T
D65-90
PAD

DESIGN OF A FUNCTIONAL COMPUTATION MODEL FOR MULTIPROCESSOR ARCHITECTURE

A THESIS

submitted in fulfilment of the
requirements for the award of the degree
of
DOCTOR OF PHILOSOPHY
in
ELECTRONICS AND COMPUTER ENGINEERING

By

PADAM KUMAR

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
UNIVERSITY OF ROORKEE
ROORKEE-247 667 (INDIA)

September, 1990

Dedicated

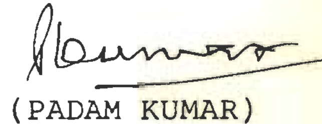
to

my Parents

CANDIDATE'S DECLARATION

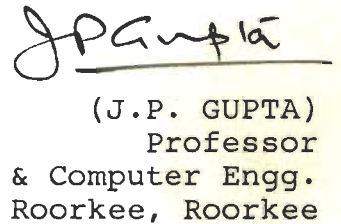
I hereby certify that the work which is being presented in the thesis entitled **Design of a Functional Computation Model for Multiprocessor Architecture** in fulfilment of the requirement for the award of the Degree of Doctor of Philosophy submitted in the Department of Electronics and Computer Engineering of the University is an authentic record of my own work carried out during a period from September 1987 to August 1990 under the supervision of Dr. J.P. Gupta.

The matter embodied in this thesis has not been submitted by me for the award of any other Degree.



(PADAM KUMAR)

This is to certify that the above statement made by the candidate is correct to best of my knowledge.



Date: Sept. 4, 1990

(J.P. GUPTA)
Professor
Dept. of Electronics & Computer Engg.
University of Roorkee, Roorkee

The Ph.D. viva-voce examination of Sri Padam Kumar Research Scholar has been held on 22 Nov. 1990.

(Signature of Guide) (Signature of External Examiner)

ABSTRACT

Functional languages, due to their straightforward declarative way of expression, are finding favour as an elegant programming medium. Further, their properties of referential transparency and freedom from side-effects make them highly attractive for programming multiprocessor systems. These merits are pushing them more and more into the domain of computer research directed towards achieving higher execution speeds and/or increasing programming comfort. The research has generated radically different computation systems called reduction computers.

This thesis reports the design of a multiprocessor computation model based on the functional approach. The various features that the model supports include pattern-matching, data structures, lazy evaluation of conditional expressions, and recursion. The complete design consists of two phases : first, the translation of program definitions into an intermediate form suitable for machine interpretation, and second, the evaluation of the translated program through reduction in a multiprocessor environment.

Program input to the model is a set of supercombinator definitions and an expression for evaluation. The definitions are like user defined functions having no fixed reduction rules, and have been compiled down to an intermediate representation called Structured Director String (SDS) term. The terms express the supercombinator definition bodies as variable-free annotated graph structures and are used as templates for function instantiation. They are a generalisation of Kennaway & Sleep's DS terms, and are obtained through a pattern-abstraction process for which an algorithm has been developed and tested.

For dealing with local definitions within definitions, the SDS term notation has been enriched by including pointers and a concept of context-list. Various types of local definitions (non-recursive, recursive and mutually recursive) have been interpreted via lambda calculus into the enriched notation.

A coarse-grain, message-passing multiprocessor reduction scheme has been proposed. SDS term reduction rules, which are based on a modified set of ß-reduction rules framed for dealing with structured arguments, have been developed and are used during template-instantiation of a function. The reduction strategy is basically applicative (eager) so as to exploit parallelism, wherever possible.

For reduction, a program expression is organised into a task-graph where each task is the smallest unit of computation consisting of a function applied to all its arguments. Tasks in task-graph reduce (as per the proposed conditions of reducibility) and communicate through messages enabling other tasks to reduce. The

process continues till the result of the expression is obtained.

Several control mechanisms, in the basic scheme, have been incorporated to support selective laziness in dealing with infinite data structures, lazy evaluation of conditionals, and controlled recursion through the fixed-point combinator. Safety aspect of the applicative order has been improved to some extent by leaving a sub-expression in unorganised form, although no strictness analysis of functions has been performed.

The complete reduction strategy, the organisation algorithm and the message handling schemes have been formally specified in a Pascal-like notation.

ACKNOWLEDGEMENTS

I wish to express my gratitude to my supervisor, Prof. J.P. Gupta, for offering guidance, help, encouraging comments and useful suggestions throughout.

The co-operation and help extended by Prof. R. Mitra, the Head of the Department of Electronics and Computer Engineering, is gratefully acknowledged.

I am indebted to Dr. Surendra Kumar, Reader Chemical Engineering Department and Prof. R.K. Gupta, Mathematics Department for some useful and critical discussions and friendly advice. I have a word of special thanks to Prof. Kailash Chandra, Director USIC, for constant encouragement.

I am also indebted to Prof. D.R. Wilson and Dr. S.C. Winter at Polytechnic of Central London, London, for providing useful suggestions and comments.

Finally, I am grateful to all my colleagues and friends here for their friendly co-operation and encouragement.

CONTENTS

ABSTRACT (i)

ACKNOWLEDGEMENTS (iii)

CONTENTS (iv)

LIST OF SYMBOLS AND ABBREVIATIONS (viii)

1 INTRODUCTION 1

1.1 INTRODUCTION 1

1.2 FUNCTIONAL BASED COMPUTATION 4

1.3 STATEMENT OF THE PROBLEM 7

1.4 ORGANISATION OF THE THESIS 9

2 FUNCTIONAL PROGRAMMING AND LAMBDA CALCULUS 11

2.1 FUNCTIONAL PROGRAMMING 12

2.2 LAMBDA CALCULUS 17

2.2.1 Lambda Syntax 17

2.2.2 Lambda Semantics 19

2.2.3 Reduction Order 21

2.2.4 Recursion 23

3 REVIEW OF COMPUTATION MODELS 26

3.1 INTRODUCTION 26

3.2 REDUCTION COMPUTER 28

3.2.1 Program Representation 30

3.2.2 Reduction Options 31

3.2.3 Intermediate Forms As Machine Languages 33

3.2.3.1 Combinators 34

3.2.3.2 Director string terms 35

3.2.3.3 Supercombinators 37

3.3 SUPERCOMBINATOR REDUCTION MACHINES 41

3.4 THE PROPOSED MODEL 43

4 IMPLEMENTATION OF PATTERN-MATCHING : A LAMBDA CALCULUS BASIS 45

4.1 PATTERN-MATCHING 46

4.2 LAMBDA CALCULUS INTERPRETATION 48

4.2.1 Matching Algorithm 52

4.2.2 Reduction Semantics Of $\lambda p.E$ 55

4.3 CONCLUDING REMARKS 59

5 COMPILATION OF PATTERN-MATCHING DEFINITIONS 61

5.1 COMPILATION PROCESS 62

5.1.1 Compilation For Clause Selection 62

5.1.2 Compilation For Reduction 65

5.1.2.1 Atomic directors 65

5.1.2.2 Structured directors 67

5.1.2.3 Pattern-abstraction 72

5.1.2.4 Pattern-abstraction rules 76

5.2 AN EXAMPLE 81

5.3 GUARDED EQUATIONS 85

5.4 CONCLUDING REMARKS 87

6 COMPILATION OF LET AND LETREC DEFINITIONS 89

6.1 DEALING WITH LOCAL DEFINITIONS 90

6.2 ADDITIONAL ABSTRACTION SCHEMES 92

6.3 COMPILATION PROCESS 96

6.4 D-CODE STRUCTURE 105

6.5 CONCLUDING REMARKS 109

7 MULTIPROCESSOR REDUCTION 112

7.1 SDS TERM REDUCTION 113

7.2 TASK STRUCTURE 116

7.3 ORGANISATION OF PROGRAM EXPRESSION 122

7.3.1 Organisation Algorithm 126

7.4 TASK REDUCTION 134

7.4.1 Task Reducibility 136

7.4.2 Reduction Mechanism 137

7.4.2.1 Application 138

7.4.2.2 Organisation 140

7.4.2.3 Communication 140

7.4.2.4 Removal 141

7.4.3 Message Handling 142

7.4.3.1 Result message 143

7.4.3.2 Link message 147

7.4.3.3 Argument (arg) message 149

7.5 MODIFICATIONS 152

7.5.1 Laziness 153

7.5.2 Simple Recursion 160

7.5.3 Mutual Recursion 164

7.5.4 Pattern-matching 166

7.6 CONCLUDING REMARKS 169

8 CONCLUSIONS 172

8.1 CONCLUSIONS 172

8.2 RECOMMENDATIONS FOR FUTURE WORK 180

REFERENCES 182

APPENDIX 191

RESEARCH PAPERS OUT OF THE WORK 203

LIST OF SYMBOLS AND ABBREVIATIONS

$\tilde{a}$	$(a_1, a_2, \ldots, a_n)$, a n-tuple structured variable
$@_1$	Unary application node-type
$@_2$	Binary application node-type
A_x	A-scheme abstraction operator for abstracting out x
B_x	B-scheme abstraction operator for abstracting out x
c	Constructor function
C	Constant
CN	Computability number
d	Director
d_1	unary director
d_2	binary director
d_r	r-ary director
/\	binary director for bothways
/	binary director for left
\	binary director for right
-	binary director for discard
!	unary director for substitution
#	unary director for discard
■	hole
{d}	pattern director
$\{d_1\}^k$	list director made up of k number of d_1 directors
dA_x	director resulting from the abstraction through A_x
D	Director string
D_1	director string of unary directors
D_2	director string of binary directors
D_r	director string of r-ary directors
DST	Director String Term
e, e_i, E	Expression
$\in$	belongs to (in sets)
$\notin$	does not belong to (in sets)

p, p_i	pattern
Ir	Irrefutable pattern-type
s	sum-constructor for a pattern
t	product-constructor for a pattern
$T(p_i)$, τ_i	type of pattern p_i
SDST	Structured Director String Term
t, t_i	SDS term
$\bigcup_{i=1}^{r} A_i$	$A_1 \cup A_2 \cup \ldots \cup A_r$ (set union)
w, w_i	Task
D	dummy task-type
E	executable task-type
P	partial task-type
W	waiting task-type
N	name-field in task structure
A	ancestor field in task structure
N_A	name of an ancestor A
F	function-field in task structure
S(I)	Ith successor of a task
SC	successor count field in task structure
$	Identifying symbol as first character in a supercombinator name
^	Pointer
≡	Syntactic equality

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Despite the extraordinary progress in computers, the basic model of computation (von Neumann architecture [1]) has not changed much over the past 40 years. In its most simplified form, the model consists of a single processor with a single connection to the memory store (Fig. 1.1). While executing a program, the machine passes instructions and data, one word at a time, through this single path between the processor and memory.

Quest for higher and higher computation speed has been a main endeavor of computer research. In von Neumann model, improvements in speed have been possible only by faster data transfer through the single connection which, very appropriately, has been referred as von Neumann bottleneck by Backus [2]. The attempts can be seen in the development of faster and faster semiconductor technologies, the use of registers and cache memories (which limit the number of memory accesses and thus relieve traffic in the bottleneck), and wider instruction buses. But the advances in semiconductor technology are fast approaching theoretical limits of speed whereas the growth of demands from scientific and engineering applications is showing no signs of saturation.

