. If we do not want to explore all possibilities, we use some heuristic method to improve the transformation of trees. Of course, block sharing should also be allowed; any identical subtrees can be shared.

6.3.2 Constant and Table Block

In solving this kind of simulation problems, we need constant scalings, constant additions, and table look-ups. These data are not part of the program input but rather part of the program.

6.3.3 Flow Block

Program statements are written in this block. It is to the user's advantage and compiler writer's advantage to allow the infix notation only. Some functions may need to be defined as primitives, such as iteration, SIN, COS, TAN etc. The functional form *composition* allows the user to express the ordering of the defined functions. All state variables within the *construction* functional form are calculated at the same time. If the program is exe-

cuted in a data-driven fashion, at any point of execution, all data necessary are ready for that particular evaluation. This style of execution allows the compiler to explore the parallelism among the different equations.

6.3.4 Input Block

This block contains the input vector, and any special conditions for the system. It may also contain information about how the program is to be applied. Other information included in this block, is related to hardware resources such as number of processors, memory size, etc. This is included to help the compiler to do pre-run time optimization and allocation.

6.4 An Example

The system we are modeling has two pendula and a spring. The spring is attached to both pendula to create feed-back effects. See figure 6.1 for the physical set-up. [KARP-81]

6.4.1 Mathematical Derivation

Two O.D.E.s are employed to model the system. For a single compound pendulum the second order differential equation is given by:

$$-WSsin(\theta) = \frac{W}{GL^2 \frac{d^2\theta}{dt^2}}$$

where
W is weight of the pendulum
G is gravitational constant
S is the length from pivot to center of gravity
L is total length of pendulum

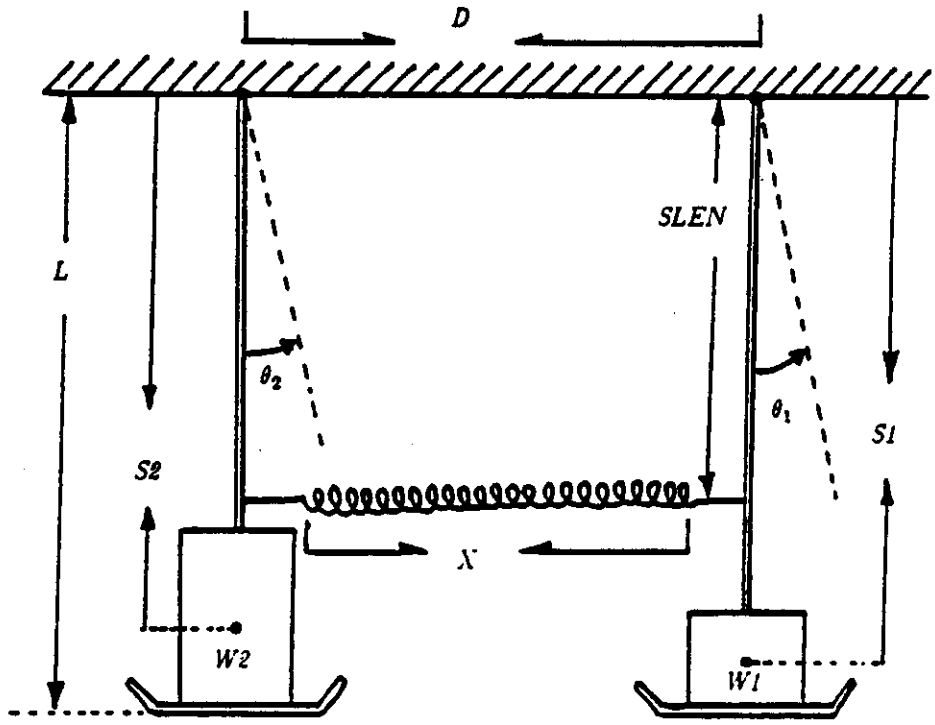


Figure 6.1 Physical set up of the coupled pendulums.

A friction term is assumed to be proportional to $d\frac{\theta}{dt}$. The total force acting on the single pendulum comprises : forces from gravity and from the spring. The force from gravity is already included in equation 6.9. We then have to introduce another term for the spring force. This force is proportional to the displacement of the spring *(F=kx)*.