A rather obvious alternative in the situation is to employ several processors together. The idea has been made more feasible, both economically and technically, with the advent of VLSI technology

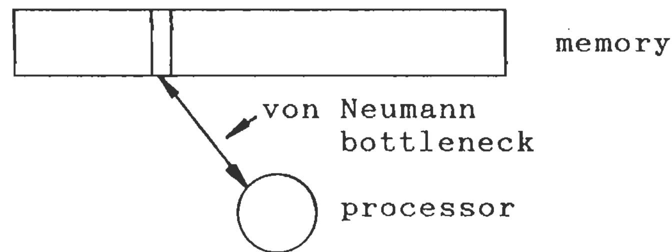


Figure 1.1 The von Neumann computer

which has reduced the cost-status of processors to that of circuit components [3]. Initial attempts in the direction of multiprocessing have been limited to applying some clever engineering extensions to the von Neumann model, e.g., pipelining (where several instructions are inserted in a pipeline of processors, each executing a part of an instruction), vector processing (where same instruction is executed in different processors on different data - Single Instruction Multiple Data (SIMD) approach [4]) and multiprocessing (where different instructions on different data can be executed asynchronously by each processor - Multiple Instruction Multiple Data (MIMD) approach). It may be observed that by the same terminology, von Neumann model is a Single Instruction Single Data (SISD) machine. The first two approaches have given rise to some powerful computers such as CRAY [5], ICL's DAP [6], Burrough's Scientific Processor (BSP) [7] etc..

The MIMD approach to parallelism is the most generalised and flexible one and seems potentially capable of exploiting the gains of VLSI technology. This approach, however, when realised using the von Neumann model and, rather more importantly, when programmed using conventional languages (FORTRAN, Pascal etc.), leads to severe difficulties of coordination and communication between processors which is necessary for ensuring that different parts of the program are executed in the prescribed sequence and that there are no unanticipated side effects. Although a number of newer programming languages such as Ada, Occam allow organising programs into relatively independent communicating processes, but the programmer is required to generate parallelism and indicate it explicitly. Thus the features may be useful for handling small amounts of parallelism on a gross level but to ask the programmer to specify large scale parallelism and take care of coordination among thousands of processes is too much to expect from human capability to handle complexity.

The challenge of multiprocessing requires seeking alternatives to von Neumann concepts both in hardware and languages. In the last decade, research in computer architectures has given rise to a novel class of computing systems. The principal stimuli for these have come from the pioneering work on data flow by Dennis [8,9] and on reduction machines by Berkling [10]. These architectures are inherently parallel and there is no concept of sequencing through program counter. Similarly, on the software front, there has been a growing interest in a new class of programming languages called declarative languages which are naturally

compatible with parallel processing and do not depend on the programmer to specify parallelism. Removing the concept of step-by-step sequential execution, they allow the programmer to think declaratively (specifying what is to be done) rather than imperatively (specifying precisely how a task is to be done).

Functional languages, forming a sub-group of the declarative group, have a mathematical base and possess the useful property of referential transparency which means that the meaning of an expression depends on the meaning of its component sub-expressions only and not on any history of the computation. These languages do not suffer from any side effects and lend themselves naturally to a parallel mode of evaluation. Interestingly, their development initially was not prompted by the special needs of parallel processing but by a desire to do something about the lamentable state of imperative programming [2]. Subsequently, the interest in their use in parallel processing, through the development of efficient implementation schemes using reduction architectures, has been constantly increasing [11].

1.2 FUNCTIONAL BASED COMPUTATION

A program in a functional language consists of **function definitions** and **an expression** for evaluation where the definitions act as user defined reduction rules. Expressions in this type of language have a natural representation as tree/graph structures because they are mainly built through the binary operation of applying an operator to an operand. These expressions can be evaluated through graph transformation steps called reductions,

the whole process being called graph reduction. The order of reductions, which can be parallel too because there are no side effect causing constructs, is not required to be specified by the programmer.

For implementation purposes, it is necessary to translate the higher level functional program into an intermediate language suitable for interpretation by a machine. Lambda calculus [12] is a mathematical tool of great relevance in this context and it acts as the basis of all other intermediate forms used. Although its simple syntax can represent all features of functional languages, yet its direct use as an intermediate machine language is not so efficient due to the presence of free variables. Further, the process of β-reduction (substituting an argument for each occurrence of a formal parameter) is slow due to the need to visit every leaf (in the tree of a function body) to look for an occurrence of the formal parameter (bound variable). To overcome these inefficiencies, some variants of lambda calculus such as SK-combinators [13], Director String Terms (DST) [14], Super-combinators [15] etc., have been developed. The use of SK-combinators converts an expression into variable-free form eliminating the problem of free variables. It also helps in simplifying graph reduction due to fixed reduction rules for combinators. DST (simply an alternative form of SK-combinators) is also a variable-free representation. Instead of guiding an argument to its destination in a function body through reduction of combinators, the DST attaches directing symbols, called director strings, at each node of a function tree. An argument

coming in for substitution against a variable is guided by these at run time.

In a parallel reduction machine, granularity of work division is also an important factor which affects the ratio of time spent on administrative overheads to total execution time. SK-combinators are not so satisfactory from this point of view as they divide the work into uneconomically tiny steps. The use of supercombinators improves the situation by increasing the grain size of work, though at the cost of making the reduction rules more generalised as compared to the fixed ones for SK-combinators. Definitions of supercombinators look like those of user defined functions only but they have no free variables. Refinement of supercombinators in the form of Serial combinators [16] represents an attempt towards optimising grain size without losing parallelism.

After translating the source program into a suitable intermediate language, the next step in computation is the reduction of graph representing the program expression. The matter requires decisions about reduction order and argument passing mechanisms. Choice for reduction order is mainly between normal order and applicative order. The former uses arguments in unreduced form and implements a call-by-need semantics i.e. a computation is taken up only when needed. The applicative order, on the other hand, reduces the arguments before substitution thus giving a call-by-value effect. It is eager in approach because all arguments are simplified without establishing need. Normal order is safe as it never gets entangled in unneeded work and thus

always terminates unless the program itself is non-terminating. Applicative order, though unsafe, provides better utilisation of parallelism in a program. Choice for argument passing mechanism is mainly between string reduction (where a complete copy of the argument is substituted) and graph reduction (where a pointer to the argument is substituted). Normal order coupled with graph reduction implements lazy evaluation i.e. an expression is evaluated only when needed (due to normal order) and that too only once (due to graph reduction).

Research on the reduction models of computation, based on above ideas of functional computation, is continuing and is still in its developmental stage. The present work is concerned with the design of a reduction type multiprocessor computation model based on the functional approach.

1.3 STATEMENT OF THE PROBLEM

A major goal set for the model is to support, (i) pattern-matching function definitions (functions defined on structured parameters), (ii) simple and recursive local definitions within definitions, and (iii) recursion. In addition, the model should be able to utilise parallelism without excessive communication overheads.

A brief explanation of the solutions suggested in the thesis for meeting the above objectives is now given. The model takes programs in supercombinatory form (algorithms are already available for compiling programs in higher level functional languages into this form). Thus the input to the model is a set

of supercombinatory definitions and an expression for evaluation.

The problem of computational model design then, as treated in the thesis, consists of three main parts:

(1) **development of a lambda calculus basis for the concept of pattern-matching keeping in view a parallel processing of the matching work** - Here an algorithm for the matching, and a set of modified β-reduction rules for dealing with lambda abstractions which bind general patterns rather than simple variables have been developed.

(2) **compilation of supercombinator definitions** - The main work here is compilation of definition bodies into a language of Structured Director String Terms (SDST) which is an extension of DST [14]. A concept of structured directors (pattern director and list director) has been introduced for dealing with structured arguments appearing due to pattern-matching. These directors advise structure breaking only when essential and to the extent it is necessary. The modified β-reduction rules form a semantic base for these directors. For local definitions, a concept of context list whose elements are connected to the main body through pointers, has been suggested.

(3) **development of a reduction mechanism in a multiprocessor environment for the compiled program** - The work here starts with the framing of reduction rules for SDS terms representing a definition body. The program expression is organised into a graph of tasks (each may be handled by an

individual processor) where a task is an indivisible piece of work comprising a function and its arguments. Design of task structure takes care to keep the communication between processors low even at the cost of elaborate fields in task structure.

Although compiling supercombinators into director strings may look like going from large execution steps to small fine grained ones and thus losing the advantage of lesser communication overheads, but the model still has coarse grain reductions. A redex, here, is a supercombinator applied to all its arguments and it is considered as a monolithic piece of work (a task) handled by one processor only. The compilation to structured director strings is being used only to simplify and mechanise the process of argument substitution.

1.4 ORGANISATION OF THE THESIS

We begin in chapter 2 by giving a brief introduction to functional programming and lambda calculus which has been included feeling that the general reader in the area of computer science is not so familiar with these topics. Chapter 3 reviews the computation models and experimental machines based on reduction concepts for functional languages. A brief discussion of the proposed model is also included in this chapter. The design of the model begins in Chapter 4 with the development of a lambda calculus basis for pattern-matching. In Chapter 5, a compilation scheme for converting pattern-matching supercombinator definitions into the

variable-free form of SDS terms is developed which includes a pattern abstraction algorithm. Chapter 6 continues with the compilation while dealing with simple and recursive local definitions in a supercombinator body. The results of the complete compilation are packed here into a structure named as D-code. Chapter 7 is then devoted to the development of a multiprocessor reduction mechanism where the compiled definitions are used as 'mechanised' rules for evaluating the program expression. The structure of a task is developed and the message passing schemes described. Handling of recursion and shared sub-expressions is also discussed in this chapter. The complete reduction mechanism is illustrated through some examples in the Appendix. Finally conclusions and scope for future work are given in Chapter 8.

CHAPTER 2

FUNCTIONAL PROGRAMMING AND LAMBDA CALCULUS

Refinement of programming languages has been a continuous process directed towards easing the job of a programmer. Backus, however, in his Turing award lecture [2], says that the development of various imperative languages after FORTRAN has not contributed much in this direction, and stresses that a radical change in programming methodology is necessary to achieve a breakthrough. A program in an imperative language is a cunningly designed maze, through a single thread of control, and a reader is forced to go round this criss-cross, doing a mental execution all the time, to find its meaning. In other words, programs have an operational reading. Although structured programming, Backus says [2], is an effort to bring some order to this chaotic world, but it accomplishes little in attacking the fundamental problems created by the word-at-a-time considerations in imperative style of programming.

Functional programs, on the other hand, have, like mathematical analysis, a denotational reading i.e. programs offer a static meaning without the need to go through the process of a mental execution. Functional languages have more expressive power [17,18], and their use can increase programmer productivity by allowing him to concentrate more on algorithmic thinking instead

of worrying about low level details such as keeping track of variables through the not-so-orderly sequencing of imperative languages which is made further complicated by the presence of side effect causing constructs such as assignment.

The first functional language was LISP invented by McCarthy as a formalism for reasoning about recursion equations as a model of computation [19]. Some other functional languages which have been developed are FP [2,20], SASL [21], HOPE [22], KRC [23], Ponder [24], Lazy ML [25], Miranda [26], Orwell [27], Haskell [28].

2.1 FUNCTIONAL PROGRAMMING [11,29-33]

The type of 'statements' contained in a functional language are definitions and expressions. An expression states the intention of program while definitions act as rewrite rules for evaluating the expression. Expressions are built through a single concept of binary application representing an operator applied to an operand (it gives the languages another name - applicative languages). Application is expressed by juxtaposition e.g. the expression (SQ 3) denotes the application of a function SQ (square) to the argument 3. It has left-association property so that an expression (f g x) means ((f g) x) i.e. f applied to g and the result applied to x, and not (f (g x)).

Definitions are used to define functions in terms of other simpler or primitive functions. Programming allows building up of a hierarchy of functions where more complicated ones are built on the earlier defined ones. A function is a kind of program which

accepts inputs (in the form of arguments) and produces output (the value of the function call).

The concept of a function in functional languages is same as in mathematics although a little different notation is used. In mathematics, functions are written by enclosing the variables within brackets, e.g.,

```
f(x)     = (x * x) + 3
max(x,y) = if x > y then x else y                    ... (2.1)
```

In functional notation, f(x) is interpreted as f applied to x, and therefore the brackets enclosing the variable are dropped. However, an interpretation problem arises with functions of more than one variable such as 'max' or the operator '+' on the right-hand side of f(x). **Currying**, a method introduced by Schönfinkel [34] and extensively used by Curry [35], is used to resolve it. The method represents all multi-argument functions as sequences of unary ones. For example, the '+' operator takes two arguments, and following the mathematical notation it would be written as (+ (x,y)). Its type is expressed as [N x N --> N] where N is the set of natural numbers. In Curryed form, it will be expressed as ((+ x) y) with type representation as [N --> [N --> N]]. Now '+' is understood as a unary operator which when applied to x, yields another unary function (+ x) which adds x to an argument presented to it. Based on the above, the function definitions in Eq. 2.1 are written as (all function names are typed in italics for easy recognition)

```
f x       = + (* x x) 3
max x y = IF (> x y) x y                                        (2.2)
```

In the definition of *max*, the if-then-else construct has been replaced by a Curryed IF operator which is a primitive function whose semantics is given by

```
IF True x y  = x
IF False x y = y
```

A function for finding the maximum of three numbers can be built on the definition of function *max* as

```
max_of_three x y z = max (max x y) z
```

An expression (*max_of_three* 2 7 5) is evaluated using definitions of *max* and *max_of_three* through following rewrites:

```
max_of_three 2 7 5 --> max (max 2 7) 5
                   --> max (IF (> 2 7) 2 7) 5
                   --> max (IF False 2 7) 5
                   --> max 7 5
                   --> IF (> 7 5) 7 5
                   --> IF True 7 5   --> 7
```

A powerful feature of functional style is to allow recursion in definitions. A very common example is the factorial function

```
factorial n = IF (= 0 n) 1 (* n factorial (- n 1))
```

In a recursive definition, a function invokes itself on the right-hand side.

Data structures are introduced in functional languages through constructor functions for data types. For example, CONS is a list constructor, sometimes written as infix operator ':' also. Similarly, BRANCH is constructor for tree type structures. Data structures can appear as parameters to a function e.g., a function *length* returning the length of a list is defined as

length NIL = 0

length (x:xs) = + 1 (*length* xs)

where NIL stands for an empty list. The above definition illustrates another feature, called pattern matching, where several equations (clauses) are written each defining the function with a particular case of the data structure involved. Pattern matching is discussed in more detail in chapter 4.

Functional languages draw no distinction between functions or data. Functions can be passed around as data objects serving as arguments for other functions. Such functions which take other functions as parameters are called **higher order functions**. Turner [21] has described it saying that functional languages treat all objects equally and that there are no "first or second class citizens". An example of a higher order function is a function *map* defined as

map f NIL = NIL

map f (x:xs) = (f x) : (*map* f xs)

The function *map* takes a unary function f and a list as arguments and produces another list whose elements are the results of

applying f to the elements of original list.

Functional style of programming is further illustrated in the following program (in Miranda style) that returns a list of square upto a given integer, and uses the function *map*:

```
----------------------------------------------------------------
square_list n    = map SQ (count 1 n)
map f NIL        = NIL
map f (x:xs)     = (f x) : (map f xs)
count i k        = IF (> k i) NIL (i : count (+ 1 i) k)
----------------------------------------------------------------
square_list 5
```

The function SQ, taken as a primitive operator, has not been defined further. The program returns a list [1, 4, 9, 16, 25]. It may be seen that there is no execution ordering implied by the program nor the programmer has to worry about it. An implementation, however, applies a dynamic ordering which is decided by the reduction strategy. The matter is discussed in next section. The variables, used in a definition, have a scope limited to the right-hand side and they get bound to arguments during the reduction of an application.

All functional languages derive their semantic base in the lambda calculus developed by Alonzo Church [12]. Ability to translate a high level program into lambda notation has given rise to the possibility of their efficient machine implementations. Besides that, lambda calculus provides an appropriate framework for mathematical reasoning about functional programs.

2.2 LAMBDA CALCULUS [12,31,33,36-38]

Lambda calculus is a product of the work on computability theory regarding the ability to define precisely the notion of computable functions. Although it has been used as a semantic base for functional languages, but it is interesting to note that certain concepts of ALGOL 60 (and similar languages) can be viewed as syntactic variations of lambda calculus [39].

2.2.1 Lambda Syntax

In mathematical notation, we are used to expressing functions as f(x) = exp where x is a variable over which the function is defined, and exp is the body of the function. A general view is that a function takes some values as arguments and returns another value as result. Lambda calculus discards this viewpoint and allows functions to take functions and return functions. For example, lambda calculus would allow a function *apply* to be defined as (in mathematical notation)

apply (f,y) = f(y)

One of the arguments to the function *apply* is a function. Concept of higher order functions in functional programming is similar to this idea.

Another big difference in expressing functions in lambda calculus is the flexibility in naming them. Lambda calculus advocates that in order to create and manipulate functions as objects, a notation for unnamed functions is required. This is achieved by a process

known as **abstraction**. Through abstraction, an expression can be made to behave like a function. As an example, consider a function f(x) = x * x + 2. The right-hand side of this definition is converted into a new form as

$$\lambda x.x * x + 2 \qquad ...(2.3)$$

The above form is called a **lambda-abstraction** where 'λ' is an abstraction operator, the prefix 'λx.' abstracts the function f (with respect to variable x) from the expression on right-hand side, and the expression following the dot is known as the **body** of the abstraction.

The unnamed function in Eq. 2.3 can be applied to values just like the function f. In mathematical notation, f(3) represents the value of f at x = 3. In lambda notation, it is interpreted as an application and is written as (f 3) because juxtaposition denotes application. Using the unnamed function, the application (f 3) is written as

$$(\lambda x.x * x + 2)\ 3$$

Functions of several variables are expressed through multiple abstractions as

$$\lambda x_1.\lambda x_2. \ldots \lambda x_n.E$$

or as [36]

$$\lambda \tilde{x}.E \quad \text{where } \tilde{x} = x_1, x_2, \ldots, x_n$$

It may be noted that abstraction associates to the right. The

function *max* in Eq. 2.2 would be expressed as a lambda abstract by

$\lambda x.\lambda y.$IF (> x y) x y

Applicative expressions in lambda notation are written as Curried applications (as discussed in section 2.1). Thus, strictly according to lambda syntax, the abstraction in Eq. 2.3 should be written as

$\lambda x.$+ (* x x) 2 ... (2.4)

The complete syntax for lambda expressions is summarised in the following definition [30]:

E ::= constant |	built-in constant
variable |	variable names
E_1 E_2 |	application
(E) |	bracketing
λvariable.E	abstraction ...(2.5)

2.2.2 Lambda Semantics

Lambda expressions having syntax governed by Eq. 2.5 are well formed expressions in lambda calculus. The calculus is completed by a set of lambda conversion rules which convert one expression to another having an equivalent meaning but, may be, of a simpler form. To introduce the rules, we need to know the terms bound and free occurrences of a variable.

An occurrence of a variable in a lambda expression is **bound** if there is an enclosing lambda which binds it and is **free** otherwise.

For example, in the λ-expression (λx.+ x y), x occurs bound and y is free, but considering the body only, both x and y occur free.

Returning to conversion rules, the rule of maximum importance is β-rule which deals with the application of a lambda abstraction to an argument. The rule, in words, says,

> "The result of applying a λ-abstraction to an argument is an **instance** of the body with argument substituted for all free occurrences of the bound variable."

Thus the result of applying the abstract in Eq. 2.4 to an argument '3' is obtained as follows:

```
(λx.+ (* x x) 2) 3 --> + (* 3 3) 2        (by β-rule)
                   --> + 9 2  --> 11      (by operator rules)
```

One application of β-rule removes one layer of abstraction and performs the job of substituting arguments into the body. This process has resemblance to the association of actual parameters with formal parameters in a procedural language. The action of β-rule is also expressed through a notation [Q/x]E meaning, 'the expression E with Q substituted for all free occurrences of x in it'. Using this notation, all conversion rules are stated as

```
(α-rule) :    λx.E      <--> (λy.[y/x]E),  y is not free in E
(β-rule) :    (λx.E) Q <--> [Q/x]E
(η-rule) :    (λx.E x) <--> E,   x is not free in E and E is a
                                 function.
```

α-rule is mainly a variable-name change rule. It says that an expression remains unchanged if a variable in it is changed

consistently. It is like in algebraic equations where a change in a variable name everywhere makes no difference. The rule is basically used to avoid name clashes [11]. η-rule is a kind of optimisation because it removes redundant abstractions. For example, the expression (λx.+ 1 x) is η-convertible to (+ 1) because both ultimately add 1 to an argument. The rules when used from left to right, are called reductions (instead of conversion) and are written with a '-->' symbol instead of '<-->'.

2.2.3 Reduction Order

An important issue, while simplifying an expression through reduction rules, is that of the reduction order. This is important because an expression may contain several redexes (reducible sub-expressions) and reduction can proceed via several alternate routes. When the expression contains no redex, it is said to be in **normal form**.

All expressions do not have a normal form, and not all reduction sequences necessarily reach the normal form, if one exists. For example, the expression (λx.x x) λx.x x has no normal form (because a β-reduction results back in the same expression), and the expression

(λx.3) (λx.x x) λx.x x

has a normal form '3' which can be reached if the left-most redex is evaluated first. But if we start reducing the sub-expression ((λx.x x) λx.x x) first, the evaluation fails to terminate. It means that the order of reduction affects the outcome of an

evaluation. However, two theorems, called Church-Rosser Theorems (CRT) I and II [40], are highly reassuring in this state of affairs.

__Theorem__ __I__ : If E --> E_1 and E --> E_2 then there exists an expression F such that E_1 --> F and E_2 --> F.

An important corollary of this theorem is that no expression can be converted to two distinct normal forms (two normal forms are distinct if they are not α-convertible). Thus all reduction sequences will reach the same result provided they terminate. The theorem assures that we could use any reduction sequence without any fear of reaching a wrong result, but a particular sequence may not terminate.

__Theorem__ __II__ : If E_1 --> E_2 and E_2 is in normal form, then there exists a normal order reduction sequence from E_1 to E_2.

The theorem says that the normal order reduction always reaches the result if it exists. Put together, the theorems say that there is at most one possible result and normal order must reach it. The proofs of the theorems can be seen in [37,40].

A normal order reduction sequence selects, at every stage, the left-most (outermost) redex for reduction. Another reduction sequence is called an **applicative order** where the left-most redex free of internal redexes (innermost) is chosen. The two orders have their own merits and demerits. While normal order is safe (it guarantees to terminate in normal form if one exists), it generally takes more number of reductions than the applicative

order because it substitutes the arguments in an unreduced form into a body and hence reduces them as many times as the bound variable occurs in the body. Its advantage is that it does not reduce the argument if it is not needed. The same thing makes it safe too. The applicative order, on the other hand, reduces the argument to normal form before substitution and thus avoids repeated reductions. It does so without checking the need and hence is unsafe because an unneeded argument could be a non-terminating one. Applicative order can allow more parallelism by selecting all innermost redexes simultaneously.

The reduction strategies are related to parameter passing mechanisms of imperative languages. The applicative order is like call-by-value while normal order implements call-by-need. A very appropriate term for normal order is that it is **lazy** (taking up a job only when needed). The applicative order is similarly called **eager**. Wadsworth [41] improved upon normal order by suggesting to pass pointers to arguments (graph reduction) instead of actual arguments and thus avoid repeated computations. This makes normal order further lazy in the sense that an expression will be evaluated only when needed and that too only once.