Writing this model as differential equations we obtain:

$$M_1L^2\frac{d^2\theta_1}{dt^2} + M_1GS_1\sin\theta_1 + SLEN\,K\,(y-x) + \frac{Dd\theta_1}{dt} = 0 \qquad (6.10)$$

$$M_2L^2\frac{d^2\theta_2}{dt^2} + M_2GS_2\sin\theta_2 - SLEN\,K\,(y-x) + D\frac{d\theta_2}{dt} = 0 \qquad (6.11)$$

Each of the terms in figure 6.1 are defined as:

*W1 : M1*G (M1 : mass for the right pendulum)*
*W2 : M2*G (M2 : mass for the left pendulum)*
S1 : length from pivot to the center of gravity of right pendulum
S2 : length from pivot to the center of gravity of left pendulum
SLEN : Ceiling to Spring
D : the distance between two pendulum
X : is the original length of the spring
Y : is the length while in motion
MU : is the friction coefficient

Constants are obtained from standard tables. Physical measurements are actually taken to determine the length of the spring (X), the distance between two pendula (d), the length of the pendulum(L), and their weights ($W1, W2$). They are :

G=980
X=44 cm
L=59 cm
D=40 cm
K=1.5
M1=10 gm
M2=10 gm
SLEN=46.5 cm
S1=48 cm
S2=48 cm
MU=0.03

The O.D.Es derived are:

$$\frac{d^2\theta_1}{dt^2} + b_1\frac{d\theta_1}{dt} + F_1(\theta_1,\theta_2) = 0$$

$$\frac{d^2\theta_2}{dt^2} - b_2\frac{d\theta_2}{dt} + F_2(\theta_1,\theta_2) = 0 \qquad (6.12)$$

where;

$$F_1 = \frac{1}{a_1}\left(w_1\sin\theta_1 + SLEN\ K\ (Y-X)\right) \qquad (6.13.a)$$

$$F_2 = \frac{1}{a_2}\left(w_2\sin\theta_2 - SLEN\ K\ (Y-X)\right) \qquad (6.13.b)$$

and

$$Y = \sqrt{(D^2+2L(\sin(\frac{\theta_1-\theta_2}{2})-4LD(\sin(\frac{\theta_1-\theta_2}{2})\cos(\frac{\theta_1-\theta_2}{2})} \qquad (6.14)$$

Elements *a1, a2, b1, b2, w1, w2, X, D, L, SLEN, K,* ... are all constants.

$$a_1 = M_1 L^2 \tag{6.15.a}$$

$$a_2 = M_2 L^2 \tag{6.15.b}$$

$$b_1 = \frac{MU}{a_1} = \frac{MU}{M_1 L^2} \tag{6.15.c}$$

$$b_1 = \frac{MU}{a_2} = \frac{MU}{M_2 L^2} \tag{6.15.d}$$

$$w_1 = M_1 GS_1 \tag{6.15.e}$$

$$w_2 = M_2 GS_2 \tag{6.15.f}$$

Therefore, there is only one term on the left side :

$$\frac{d^2\theta_1}{dt^2} = -\left[b_1 \frac{d\theta_2}{dt} + F_1 \right] \tag{6.16.a}$$

$$\frac{d^2\theta_2}{dt^2} = \left[b_2 \frac{d\theta_2}{dt} - F_2 \right] \tag{6.16.b}$$

where

$$F_1 = \frac{GS_1}{L^2} \sin(\theta_1) + \frac{SLEN\ K\ (Y-X)}{M_1 L^2} \tag{6.17.a}$$

$$F_1 = \frac{GS_2}{L^2} \sin(\theta_2) + \frac{SLEN\ K\ (Y-X)}{M_2 L^2} \tag{6.17.b}$$

A mapping from this mathematical definition into an FP program is obtained more directly than obtaining a mapping into a conventional language program.

6.4.2 Programming in Functional Language

Let us convert this model into a graphical representation similar to the solution on an analog computer as shown in figure 6.2.