2.2.4 Recursion

Recursion is generally handled through Y combinator, known as the fixed point combinator. It is a lambda-abstraction of the form

$$Y = \lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x))$$

The definition can be used to show that the reduction rule for Y combinator is

Y H --> H (Y H)

Almost all implementations for functional languages use Y as a built-in operator with above reduction rule rather than using its lambda definition. Applicative order has a serious defect that it cannot come out of a (Y H) reduction because of its eagerness to reduce (Y H), the argument to H after first reduction, to normal form. In this situation, the laziness of normal order is a great help.

It may be observed that lambda calculus has a simple syntax, and functional languages tend to follow its style. Programs written in functional languages can be translated into lambda calculus and then evaluation can be done according to its conversion rules. There are some drawbacks in the process. Lambda expressions are not easily amenable to mechanised reduction, firstly due to the presence of free variables, and secondly due to β-reduction requiring to search for all occurrences of bound variable in the body (specially difficult if the body is large). Besides, α-conversions may be required to take care of name clashes.

In view of these problems, some other representations which are basically variations of lambda calculus only, have been developed and used to implement functional languages. SK-combinators [13] is one example which is a variable-free representation. It eliminates the problem of free variables. Some other

representations are director strings [14], de Bruijn numbers [42], supercombinators [15], categorical combinators [43] etc.. The next chapter reviews the computation models and reduction machines based on these variants of lambda calculus and presents a view of our model in that context.

REVIEW OF COMPUTATION MODELS

3.1 INTRODUCTION

A computation model acts as a bridge between the software and hardware levels of descriptions. The traditional von Neumann model [1] of computation essentially says that a computation is a sequence of fetch/compute/store cycles. The imperative style of programming is greatly influenced by this model. A program in these languages is a sequence of commands where the variables are visualised as locations in the memory, and the assignment statement imitates the fetch/compute/store cycles. In short, the imperative languages are "high level versions" of the von Neumann computer.

The concept of functional languages has developed more or less independently of an underlying computation model. Realising their importance as a highly expressive and elegant programming medium, the research on designing suitable models for them (or adapting von Neumann model) is active. Obviously, in this case, the language is having an upper hand in the model design rather than model dictating terms with the language.

There are three main models available for the implementation of a functional language: Control Flow, Data Flow and Reduction

[44,45]. The von Neumann computer is a sequential control flow model where control is passed from instruction to instruction either implicitly (through program counter to the next instruction in sequence) or explicitly (through GOTO). Control flow considers instructions as active agents that transform the passive data.

Data flow [46-48], in contrast, treats data as active agents moving through the passive instruction graph (hence the name data flow). The programs in data flow model are represented as directed graphs (DG), called data-flow graphs, where arcs indicate the flow of data between instructions. The arcs also serve as communication paths for the messages (tokens) generated by nodes or supplied from the external environment. The nodes are executed (fired) in a data-driven manner. Sequencing of firing is based on data dependencies only and so a greater degree of parallelism, compared to control flow, can be supported. Several data flow architectures are operational/under development [49-54].

Reduction model has no concept of a 'flow' of control either program based (Control flow) or data based (Data flow). The model makes no distinction between program and data. Both instructions and operands are expressions. Reduction model is based on mathematical reasoning where program execution goes like simplifications of mathematical expressions. Depending on how the expressions are passed as operands, one can have either string reduction (where expressions are formed of literals or values) or graph reduction (where expressions are literals or references).

Discussing the various implications of the three models, Treleaven

et al [45] have concluded : control flow lacks useful mathematical properties for reasoning about programs, and parallelism is alien to its concept; data flow permits highly parallel implementations, but its utility as a general-purpose program organisation is questionable and is more suited to specialist applications; reduction appears to be the most natural candidate for providing efficient support to functional programming. Graph reduction is inherently a parallel activity supporting simultaneous and asynchronous reductions at several sites within the graph, and hence functional languages, using pure functions with no side effects, would be best utilised by this kind of model.

As our work is about the design of a reduction based model, the discussion is now restricted to the developments in reduction models.

3.2 REDUCTION COMPUTER

A reduction machine does not run a program in the conventional sense. It rather operates on an expression, continually simplifying (reducing) it until it is in the simplest possible form which is then delivered as the value of the expression. If the expression given to the machine is a program in a high level functional language, then the final result is the value of program. However, expressions in a high level language would be quite unsuitable for a machine and hence there is a need for an intermediate language in which the high level expressions can be conveniently represented within the machine, and which has some simple rules for reduction. The design of such an intermediate

language and the translation of user programs into it is a major concern of any model design.

In its simplest form, a reduction model can be represented as shown in Fig. 3.1. Various reduction models differ in the intermediate language used. Lambda calculus [36] is one such language and it forms the basis for all others. Through its simple syntax (Eq. 2.5), it can represent all features of a high level functional language. Lambda expressions are easily expressible as tree structures in a machine.

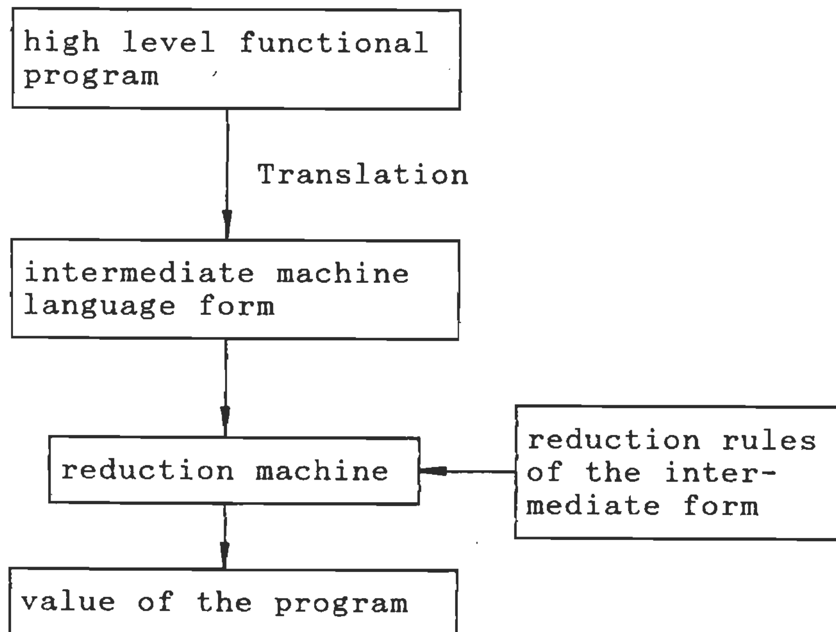
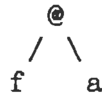


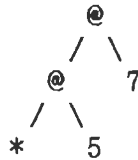
Figure 3.1 Representation of a reduction computer

3.2.1 Program Representation

A λ-expression is represented in the form of a tree which reflects its syntactical structure. The leaves of the tree are constant values (such as 0, True), built-in operators (such as +, IF, =), or variable names. The application of a function to an argument is represented as (binary tree)



where '@' sign indicates an application type node. Functions of several arguments, dealt through Currying, are shown through left-associative application resulting in a left linear binary tree such as



The tree denotes the expression (* 5 7). A function (lambda abstract) is of the form λ(formals).(body), and its application to actual parameters is reduced through β-reductions expressed as

(λ(formals).(body))actuals --> body (with actuals substituted for formals)

A lambda abstract is represented by a lambda node as shown in Fig. 3.2(a) and its application to arguments as in Fig. 3.2(b). A β-reduction constructs a new copy of the lambda body (an instance) where free occurrences of a bound variable are replaced by an argument. The process is called **instantiation.**

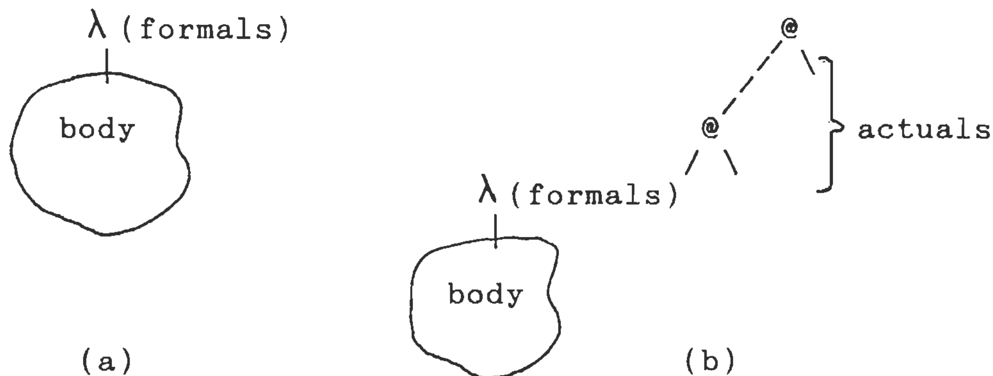


Figure 3.2 Representation of a lambda abstract - (a) lambda node, (b) lambda abstract applied to actuals

3.2.2 Reduction Options

The lambda body in Fig. 3.2(b) may have several occurrences of a formal parameter so that while reducing, there are two options :

(1) the argument may be copied for substitution into each occurrence (string reduction), or

(2) a pointer to the argument may be substituted into each occurrence (graph reduction).

String reduction suffers from heavy copying (if the argument is a large expression) and wasteful work (if argument contains redexes which get duplicated in copying). The performance of the method can be improved if used with applicative order where the argument is always in normal form before a β-reduction is taken up. GMD reduction machine [55] is a lambda calculus based string reduction machine. The aim of this project was to demonstrate reduction machines as an alternative to conventional architectures. It

concluded that string reduction is a useful technique but may be inefficient if adhered rigorously. Mago's [56,57] cellular tree machine, implementing Backus FP [2] class of languages, is based on distributed string reduction and relies on massive parallelism to overcome its inefficiency. CTDNet [58,59] is an applicative order lambda reducer which uses a modified form of string reduction (permitting limited use of pointers) to improve performance. Berkling has done extensive work on string reduction systems based on applicative λ-expressions [60].

The second option, of graph reduction, was suggested by Wadsworth [41]. The method eliminates the inefficiency due to repeated computations and also the need for copying big expressions by converting multiple references for a common sub-expression into multiple arcs (pointers) directed to the root of the common sub-graph. Using normal order with graph reduction, Wadsworth combined the benefits of both normal order and applicative order yielding a reduction system called normal order graph reduction or lazy evaluation [61]. The method is safe (having the Church-Rosser property [40] of normal order) and, like applicative order, reduces expressions once (rather at most once) only.

Graph reduction serves as a general reduction model, and is used with many different types of intermediate forms of machine languages although the above discussion was in the context of lambda calculus language.

3.2.3 Intermediate Forms As Machine Languages

Lambda calculus has some drawbacks when directly used as a machine language. β-reduction, a fundamental operation here, is inefficient because the instantiation process has to visit every leaf of the lambda body to check for a free occurrence of the bound variable. There is a danger of inefficiency while substituting into very large bodies. Further, new instances of sub-expressions containing no free occurrences of the bound variable get constructed unnecessarily.

A solution to the above problem is to compile each lambda body into a fixed code which, when executed, builds an instance. Unfortunately, the presence of free variables in lambda expressions makes a complete mess of it because the compiled code is going to be different for different values of a free variable. SECD machine [29,62] permits parameterising of code sequences on the values of free variables. It is an abstract architecture that implements applicative order reduction of λ-expressions. The machine maintains four data structures called the Stack, the Environment, the Control and the Dump. In the machine, lambda abstractions are compiled into a package, called closure, consisting of the abstract and an environment for its free variables.

Another approach, different from the above 'environment' model for dealing with variables in applicative languages, has been developed by Turner [13] which is based on a result in logic that variables, as used in logic and mathematics, are not strictly

necessary [34]. Certain intermediate languages, for program evaluation, have been developed around these ideas which are being discussed now.