Each box can be easily defined in FP infix form. Let us use the Euler method to solve these integrations. We give the functional language specification in the following section and its graphical representation in Figure 6.3.

```
/* FUNCTIONS */

DDTH(b, DTH, f) = (-1)*(b*DTH+f);
  F1(theta1, theta2) = (G*S1/(L*L))*SIN(theta1)+
                       (SLEN*K*(Y-X))/(M1*L*L);
  F2(theta1, theta2) = (G*S2/(L*L))*SIN(theta2)-
                       (SLEN*K*(Y-X))/(M2*L*L);

  /* WHERE */

      Y(theta1, theta2) = SQRT(D*D+(SIN((theta1-theta2)/2)*2*L)**2
                  -2*D*(SIN((theta1-theta2)/2)*2*L)*
                   COS((theta1-theta2)/2);


  B(mu,m,L) = mu/(m1*L**2)
  INT(d,ic,h) = d*h +ic

/* CONSTANT */

  G = 980.0
  X = 44.0
  L = 59.0
  D = 40.0
  SLEN = 46.5
  M1 = 10.
  M2 = 10.
```

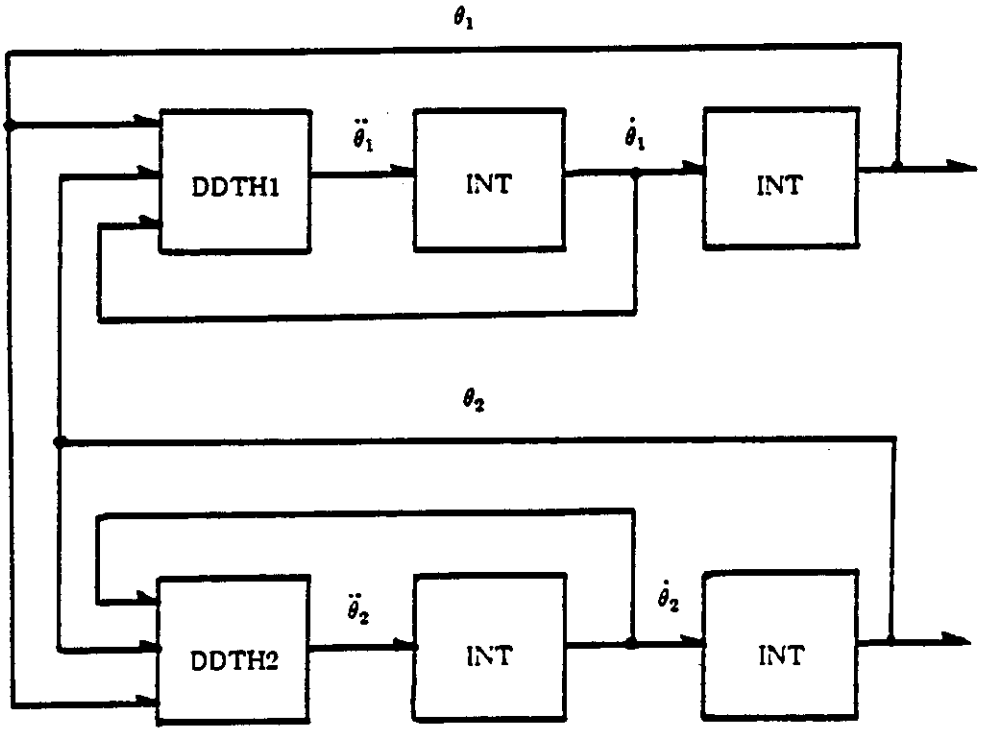


Figure 6.2 Block Solution of the Example

```
S1 = 48.0
S2 = 48.0
MU = .03
K = 1.5

/* MAIN PROGRAM */

/* FLOW */

main = IF(timeup, next, CM(main, [next, h, Tminush]));

  next = [ [nexttheta1, nextdtheta1], [nexttheta2, nextdtheta2] ];
  timeup = CM(LT,[SL3, K0]);

                h = SL2;
                T = SL3;

  nextdtheta1 = CM (INT , [ddtheta1,dtheta1,h]);
  nextdtheta2 = CM (INT , [ddtheta2,dtheta2,h]);
  nexttheta1 = CM (INT , [dtheta1,theta1,h]);
  nexttheta2 = CM (INT , [dtheta2,theta2,h]);

     theta1 = CM(SL1,SL1,SL1);
     theta2 = CM(SL1,SL2,SL1);

     dtheta1 = CM(SL2,SL1,SL1);
     dtheta2 = CM(SL2,SL2,SL1);

     ddtheta1 = CM(DDTH1.[b1, dtheta1, f1);
     ddtheta2 = CM(DDTH2.[b2, dtheta1, f2);


/* calculate f1, f2  and b1, b2 */

        f1 = F1(theta1, theta2);
        f2 = F2(theta1, theta2);
        b1 = B(MU, M1, L);
        b2 = B(MU, M2, L);
        ;
/* END */

/* INPUT */

   (((0.1, 0), (0, 0)), 0.01)
```

This program is analyzed using the Berkeley version of the FP dynamic trace and static package [BADE-83]. The number of specific operations, the maximum number of operations per iteration, and the use of

constants are given in the following tables.

Function	Total	Max./step	Average/step
+	198	4	2.25
-	263	7	2.99
*	880	0	5
/	264	0	6
sin	132	6	3
cos	44	2	2
sqrt	44	2	2
power	88	4	4
all arith.	1605	23	4.56
all	1913	25	4.83

Table 6.1 Parallelism of Operations

We can also study the usage of constants to determine where and when to put the value of the constant into the system.

Constant	Total	Max./step	Average/step
2	352	16	16
-1	22	1	1
0	22	1	1
G	44	2	2
X	44	2	2
L	308	14	14
D	132	6	6
SLEN	44	2	2
K	44	2	2
M1/M2	88	4	4
S1/S2	44	2	2
MU	44	2	2
ALL	1210	50	3.75

Table 6.2 Frequency of Constants in the Example

This information gives us a preliminary feedback on the maximum parallelism which can be obtained for a particular problem assuming a given hardware organization. We observe that with the maximum parallelism obtainable we still need 16 serial steps.

constants are given in the following tables.

Function	Total	Max./step	Average/step
+	198	4	2.25
-	263	7	2.99
*	880	0	5
/	264	0	6
sin	132	6	3
cos	44	2	2
sqrt	44	2	2
power	88	4	4
all arith.	1605	23	4.56
all	1913	25	4.83

Table 6.1 Parallelism of Operations

We can also study the usage of constants to determine where and when to put the value of the constant into the system.

Constant	Total	Max./step	Average/step
2	352	16	16
-1	22	1	1
0	22	1	1
G	44	2	2
X	44	2	2
L	308	14	14
D	132	6	6
SLEN	44	2	2
K	44	2	2
M1/M2	88	4	4
S1/S2	44	2	2
MU	44	2	2
ALL	1210	50	3.75

Table 6.2 Frequency of Constants in the Example

This information gives us a preliminary feedback on the maximum parallelism which can be obtained for a particular problem assuming a given hardware organization. We observe that with the maximum parallelism obtainable we still need 16 serial steps.

6.5 Discussion

From the above example we observe that there is a close relation between the mathematical derivation and the functional programming technique. It would be more natural and intuitive for engineers and scientist in FP than in FORTRAN or other procedural languages. Because the graphical representation for the model and the FP solution are similar, it is easier to program in FP than in conventional procedural languages.

CHAPTER 7

CONCLUSION

Functional style of programming offers many advantages. It provides an alternative in solving problems and utilizing concurrent hardware resources. Backus' functional programming language has been used as the base for many unconventional architectures. Most of these architectures, however, are based upon the reduction scheme of execution. This thesis discusses a way of decomposing functional languages into an intermediate form which can then be executed both in a demand-driven or in a data-driven fashion. This intermediate form, called Complete Decomposed Form (CDF) may not necessarily be the only intermediate form into which FP can be translated, but it provides a natural method of extracting the sequential aspects implied by the *composition*. It provides not only an intermediate level between the FP language and the hardware resources but also a bridge between two extremes of machine model - sequential conventional machine and complete asynchronous parallel scheme.

In this thesis we experimented with translation of FP programs into CDF and CDF into C codes. A simple example of a real system is modeled and solved with functional language. There are basically three levels of parallelism that can be exploited in this application. The first is in the simultaneous equations. Each equation can be executed separately, exchanging parameters after each cycle. The second level of parallelism is in the function

evaluation. The last level is in the intergration algorithms. In this application we restricted the method of execution to the parallelism inherent in the functional form *construction*. Parallelism in functional forms *apply-to-all* and *associate insert* have not been taken advantage of.