3.2.3.1 Combinators

Schönfinkel [34] and Curry and Feys [35] have developed a theory of combinators which is a recast of the entire lambda calculus. Combinators are equivalent to closed λ-expressions i.e. the ones which contain no free variables. Turner suggested the use of this combinator theory in developing a variable-free representation of expressions. His algorithm is a process of abstraction which removes all occurrences of a variable from an expression yielding variables-free abstracts. For a function f, defined by f(x) = exp, an object equivalent to f is obtained by abstracting out the variable x from **exp**, which is written as [x]**exp**. Application of this functional object to a value v is written as ([x]**exp**) v, and the connection between application and abstraction is defined by

([x]**exp**) x = **exp**

which is same as (λx.body) x = body, in effect. Turner's algorithm is based on three simple abstraction rules :

[x] x	= I	(I for identity function)
[x] y	= K y, $x \neq y$	(K for constant)
[x] e_1 e_2	= S ([x]e_1) ([x]e_2)	(S for steering)

where S, K and I are primitive functions called combinators and their application is defined by following reduction rules :

```
I x     --> x
K y x   --> y
S f g x --> f x (g x)                                    ... (3.1)
```

Through repeated abstractions, all variables may be eliminated from an expression and thus the substitution operation of β-reduction becomes meaningless. The problem of name-clash in λ-expressions is also eliminated. A problem with SK combinators is that they yield a code quadratically expanding with successive abstractions. Turner optimised [63] the algorithm to convert the quadratic expansion into a linear build up by introducing some additional combinators. Two of them have the following reduction rules:

```
B f g x --> f (g x)
C f g x --> f x g                                        ... (3.2)
```

The B and C combinators are optimised versions of S. B steers its third argument to the second while C to the first. Various experimental computing engines based on combinator reduction have been developed and studied [64-67]. Hybrid graph reducer based on a unified model of combinator reduction and λ-style reduction has also been reported [68,69].

3.2.3.2 Director string terms

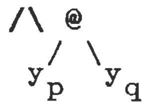
Reduction rules for combinators show that they behave like argument directing operators. They replace the substitution process of β-reduction by combinator reduction. Kennaway and

Sleep [70] suggested the use of directors as annotations at the nodes of a tree (representing an expression) to achieve the same effect. As an example, let (p q) be an expression where p and q are some further expressions. Let both contain references to a variable y. Abstracting y from (p q) gives

$$[y](p\ q) = S\ {}^{y}p\ {}^{y}q$$

where ${}^{y}p$ denotes the result of [y]p and ${}^{y}q$ that of [y]q. The tree structures for (p q) and (S ${}^{y}p$ ${}^{y}q$) are

Here the intention of S combinator can also be expressed using a director at the node in the expression (${}^{y}p$ ${}^{y}q$) as



The director '/\', annotating the application, indicates that the node is a function expecting one argument which on arrival, should be sent into both the branches (/\ indicating both ways). Abstracting out more than one variable leads to a string of directors at the nodes, called director strings. Comparing the syntax trees for (S ${}^{y}p$ ${}^{y}q$) and (${}^{y}p$ ${}^{y}q$), it can be seen that the former requires an extra node. Thus the use of director strings leads to a more compact variable-free code.

Kennaway and Sleep have given an algorithm [71] for converting λ-expressions into what they call Director String Terms (DST). In DST notation, a node is represented as a 3-tuple term written as

($d_2D_2@_2$, t_1, t_2) where d_2D_2 is a binary director string (indicated by subscript 2) with d_2 as the leading director; $@_2$ stands for a binary application node; t_1, t_2 are the DS terms for left and right sub-trees at this node. Similarly, the leaves are expressed as pair terms ($d_1D_1@_1$, t_1). The subscript 1 now represents a unary case. A leaf, having variable, is converted into a hole representing an empty space to be filled by an argument which reaches it following the directors. The Director String Calculus (DSC) [71], a calculus of DST, captures the essence of the combinators introduced by Turner. DSC reduction has been shown to preserve beta-equivalence under backward translation to lambda terms. The algorithm for conversion to DST has been further optimised from code size point of view [72,73].

Director strings have been used in one of the versions of SKI machine [74]. CTDNet [59] is a lambda reducer which, using director string annotated process graphs, improves the efficiency of ß-reduction. Another use of director strings has been reported in COBWEB-2 [75].

3.2.3.3 Supercombinators

The idea of SK combinators (or of director strings) is quite appealing from implementation point of view because the machine has to handle a fixed set of combinators which can be implemented directly on hardware eliminating the need for an extra level of interpretation. However, an individual reduction does very small amount of work. Each reduction simply produces the effect of pushing an argument down by one step in the syntax tree of a

function body. Their fine grain size is 'uneconomical' due to the overheads (such as finding next combinator, accessing the argument etc.) dominating the execution time. It can be argued that each reduction should do sufficient work to justify the time spent on preparation. Hughes [15] proposed the idea of 'customised' combinators derived from user programs. His combinators, named as supercombinators, are bigger in grain size and hence avoid the dominance of overheads in the execution time.

A supercombinator \$S ('\$' sign indicating a supercombinator identifier) of arity n is equivalent to a λ-expression of the form [11]

$$\lambda x_1.\lambda x_2. \ldots \lambda x_n.E, \quad n \geq 0 \qquad \ldots (3.3)$$

where E is not a λ-abstraction such that, (i) \$S has no free variables, and (ii) any λ-abstraction inside E is a supercombinator. Thus a supercombinator is a function of n variables containing no reference to any other variable. Its definition may be written as

$$\$S\ x_1\ x_2 \ldots x_n = E$$

The definition acts as its reduction rule. A reduction consists of applying the supercombinator to all n arguments, and the result is an instance of the supercombinator body E with appropriate arguments substituted for all occurrences of the formal parameters. Thus the reduction is equivalent to performing several β-reductions simultaneously.

Johnsson has developed a transformation algorithm [76], called 'lambda-lifting', which converts a λ-expression into super-combinatory form. The algorithm starts with the innermost lambda abstraction, making its free variables into extra parameters of a supercombinator. The process is continued till no lambda is left in the expression. To make the supercombinators fully lazy (so that no repeated evaluations may occur), Hughes generalised the idea of free variables to free expressions i.e., ones not containing any occurrence of the bound variable. According to him, the value of a free expression will be same between all instances, and hence should be shared to preserve laziness. His transformation [77] makes each **maximal free expression** (instead of a free variable which is actually a minimal free expression) into an extra parameter of a supercombinator.

Using the above transformations, a program in a higher level language can be converted into a supercombinatory program of the form

```
-----------------------------------
   Supercombinator definition
   ...
   ...
-----------------------------------
   Expression to be evaluated                          ... (3.4)
```

The stages involved in the conversion are shown in Fig. 3.3.

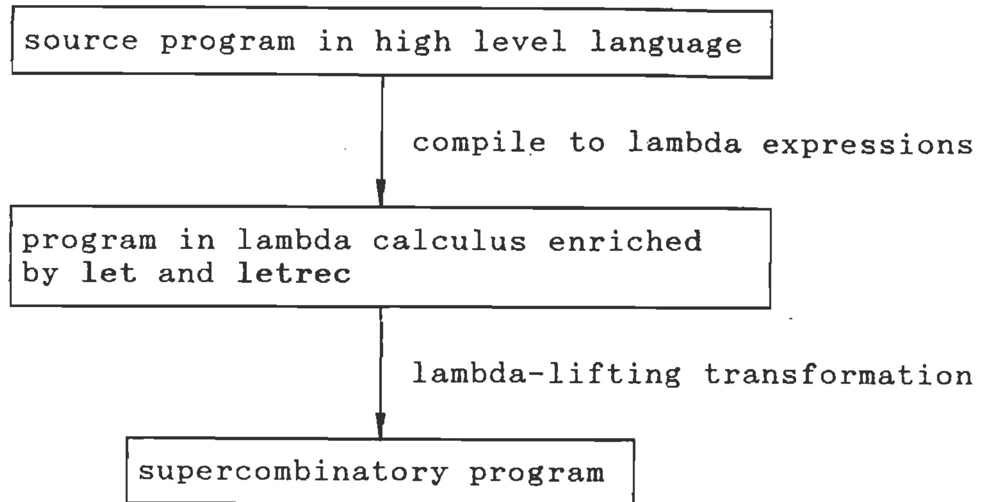


Figure 3.3 Compilation to supercombinatory form

Many systems consider supercombinator definitions as a set of rewrite rules [78]. A reduction then consists of rewriting an expression which matches the left-hand side of a rule with an instance of the corresponding right-hand side, thus constituting a term rewrite system [79-82].

There have been some extensions to the idea of supercombinators such as Serial combinators [16,83] which have an optimised level of grain size without sacrificing parallelism; or refined supercombinators [84] which are based on the detection of sharing of partial function applications, and they avoid unnecessary overheads in cases where no sharing occurs.

Categorical combinators [43,85] is another line of action which aims, like supercombinators, at performing the equivalent of several β-reductions simultaneously. The combinators utilise the

concept of de Bruijn numbers [42]. The related research and some abstract machines based on them are discussed in [85-87].

3.3 SUPERCOMBINATOR REDUCTION MACHINES

Supercombinator reduction has generated maximum interest as an efficient method for implementing functional languages e.g., G-machine [88-90], GRIP [91,92], ALICE [93,94], Flagship [95-99], TIM [100]. As supercombinators are no primitive operators having fixed reduction rules, a pre-processing of their definitions in a program is essential to facilitate reduction. Thus all machines which use supercombinators, have an additional step of compiling the supercombinator program (Fig. 3.3) further.

G-machine, developed at Chalmers Institute of Technology, Sweden, is an extremely fast implementation of supercombinator graph reduction. In this machine, the pre-processing consists of compiling each supercombinator definition into a sequential code of stack manipulative instructions (the G-code). During expression evaluation, whenever an application of a super-combinator is encountered, the machine executes the G-code compiled from its definition. The execution builds an instance using the arguments from stack. This reduction in G-machine has been termed as programmed graph reduction [89].

The original G-machine was a uniprocessor model but research effort in the direction of realising a parallel G-machine is continuing. One attempt has been to allow multiple execution threads of control, allocating a stack for each thread [101-103].

These designs lead to multiple and arbitrarily deep stacks giving a 'cactus stack' structure which is difficult to implement efficiently. Another direction is represented in the development of GRIP architecture [91,92] - a project supported by UK Alvey Programme in collaboration with ICL and High Level Hardware Limited. The machine implements supercombinator graph reduction in parallel. Initially, it planned to represent supercombinators as tree-structured 'templates' which are instantiated when applied to arguments. The machine is based on a high-bandwidth bus architecture. The bus provides access to a large, distributed shared memory, and by using Intelligent Memory Units (IMU) and packet-switching protocols, it can support a large number of processors working in parallel. Several other implementations of parallel functional programming such as parallel combinator reduction, parallel G-machine etc., are being tried [104,105] using this architecture.

ALICE [94] and Flagship [95] (later version of ALICE) compile a supercombinator definition into an imperative code block which is kept in memory, with an identifier, as code packet. The machine uses some other types of packets also which represent an application of a supercombinator to its arguments [97]. The application is reduced by executing the code packet of that particular supercombinator. All packets reside in different memory locations and constitute a graph through their links. The architectures allow parallel threads of control, and a packet communication system maintains track of the graph transformations being done in parallel. Simulation studies on Flagship measure

program parallelism in terms of the length of critical path - the longest sequential thread of computation [99].

TIM (three instruction machine) [100] draws ideas from G-machine and SECD machine [62], and is claimed to be faster than G-machine. It compiles a supercombinator definition into a sequential code made using instructions from a small set of three instructions only. The execution follows the style of SECD machine.

3.4 THE PROPOSED MODEL

The aim of the research work taken up was to design a supercombinator based computation model for a multiprocessor environment. The pre-processing of supercombinator definitions in the models, discussed above, consists of compiling them into linear sequences of instructions which when executed, build an instance. The other approach to supercombinator reduction is template-instantiation [11,92]. Here a supercombinator body is kept as a tree and instantiation is done through tree-walking as in β-reduction.