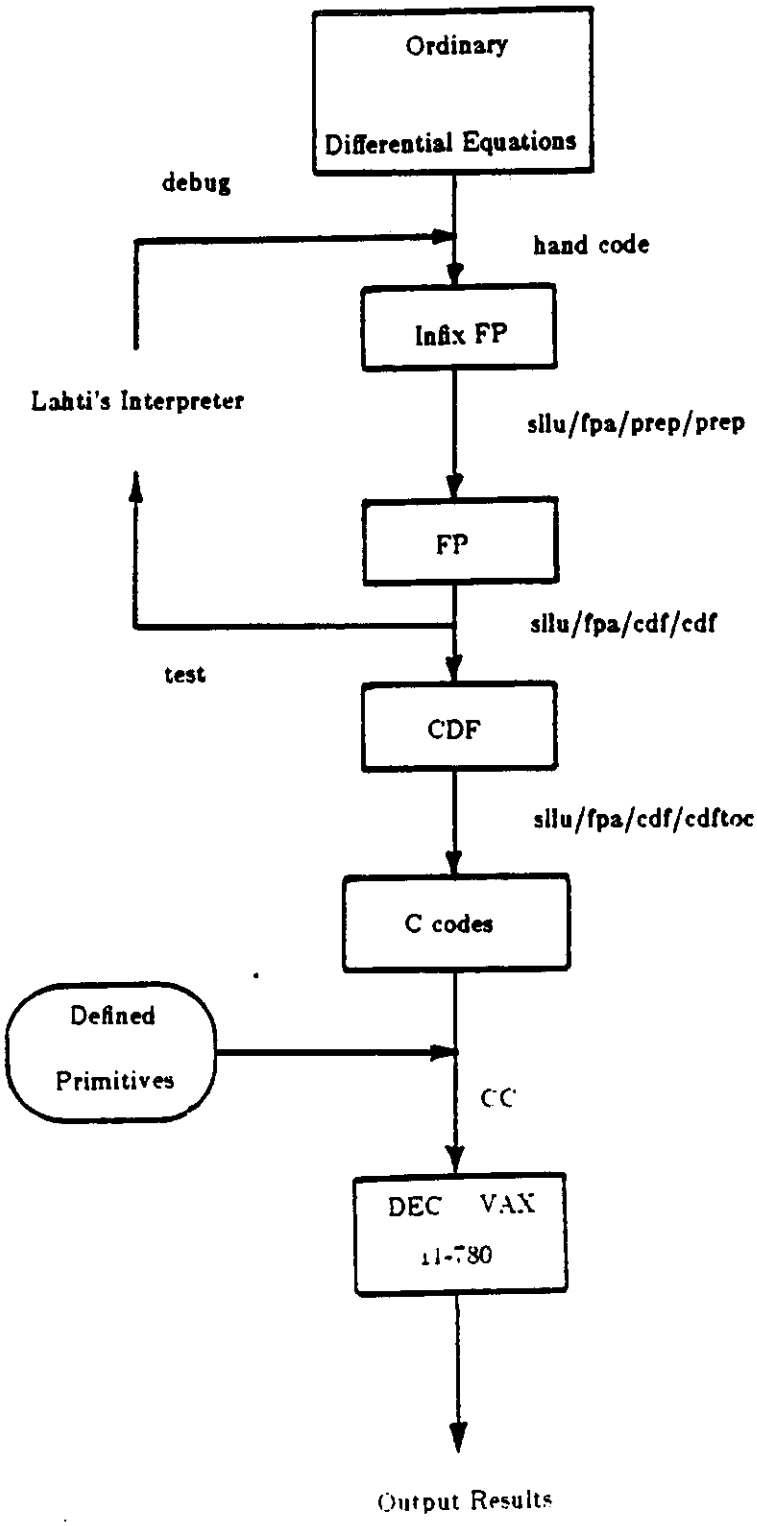
During the process of writing this translator, we also observed that the optimization is different from conventional compiler optimization. There are three possible levels of optimization in the FP compiler environment. On the FP level, algebraic transformation provides global modifications to the FP program without a full understanding of the algorithm. On the CDF level, space compaction is performed by the elimination of common subtasks. On the C level, conventional compiler optimization techniques can be used.

A more complicated benchmark could be studied to experiment this methodology. The complexity of the specification, the parallelism inherited from the problem, the speed-up acquired with this approach and the overhead needed should provide us an understanding to the further study of using FP in high speed simulation.

Other methods of decomposing FP programs should be investigated and their implications should be studied. Task allocation is another subject needed to be investigated further. Given a type of intermediate form of FP, allocating these codes into separate processors requires a knowledge of program behavior and hardware resources. In particular, the communication cost between tasks should be considered.

APPENDIX A

The Methodology of Approach


Ordinary
Differential Equations
debug
hand code
Infix FP
Lahti's Interpreter
sllu/fpa/prep/prep
FP
sllu/fpa/cdf/cdf
test
CDF
sllu/fpa/cdf/cdftoc
C codes
Defined
Primitives
CC
DEC VAX
11-780
Output Results

APPENDIX B

Brief Specification of Language

This appendix includes an informal description of the primitives and functional forms employed by the compiler.

SLk selects the k'th element of a vector.
PK picks the k'th element out of a vector.
ID returns the object itself.
TL returns the tail of a vector, (all but head).
FR returns the front of a vector, (all but last element).
LA returns the last object of a vector.
IX returns the first permutation of n integers.
AT is true if the object is an atom, false if not.
NL is true if the vector is the null sequence, false if not.
LN returns the length of the vector.
+ returns y+z. (integer arithmetic)
- returns y-z. (integer arithmetic)
* returns y*z. (integer arithmetic)
/ returns y/z. (integer arithmetic)
#+ returns y+z. (floating point arithmetic)
#- returns y-z. (floating point arithmetic)
#* returns y*z. (floating point arithmetic)
#/ returns y/z. (floating point arithmetic)
#SIN returns sin(y). (floating point arithmetic)
#COS returns cos(y). (floating point arithmetic)
#TAN returns tan(y). (floating point arithmetic)
#EXP returns exp(y). (floating point arithmetic)
#POW returns y^z. (floating point arithmetic)
#LOG returns log(y). (floating point arithmetic)
#SQRT returns sqrt(y).(floating point arithmetic)
& returns y AND z.
| returns y OR z.
! returns NOT x.
TR returns the transpose.
EQ is true if y = z, false if not.
GT is true if y > z, false if not.
LT is true if y < z, false if not.
DL distributes left element to all vectors in x.
DR distributes right element to all vectors in x.

AL appends first element to front of vector.
AR appends last element to end of vector.
CT concatenates vectors to form one vector.
PR forms vector into vector of pairs.
SP splits a vector into two halves.
RL rotates the vector circularly to the left.
RR rotates the vector circularly to the right.
CL circularly rotates vector k times to left.
CR circularly rotates vector k times to right.
RV reverses the order of objects in a vector.
CM routes outputs of function into the input of the next.
[] forms vector of results of many functions applied to x.
IF conditionally returns the result of one of two functions.
Ky returns y.
IN recursively applies a function into a sequence x.
AI recursively applies a function into binary tree x.
AP applies the function to all objects in x.

REFERENCES

[ACKE-79] Ackerman, W., "Data Flow Language", *Proc. NCC,* 1979, p.1087-1095

[ARVI-79] Arvind, K.P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine", *Dept. Information and Computer Science, University of California at Irvine,* TR 114a, 1978.

[ARVI-80] Arvind, K. P. "Decomposing a Program for Multiple Processor Systems", *1980 Intl. Conf. on Parallel Processing,* 1980.

[ARVI-81] Arvind, K. P. and Kathail, V., "A Multiple Processor Data Flow Machine That Supports Generalized Procedures", *Proceeding of Symp. on Comp. Architecture 1981,* 1981, pp. 291-302.

[ARVI-82] Arvind, K. P. and Gostelow, "The U-Interpreter" *Computer,* February 1982, pp. 42-49.

[BACK-78] Backus,J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *CACM,* Vol.21, No.8, August 1978, pp.613-641.

[BADE-83] Baden, S and Patel, D., "Berkeley FP - Experiences with a functional Programming Language". *Proceeding COMPCON* Spring 1983. pp.274-277.

[BEKE-68] Bekey, G. A. and Karplus, W. J., *Hybrid Computation,* John Wiley and Sons, New York, 1968.

[BERK-75] Berkling, K. J., "Reduction Languages for Reduction Machines", *Proceeding of 2nd Annual Symp. on Computer Architecture,* Jan. 1975, pp. 133-140.

[COMT-77] Comte, D. and Hifdi, N., "LAU Multiprocessor; Microfunctional Description and Technical Choices" *C.E.R.T*

[DARE-81] Darlington, J. and Reeve, M. "ALICE" *Proceeding of the 1981 Conf. on Functional Programming Languages and Computer Architecture,* Portsmart, New Hampshire

[DENN-79] Dennis, J.B., "The Varieties of Data Flow Computers", *Proceedings of International Conference on Distributed Systems,* Huntsville, Alabama, 1979.