In the proposed model, the template-instantiation is being used, and the language of director string terms (DST) [71] is adapted to improve the efficiency of tree-walking mechanism. Director string annotations at the nodes in a definition body tree act as 'signposts' so that the instantiator need not 'try' every leaf or sub-tree to find an occurrence of the formal parameter. Lot of unnecessary tree-walking can be avoided by this. Hence the pre-processing of supercombinator definitions in this model consists

of compiling each definition body into director string terms through a process of abstracting out the formal parameters. To take care of pattern-matching definitions, the language of DST has been enriched by introducing a concept of **structured directors** - the directors which can extract and direct components of a structured argument into different sub-trees. Terms of the enriched language have been called Structured Director String Terms (SDST).

It might appear that compiling supercombinators to director strings amounts to going back from the large execution step (and hence lesser overheads) of supercombinators to small ones of the director string model. It is not so. The model implements supercombinator reduction using SDST, not as an intermediate language for program representation, but as a tool to mechanise the process of instantiation. A redex still consists of a supercombinator applied to its arguments, and is handled by a single processor using SDS term of the function body like a template having 'prompts' for argument substitution. In other words, the instantiation takes place in one large step of reducing a supercombinator (where the processor walks over the template) rather than through several small steps of reduction in director string spirit.

CHAPTER 4

IMPLEMENTATION OF PATTERN-MATCHING : A LAMBDA CALCULUS BASIS

Higher level functional languages make extensive use of pattern-matching to increase their expressive power and elegance. A pattern-matching definition of a function has several clauses and evaluation of an expression having reference to such a function requires the ability to select the applicable clause according to the structure-type of actual parameters. The selected clause is then used, alongwith parameters, in reduction. This chapter develops a lambda calculus view of the whole issue which acts as a basis for the compilation of pattern-matching definitions discussed in next chapter. The discussion, here, is divided into two parts: one for the clause selection phase where a lambda calculus interpretation suggesting concurrent action on matching has been evolved, and the second for the reduction phase where a modified set of ß-reduction rules has been developed. These rules deal with the changed circumstances of ß-reduction when λ-abstractions binding structured variables (called patterns) are involved.

The chapter begins with a brief introduction to pattern-matching and its implementation. Section 2, starting with the enriched lambda calculus view of pattern-matching, gives a different interpretation suggesting matching in parallel. The section then

bifurcates into two sub-sections: one giving a matching algorithm to be used for clause selection, and the other giving reduction semantics for pattern-binding lambda abstractions in the form of modified ß-rewrite rules. Finally, section 3 gives conclusions of the work reported in this chapter and its link with the next.

4.1 PATTERN-MATCHING

It is a notational device for defining a function which requires a case analysis to be performed on its parameters. A pattern-matching definition contains several alternative equations distinguished by the use of different types of formal parameters called patterns. An invocation of such a function results in the use of that particular equation whose formal parameters match the arguments.

Patterns used in a pattern-matching definition are conceptually different from the variables used in an ordinary function definition. A pattern stands for a subset of the values belonging to a given data type whereas a variable stands for the complete set. A pattern is more specific and from that point of view, variable is a degenerate case of pattern. As an example, if N is the set of natural numbers then n is a variable representing any arbitrary value from N, and specific values such as 1, 5, 9 etc. are patterns defined over N. Structured data types lead to more complex patterns. Pattern formation can use data constructor functions also resulting in hierarchical patterns. A pattern is formally defined as [11]

Definition 4.1 : pattern

```
pattern              ::= Constant | variable | constructor_pattern
constructor_pattern ::= Constructor pattern_list
Constructor          ::= sum-constructor | product-constructor
pattern_list         ::= NIL | (pattern, pattern_list)
```

A pattern-list in a constructor-pattern has as many patterns as the arity of its constructor. If the constituents of a pattern are not further constructor-patterns then it is called a simple pattern.

A general pattern-matching definition for a function f, in a high level functional language, is given as

$$f\ p_{11}\ p_{12} \cdots p_{1m} = E_1$$
$$f\ p_{21}\ p_{22} \cdots p_{2m} = E_2$$
$$\ldots$$
$$f\ p_{n1}\ p_{n2} \cdots p_{nm} = E_n \qquad \ldots(4.1)$$

where p_{ij} ($i = 1$ to n, $j = 1$ to m) are patterns and E_i's are expressions in the high level language. The definition has n clauses, each defining the function in terms of m parameters. Here an expression E_i is known as the body of the ith clause and the patterns on left-hand side are its formal parameters. Following is an example of a definition in pattern-matching style for a function Sumlist, returning the sum of a list of integers:

```
Sumlist [ ]      = 0
Sumlist (x:xs) = + x (Sumlist xs)                                ...(4.2)
```

where ':' represents the list constructor CONS, x stands for the head of the list and xs for the tail. The definition has two clauses: the first to be used for an empty list and the second for a non-empty list.

Implementation of pattern-matching involves two steps:

(1) selecting the applicable clause by trying to match the arguments with formal parameters of each clause.

(2) evaluating the body of the selected clause in an environment where the variables referred by the body are bound to appropriate arguments or their constituents, as applicable.

For example, if the Sumlist function (Eq. 4.2) is applied to a list [5:7:9] then the pattern-matching process will select second clause, and the bindings produced for the evaluation would be, x --> 5; xs --> [7:9].

4.2 LAMBDA CALCULUS INTERPRETATION

Translation of pattern-matching into lambda calculus via an intermediate level of enriched lambda calculus [11,109] is a way of implementing it. The constructs in enriched lambda calculus provide an operational semantics whose approach to pattern-matching work is sequential. In order to take advantage of a multiprocessor environment, an interpretation providing matching in parallel has been developed. The analysis begins with an explanation of the enriched lambda calculus approach which is then modified to yield a parallel matching strategy.

In a high level functional language, a function definition is given as $g\ v_1\ v_2\ \ldots\ v_m = e$, where v_i's are variables and e is an expression in the concerned language. In lambda calculus, the same definition is seen as an expression with m abstractions and is written as [11,36]

$$g = \lambda v_1.\lambda v_2.\ \ldots\lambda v_m.E \equiv \lambda\tilde{v}.E$$

$$\text{where } \tilde{v} \equiv v_1,\ \ldots,\ v_m \qquad \ldots(4.3)$$

Here E is the lambda calculus version of e. The symbol '≡' indicates syntactic equality. Extending the interpretation to a function defined on patterns as $g\ p_1\ p_2\ \ldots\ p_m = e$, we get

$$g = \lambda\tilde{p}.E \quad \text{where } \tilde{p} \equiv p_1,\ p_2,\ \ldots,\ p_m \qquad \ldots(4.4)$$

An expression of the kind λ p.E is called a pattern-matching lambda abstraction [11] which is different from a simple lambda abstraction. Application of an ordinary lambda abstraction to an argument is reduced by ß-reduction rule. In other words, ß-reduction completely specifies the semantics of such an application. However, when a pattern-matching lambda abstraction is involved then a concept of pattern-match success/failure has also to be taken into account besides the ß-reduction process. Using this idea, the general pattern-matching definition (Eq. 4.1) is translated into enriched lambda calculus [11] as

$$f = \lambda\tilde{a}.(\ (\lambda\tilde{p}_1.E_1)a_1\ \ldots\ a_m \,\|\, \ldots \,\|\, (\lambda\tilde{p}_n.E_n)a_1\ \ldots\ a_m \qquad \ldots(4.5)$$

where $\tilde{p}_i \equiv p_{i1},\ \ldots,\ p_{im}$, $i = 1$ to n; $\tilde{a} \equiv a_1,\ \ldots,\ a_m$. Here $\tilde{a}$ is a set of new variable names and none of the a_j occurs free in any E_i. The symbol ‖ (fatbar) is an infix operator meaning

$$a \,\|\, b = a, \text{ if } a \neq \text{FAIL}$$

$$\text{FAIL} \,\|\, b = b$$

Operationally, ‖ evaluates its left argument; if the evaluation yields something other than FAIL , then ‖ returns that value (first rule); if it evaluates to FAIL, ‖ returns its right argument (second rule). The word evaluation here encloses both pattern-matching and reduction. We refer to this as a **combined** view of the semantics of a pattern-matching lambda abstraction. It can be further seen that the operational semantics suggests sequential trials of evaluation (matching + reduction) in a 'if...then...else if...' manner.

It is felt that the above sequential implementation is not suitable for a multiprocessor environment. We are proposing an alternative interpretation which draws ideas from an extended lambda calculus having list handling capabilities [106-108]. This lambda calculus dialect has, besides the usual α and β rules, two extra reduction rules called γ rules. The γ rules, given below, are meant for list manipulations.

$(\gamma 1)$ $[f_1, \ldots, f_n]\ E \;\text{-->}\; [f_1\ E, \ldots, f_n\ E]$

$(\gamma 2)$ $\lambda x.[E_1, \ldots, E_n] \;\text{-->}\; [\lambda x.E_1, \ldots, \lambda x.E_n]$

The theme of our parallel implementation proposal, using this list manipulative lambda calculus, is expressed in the following recast of Eq. 4.5:

$$f = \lambda \tilde{a}.[\lambda \tilde{p}_1.E_1, \ldots, \lambda \tilde{p}_n.E_n]\ a_1 \ldots a_m \qquad \ldots(4.6)$$

Using γ1 rule for the body of lambda abstraction in Eq. 4.6, we

245491
Central Library University of Roorkee

get

$$f = \lambda\tilde{a}.[(\lambda\tilde{p}_1.E_1)\ a_1 \ldots a_m, \ldots, (\lambda\tilde{p}_n.E_n)\ a_1 \ldots a_m] \quad \ldots(4.7)$$

Eq. 4.7 suggests that elements of the list in the function body can be handled concurrently in a multiprocessor architecture. However, this would require making full copies of each argument to be given to every processor, and is, therefore, expensive. The copies are required for the twin purposes of matching and reduction if a combined view of the semantics of a pattern-matching lambda abstraction, as understood in Eq. 4.5, is taken. However, matching, unlike reduction, does not require a full copy. It can be done with the help of a structural tag only of the argument. Hence, if we take an **isolated view** of the two issues of matching and reduction, which were mixed in the combined view, then the copying for parallel action can be restricted to appropriate tags only. After the parallel matching is over and a successful expression is available, the complete arguments can be supplied for reduction to the relevant processor only. This way, the matching work is done in parallel using multiple copies of the tags of arguments, and the reduction is done using a single copy of the arguments. To achieve this, we take away the job of matching from the semantics of $(\lambda p.E)$ and pass it on to a separate function called *Match_list*. In the light of this discussion, the translation in Eq. 4.6 is revised as

$$f = \lambda\tilde{a}.[\mathit{Match_list}\ \tilde{p}_1\ \text{Tag_list}\ (\lambda\tilde{p}_1.E_1), \ldots, \mathit{Match_list}\ \tilde{p}_n\ \text{Tag_list}\ (\lambda\tilde{p}_n.E_n)]\ a_1 \ldots a_m \quad \ldots(4.8)$$

where Tag_list = [(*Tag* a_1), ..., (*Tag* a_m)]. The function *Match_list* takes three arguments - a pattern list (pl), a tag list (tl) and a function g (this function is a pattern-matching lambda abstraction in Eq. 4.8). If pl and tl match then it returns g otherwise returns FAIL. It may be noted in Eq. 4.8 that each *Match_list* is supplied with a separate copy of the Tag-list. The parallel applications of *Match_list* may prove successful for ith clause and thus return

$$[FAIL, \ldots, (\lambda \tilde{p}_i . E_i), FAIL, \ldots, FAIL]\ a_1 \ldots a_m$$

which may be interpreted as

$$(\lambda \tilde{p}_i . E_i)\ a_1 \ldots a_m$$

so that the function definition applicable after matching is

$$f = \lambda \tilde{a}.(\lambda \tilde{p}_i . E_i)\ a_1 \ldots a_m \qquad \ldots(4.9)$$

Now the matching work is already over and the lambda abstraction of the kind (λp.E) in Eq. 4.9 can be taken as a modified form of an ordinary abstraction (λv.E). The application ((λp.E) a) is just like any other candidate for a β-reduction, but with an additional care that it is binding a pattern rather than a simple variable. It has nothing to do with matching. This matter is taken up in section 4.2.2.