[ERCE-81] Ercegovc, M. D., Private Communication, *UCLA,* 1981

[FELL-81] Feller, M. CS259 Seminar Term Project Report, UCLA Computer Science Dept., 1981

[FLYN-79] Flynn, M.J. and J.L. Hennessy, "Parallelism and Representation Problems in Distributed Systems", *Proc. Intl. Conf. on Distributed Systems,* Huntsville, Alabama, 1979.

[ISLA-81] Islam, N. Myers, T.J. and Broome, P. "A Simple Optimizer for FP-like Languages" *Proceeding of the 1981 Conf. on Functional Programming Languages and Computer Architecture, Portsmart, New Hampshire*

[JENS-74] Jensen, K. and Wirth N., "PASCAL User Manual and Report" 2nd Edition, *Spring-Verlag,* N.Y. Heidelberg, Berlin

[JOHN-75] Johnson, S. C., "YACC: Yet Another Compiler Compiler," Computer Science Technical Report No. 32, *Bell Laboratories,* Murrey Hill, NJ, July 1975.

[JOHN-77] Johnson, D., "Automatic Partitioning of Programs in Multiprocessor Systems", *IEEE Compcon Proceedings,* Spring 1980.

[KARP-77] Karplus, W.J., "Peripheral Processors for High-speed Simulation", *Simulation,* Vol.31, November 1977.

[KARP-81] Karplus, W.J., "CS271a Class Notes" *UCLA Computer Science Department,* 1981

[KELL-79] Keller, R. M., Lindstrom, G. and Patil, S., "A Loosely Coupled Applicative Multi-processing System", *Proceeding of NCC,* 1979, pp. 861-870.

[KERN-78a] Kernighan, B. W., "UNIX for Beginner," 1978.

[KERN-78b] Kernighan, B. W. and D. M. Ritchie, "The C Programming Language," *Prentice-Hall,* Englewood Cliffs, NJ, 1978.

[LAHT-81] Lahti, D. "Applications of Functional Language", Master Thesis, 1981 *UCLA Computer science Report.*

[LESK-75] Lesk, M. E. and E. Schmidt, "Lex: Lexical Analyzer Generator.", *Bell Laboratories,* Murray Hill, NJ, 07974, July 1975.

[MAGO-80] Mago, G. A., "A Cellular Computer Architecture for Functional Programming", *Proceeding of COMPCON Fall 1980,* pp. 179-187.

[MCGR-79] McGraw, J.R., "Data Flow Computing: Software Development", *Proc. Intl.Conf. .on Distributed Systems,* Huntsville, Alabama, 1979.

[NILS-74] Nilsen, R.N. and W.J. Karplus, "Continuous System Simulation languages", *Annals,* International Association for Analog Computation, Vol.16, January 1974.

[PATE-81] Patel, D. "Ring Architecture for Applicative Programming". Master Thesis, 1981 *UCLA Computer science Report.*

[PLAS-76] Plas, A. et al., "A Parallel Data Driven Processor Based on Single Assignment", *1976 Intl. Conf. on Parallel Processing,* 1976.

[RAVI-83] Ravi, T. M. and Konstantinovic, Zoran, "Task Allocation of FP", CS259 Term Project Report *UCLA,* 1983.

[THOM-78] Thompson, K. and D. M. Ritchie, "Unix Programmer's Manual," *Bell Laboratories,* Seventh Edition, 1978.

[TREL-80] Treleaven, P. C. and Hopkins, R. P. "Decentralized Computation" Technical Report Series Number 157, *University of newcastle upon Tyne, England* November, 1980, Editor, Elphick.

[TURN-79] Turner, D. A. "A New Implementation Technique for Applicative Language" *Software-Practice and Experience,* Vol. 9, 31-49, 1979, pp. 31-49

[WADL-81] Wadler, P. "Applicative Style Programming, Program Transformation, and List Operators", *Proceeding of the 1981 Conf. on Functional Programming Languages and Computer Architecture,* Portsmart, New Hampshire

[WAHL-77] Wahlstrom, B., and Juslin, K., "Simulation with Hard-wired Analog Subprograms" *Simulation,* Vol.30, April 1977.

[WATS-82] Watson, I., and Gurd, J. "A Practical Data Flow Computer" *Computer,* February 1982, pp. 51-57.

[WILL-81] Williams, J. H. "On the Development of the Algebra of Functional Programs, REPORT RJ2983, *IBM Research Laboratory,* San Jose