4.2.1 Matching Algorithm

The functions *Match_list* and *Tag*, introduced in Eq. 4.8, are now being explained in detail. *Match_list* takes two lists of same

length and a function, and matches the lists element by element. It returns a FAIL if the matching fails otherwise returns the function. It is defined as

```
Match_list [] [] g = g        ;indicating a successful completion of
                               the matching process
Match_list (p:ps) (t:ts) g
                    = if (Match p t) then Match_list ps ts g
                      else FAIL                              ...(4.10)
```

The boolean function *Match* performs individual matchings according to the following cases.

<u>Case</u> <u>I</u> : p is a constant pattern

Match p x = if (= p x) then True else False ...(4.11.1)

<u>Case</u> <u>II</u> : p is a variable

Match p x = True ;always ...(4.11.2)

<u>Case</u> <u>III</u> : p is a sum-pattern

Let p = s pc_1 ... pc_r where s is a sum-constructor of arity r and pc_j (j = 1 to r) are the components of p. In this case, the single tag x passed on to *Match* may actually be a tag-list of the kind [s', x_1, ..., x_r]. The function *Match* is then expressed as

Match p x = if (= s s')

then *Match_comp* [pc_1, ..., pc_r] [x_1, ..., x_r]

else False ...(4.11.3)

<u>Case</u> <u>IV</u> : p is a product-pattern

In a product-pattern, only the components are to be matched.

Match p x = *Match_comp* (pc_1, ..., pc_r) [x_1, ..., x_r] ...(4.11.4)

In the last two cases, another function *Match_comp* has been introduced. It is called by *Match* whenever a constructor type pattern is encountered. It takes two lists and uses function *Match* to compare them element by element. The two functions call each other in a mutually recursive manner. The *Match_comp* is defined as

```
Match_comp [] []          = True    ;indicating a successful match of
                                     all the components of a pattern
Match_comp (pc:pcs) [] = False
Match_comp [] (tx:txs) = False
Match_comp (pc:pcs) (tx:txs)
                  = if (Match pc tx) then Match_comp pcs txs
                    else False                              ...(4.12)
```

In the definition of *Match_comp*, it can be seen that the matching is abandoned whenever a particular component fails to match. Now the function *Tag* remains to be defined. This function is necessary for generating a list of tags from the actual parameters for the function *Match_list*. As indicated earlier, tags reduce the amount of copying required for parallel implementation of the matching process. The function *Tag* generates a single tag or a list of tags depending on the data type. A boxed representation of data is shown in Fig. 4.1.

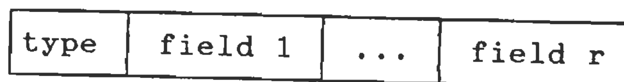


Figure 4.1 Boxed representation of data

Here a sub-field either holds a value or a further data structure having a type field and the sub-fields corresponding to the components of the structure. In terms of this representation, the function *Tag* is defined as

Tag x = **if** (x.type = simple) **then** x.field
        else (Cons (x.type) (*Map Tag* (x.field-list)) ...(4.13)

Definitions 4.10 through 4.13 form a complete algorithm for the matching process. The algorithm forms the basis for the run time multiprocessor implementation of pattern-matching discussed in Chapter 7. The necessary compilation of the left hand sides of a pattern-matching definition into a list of pattern-types is discussed in next chapter. At run time, the compiled pattern-types are matched with the tags of actual parameters to decide the selection of the applicable clause.

4.2.2 Reduction Semantics of λp.E

In the lambda calculus view of pattern-matching, developed earlier, we have taken away the work of matching from the abstraction λp.E and passed it on to the function *Match_list*. Thus the semantics of an application ((λp.E) Q) is to be seen from reduction point of view only, having nothing to do with matching. This perception of λp.E is different from the combined view of matching and reduction [11,109], hence we prefer to call a λp.E as a **pattern-binding lambda abstraction** rather than a pattern-matching lambda abstraction. We could call it simply a lambda

abstraction but the qualifier 'pattern-binding' is to remind that the abstraction binds patterns rather than ordinary variables. In this section, we propose a set of β-rewrite rules for applications involving pattern-binding lambda abstractions.

The application of an ordinary lambda abstraction to an argument is solved by β-reduction rule stating

$$(\lambda x.B)\ Q \ \text{-->}\ [Q/x]\ B \qquad \ldots(4.14)$$

where [Q/x] B means, "the body B with Q substituted for all free occurrences of x in it". The prefix [Q/x] symbolises a substitution operation. If the rule is extended as such to a pattern-binding lambda abstraction, we get

$$(\lambda p.E)\ Q \ \text{-->}\ [Q/p]\ E$$

But the difference, that the abstraction is binding a pattern rather than a variable, cannot be ignored because the argument Q is a structured object having the same structure as p (ensured by the matching process done prior to reduction), and the body E may have references to the components of p which should be substituted by corresponding components of Q and not the full Q. Thus the substitution operation here cannot be 'worded' as simply as for ordinary lambda abstractions.

Revesz [107] has given an operational semantics for functional programs where a set of α and β rules collectively implement the substitution operation without explicitly using it. We follow the same line of thought to circumvent the problem of stating the substitution in the presence of pattern type bound parameters in

lambda abstractions. Such abstractions give rise to an extended syntax of lambda calculus which allows patterns as valid λ-expressions. Taking this extended syntax and ignoring the possibility of a name-clash, the rewrite rules for a β-redex of the kind (λp.E) Q (by induction over E) are

(β1) $(\lambda p.p)\ Q$ --> Q

(β2) $(\lambda p.s)\ Q$ --> s, p and s are different patterns

(β3) $(\lambda p.p_i)\ Q$ --> q_i, $1 \le i \le r$

(β4) $(\lambda p.c_1\ E_1\ \ldots\ E_k)\ Q$ --> $c_1\ (\lambda p.E_1)Q\ \ldots\ (\lambda p.E_k)Q$

(β5) $(\lambda p.A)\ Q$ --> $(\lambda p_1.\ \ldots\ \lambda p_r.A)q_1\ \ldots\ q_r$

(β6) $(\lambda p.E_1\ E_2)\ Q$ --> $((\lambda p.E_1)Q)\ ((\lambda p.E_2)Q)$

(β7) $(\lambda p_a.\lambda p_b.\ \ldots\ \lambda p_x.F)\ Q$ --> $\lambda p_b.\ \ldots\ \lambda p_x.(\lambda p_a.F)Q$

Here A is an atomic expression representing a constant or a variable, and c_1 is a constructor function of arity k. The pattern p and argument Q are matched i.e. if

$p = c\ p_1\ \ldots\ p_r$ then

$Q = c\ q_1\ \ldots\ q_r$

Rules β1 to β4 pertain to a pattern type body where first three are special cases of the general fourth. Rule β5 is for an atomic body which breaks the redex into smaller redexes (using components of Q) for further action. This is necessary in order to find out whether the atomic body refers to some nested component of the bound pattern. Rule β6 deals with an application in the usual way, and multiple abstractions are tackled one by one through the rule β7.

Here are some examples of ß-reduction illustrating the use of above rules.

```
1.   (λ[].[]) []        --> []       ;rule ß1
2.   (λ0.1) 0           --> 1        ;rule ß2
3.   (λ(Cons x y).x)) Cons 4 5 --> 4          ;rule ß3
4.   (λ(Cons x y).Cons x y) Cons 4 5 --> Cons 4 5)    ;rule ß1
5.   (λ(Cons x y).Cons x (Cons x y)) Cons 4 5
       --> Cons ((λ(Cons x y).x) Cons 4 5)
                ((λ(Cons x y).Cons x y) Cons 4 5)       ;rule ß4
       --> Cons 4 (Cons 4 5)            ;rule ß3 and ß1
6.   (λ(Cons x y).SQ x) Cons 4 5
       --> ((λ(Cons x y).SQ) Cons 4 5) ((λ(Cons x y).x) Cons 4 5)
                                                            ;rule ß6
       --> ((λx.λy.SQ) 4 5) 4          ;rule ß5 and ß3
       --> ((λy.(λx.SQ) 4) 5) 4        ;rule ß7
       --> ((λy.SQ) 5) 4               ;rule ß2
       --> SQ 4                        ;rule ß2
       --> 16                          ;operator rule
```

The rules ß1 through ß7 completely specify the ß-reduction process to be used for pattern-binding lambda abstractions. In the general definition of a pattern-matching function (Eq. 4.1), these rules can be used if the clauses are treated as pattern-binding lambda abstractions of the kind

$$\lambda p_{i1}.\lambda p_{i2}. \;\dots\; \lambda p_{im}.E_i, \quad i = 1 \text{ to } n$$

alternatively written as

$$(\lambda \tilde{p}_i.E_i) \text{ where } \tilde{p}_i \equiv p_{i1}, \dots, p_{im}$$

If the pattern-matching process selects kth clause then we are left with the reduction of an expression of the kind

$$(\lambda \tilde{p}_k . E_k)\ a_1\ \ldots\ a_m$$

where $a_1, \ldots, a_m$ are m arguments. This application can be reduced according to the ß-rules developed here.

4.3 CONCLUDING REMARKS

A function defined differently for different cases of its parameters is expressed in functional programming through pattern matching. An application of such a function to actual parameters requires an additional step of selecting the applicable definition besides instantiation (the reduction). The selection is done on the basis of a successful structural match between the formal parameters of a clause and the actual parameters.

In this chapter, pattern-matching has been interpreted in an extended lambda calculus which allows list manipulations. The interpretation suggests a simultaneous matching of all the clauses, and is different from that in enriched lambda calculus [11,109] which has a sequential operational semantics. For parallel action, the nuisance of making multiple copies of actual parameters is greatly reduced by the observation that matching requires only structural tag of arguments and not their full form. Accordingly, an isolated view of the semantics of $(\lambda p.E)$ has been proposed where the matching for clause selection is done in parallel followed by a ß-reduction on the selected clause only.

It helps in keeping the dynamic graph size under check as compared to implementations which proceed with matching and reduction together [11,109,110] keeping all the clauses active. The above ß-reduction is performed through a modified set of rules which have been proposed to take into account the presence of patterns as bound parameters.

Each clause in a pattern matching definition is considered to be a supercombinator definition. As indicated in chapter 3, supercombinators need some pre-processing because they do not have simple or fixed reduction rules. Accordingly, the next chapter develops a scheme for compiling a definition body into a structured director string term. The structured directors introduced in these terms allow mechanical implementation of the semantics of the modified ß-reduction specified in this chapter. The template instantiation of supercombinators through SDS terms is greatly simplified due to their fixed term-reduction rules discussed in chapter 7.

CHAPTER 5

COMPILATION OF PATTERN-MATCHING DEFINITIONS

In the previous chapter, a lambda calculus interpretation for pattern-matching was developed. It divided the whole issue of implementing pattern-matching into two separate parts: one concerned with choosing the applicable clause through a matching process, and the other with the reduction of the chosen expression through a set of modified ß-rules. This chapter develops an algorithm for compiling a pattern-matching definition so as to enable efficient reduction at run time. For the matching part, the formal parameters (patterns) in each clause are compiled into a sequence of pattern-types which are matched with the actual parameters at run time. For the reduction part, the body of each clause is compiled into a Structured Director String (SDS) term which is used in template instantiation. The two informations are put together in the form of a pair for every clause, so that a pattern-matching definition having n clauses gets compiled into a list of n pairs.

Section 1 of the chapter deals with the total compilation process and it comprises of two sub-sections corresponding to the two parts of compilation. In section 2, the compilation is illustrated through an example. The utility of SDS terms in simplifying reduction is also shown. Section 3 deals with

functions defined through guarded equations. Finally, in section 4, conclusions and link with the next chapter are given.

5.1 COMPILATION PROCESS

Given a pattern-matching definition (general form in Eq. 4.1), the compilation produces a list of pairs written as

$$L = [P_1, P_2, \ldots, P_n]$$
$$\text{where } P_i = \langle S_i, t_i \rangle \;, \quad i = 1 \text{ to } n$$

A sequence S_i and term t_i, forming a pair P_i, are expressed as

$$S_i = T(p_{i1}), T(p_{i2}), \ldots, T(p_{im}), \text{ and}$$
$$t_i = SDS(E_i) \qquad \ldots(5.1)$$

with $T(p_{ij})$ as the pattern-type of a pattern p_{ij} and $SDS(E_i)$ as the SDS term representation (to be discussed shortly) of the expression E_i. At the time of execution, the sequences S_i (i = 1 to n) are used in the process of matching whereas the SDS terms t_i support reduction. The list L constitutes a field (called the SDS field) of a definition code (D-code) structure. Later (next chapter), another field, called the header, is added to complete the D-code structure. The various informations in this code, filled at compile time from function definitions, are aimed at simplifying the multiprocessor reduction process. The discussion now is regarding compilation for the two items shown in Eq. 5.1.

5.1.1 Compilation For Clause Selection

Clause selection for an application involving a pattern-matching

function is based on a structural match between the formal and actual parameters. From the matching point of view, the patterns, used as formal parameters in a definition, can be divided into three types:

(i) Constant

(ii) Irrefutable

(iii) Refutable

The constant types include all constant patterns. Although they are refutable but they form a separate type to distinguish them from refutable constructor type patterns.

Irrefutable types [11] always match, hence it is beneficial to classify patterns as irrefutable, wherever possible, so as to save matching effort on them. An irrefutable pattern is defined as follows [11]:

A pattern p is irrefutable if it is

(i) either a variable v

(ii) or a product-pattern of the form $(t\ p_1\ \ldots\ p_r)$ where $p_1, \ldots, p_r$ are irrefutable patterns

In this definition, t is a product-constructor of arity r.

Refutable types include all constructor-patterns which are not irrefutable. They present the maximum difficulty in matching because each component of these patterns has also to be matched before deciding the outcome. All sum-constructor patterns or product-patterns having some refutable components fall in this

category.

The aim of compilation is to produce a sequence S from the formal parameters of a definition. The set from which the elements of these sequences are drawn, consists of all constants, an identifier 'Ir' (for irrefutable), and all data constructor functions. The elements of a sequence may themselves be sequences because patterns can be arbitrarily nested. The compilation algorithm is specified through the following definition of a function *pat-type* which delivers the pattern-type of a pattern:

Definition 5.1 : pat-type

---

pat-type(C) = val C ;val is a tag and C is a constant,

pat-type(v) = Ir ;v is a variable

pat-type(c p_1 ... p_r) = (c, *pat-type*(p_1), ..., *pat-type*(p_r)),
c is a sum-constructor

= **let** pt_1 = *pat-type*(p_1)
...
pt_r = *pat-type*(p_r) **in**
if (pt_1 = 'Ir') AND ... AND (pt_r = 'Ir')
then 'Ir' **else** (c, pt_1, ..., pt_r),
c is a product-constructor

It can be seen in the definition of *pat-type* for a sum-constructor pattern that some of its components could get listed as irrefutable leading to saving in matching costs when it comes to matching the components. The complete sequence S_i for the ith clause of a definition is generated by mapping the function *pat-type* on the list of formal parameters as

$$S_i = \mathit{Map\ pat\text{-}type}\ [p_{i1}, \ldots, p_{im}] \qquad \ldots(5.2)$$

Eq. 5.2 combined with Def. 5.1 for *pat-type* completely specifies the compilation for clause selection.

5.1.2 Compilation For Reduction

The compilation of the expression on the right-hand side of a definition (called its body) into SDS terms forms a major part of the total compilation process. It is even continued into next chapter for dealing with bodies having local definitions. The compilation here converts a body into a tree structure having director string annotations at the nodes. The idea is to instantiate functions using their SDS terms as templates. The annotations in these terms will provide a simple and mechanical instantiation through 'term reduction rules' developed in chapter 7. The complete idea is based on the reduction semantics of an application ((λp.E) Q) specified in section 4.2.2 of the previous chapter. This has been illustrated through an example in section 5.2. The compilation scheme for obtaining SDS terms is now discussed starting with a description of atomic directors used by Kennaway and Sleep in the formation of DS terms [71].

5.1.2.1 Atomic directors

The director strings, originally introduced by Kennaway and Sleep [70], were discussed in Chapter 3. The same authors have developed a scheme of abstracting out variables from a lambda

expression to convert it into a variable-free form called the Director Strings Terms (DST) [71]. It is a more compact variable-free representation [11] than the SK combinators, and simplifies the process of β-reduction by avoiding the need to visit every leaf or sub-graph of body to look for a free occurrence of the bound variable. It uses simple atomic directors (/\, /, \, -, !, #) for indicating the direction of movement of an argument at a node. As an example, the λ-expression (λf.λx.f(x x)) gets translated into a DS term through following steps [71]:

$$
\begin{aligned}
\lambda f.\lambda x.f(x\ x) &= \lambda f.(\backslash @_2,\ f,\ \lambda x.(x\ x)) \\
&= \lambda f.(\backslash @_2,\ f,\ (/\backslash @_2,\ \lambda x.x,\ \lambda x.x)) \\
&= \lambda f.(\backslash @_2,\ f,\ (/\backslash @_2,\ (!@_1,\ \blacksquare),\ (!@_1,\ \blacksquare)) \\
&= (/\ \backslash @_2,\ \lambda f.f,\ (/\backslash @_2,\ (!@_1,\ \blacksquare),\ (!@_1,\ \blacksquare))) \\
&= (/\ \backslash @_2,\ (!@_1,\ \blacksquare),\ (/\backslash @_2,\ (!@_1,\ \blacksquare),\ (!@_1,\ \blacksquare)))
\end{aligned}
$$

where $@_2$ and $@_1$, appearing immediately at the end of a director string, stand for a binary and a unary node respectively and the symbol '■' represents a hole. In the final form, each node has annotations, in the form of directors, for the arguments to follow at run time. DS term reduction rules [71] imitate the β-reduction of lambda calculus.

We have called the directors used in DS terms as atomic directors in view of structuring introduced in the directors. The new directors, named as structured directors, have been used in an abstraction scheme for converting a pattern-matching definition body into a 'pattern free' form called the Structured Director String Term (SDST or SDS term). The need for structured directors

arises because the atomic directors cannot distribute the components of a structured argument and neither can they encode the intention of ß-reduction rules meant for pattern-binding lambda abstractions.

5.1.2.2 Structured directors

In DST, a binary application node is represented as a 3-tuple term where the first element gives the director string annotation of the node and the other two, which may be further tuples, specify the left and right sub-trees rooted at this node. The unary nodes (leaves) are represented as pair terms. At the time of reduction, an argument follows the route indicated by the left-most director which is deleted from the string after the argument has passed (this is a mechanical way of ß-reduction similar to SK reduction). An empty director string at a node means that the sub-tree rooted at this node needs no more substitutions.

The above view of director string serves well as long as the arguments are simple objects such as primitive data values or function identifiers. Pattern-matching definitions contain patterns as formal parameters with bodies referring to them or their constituents which may contain arbitrary nesting of further patterns. Such definitions expect structured arguments and, therefore, a compilation similar to that of DST needs directors which can even point to the most deeply embedded part of the argument whenever required by the body.

We extend the language of DST by using structured directors to

take care of the situation and call the resulting terms as Structured Director String (SDS) terms. One of these directors, named as **list director**, is made up of a list of directors where each may be atomic or further structured. Notationally, its elements are separated by commas and are enclosed within curly brackets. Though it is a list, it constitutes a single element of a director string. Elements of this list refer to the various parts of a argument structure. If an argument A can be represented as

$$A = c\ a_1\ a_2\ \ldots\ a_r$$

where c is the constructor of its structure type, then the list director which distributes its components in a definition body would consist of r directors written as

$$d = \{d_1,\ d_2,\ \ldots,\ d_r\}$$

where a director d_i refers to an a_i. Any of the d_i may be a further list director, and its elements (d_{i1}, d_{i2}, ... etc.) would refer to the components of a_i. As an example, if the structure is a list represented by a CONS cell in terms of its head and tail, then its list director would consist of two atomic directors (chosen according to the need), and the various possibilities include

d = {/,\} | {!,#} | {/,/} | {/\,-} etc.

They have their usual meaning, e.g. the first means the head of the list is to proceed to left and the tail to the right.

A list director suggests that the structure be broken into pieces and the parts be distributed as indicated by its elements. If any element is again a list director then the corresponding part be further broken into sub-parts and distributed accordingly. In many situations, this may lead to wasteful work. For example, let us consider an identity function (*Id* p = p) defined with a complex and deeply nested pattern p in the formal parameter place. Now suppose the function body is compiled using a list director which meticulously expresses all the nesting in the pattern, then on application to a matched argument, the director will unnecessarily force the breaking of the argument structure which will have to be re-assembled from its pieces. All this was avoidable because the identity function never wanted the argument to be broken and then re-built. An optimisation is being introduced in the scheme by proposing another structured director called the **pattern director**.

A pattern director transports a structured object without tearing it apart. It amounts to postponing the decision of breaking. Preference to use pattern director in place of list director, as far as possible, is in the spirit of laziness (laziness in sub-dividing a structure) which is the key to avoiding wasteful work. Let us consider the following definition of a function which attaches the head of a list in the front:

Add_head (CONS x y) = CONS x (CONS x y)

The list-director approach will fail to see that the second CONS cell on right hand side is same as the formal parameter, and thus will produce a code to re-build it from the pieces x and y. The

SDST representation (actual compilation scheme in next section) using list director is given as

({/\,\}: , (!@$_1$, ■), (/ \: , (!@$_1$, ■), (!@$_1$, ■)))

where ':' symbolises a CONS node. It may be seen that at the first CONS node, the list-director {/\,\} will break an applied argument into x and y and send x both ways and y to the right. On the left, x is used in a hole while on the right, another CONS cell is built from x and y. In contrast to this, the use of pattern director would replace that complete second CONS cell by a hole and thus avoid rebuilding it. This is reflected in the SDST code given below (method of obtaining it will be discussed in next section):

({/\}: , ({!,#}@$_1$, ■) , ({!}@$_1$, ■))

where {/\} and {!} are pattern directors (a pattern director is simply an atomic director but enclosed in curly brackets). The director {/\} suggests that complete argument list (or its pointer) be sent to both left and right sub-trees. In the right sub-tree, the list is substituted as such and on the left, it is broken (because of the presence of the list-director {!,#}) into head and tail out of which tail is discarded and the head is substituted. Thus only one new CONS cell is created which was unavoidable.

In our compilation of pattern-matching definitions, the director strings consist of both types of directors - atomic and structured. The ideas of new directors are summarised in the

following definition of a director string (ds_of_length_n):

Definition 5.2 : ds_of_length_n

```
------------------------------------------------------------------
ds_of_length_n   ::= director ds_of_length_(n-1)

director         ::= atomic | structured

atomic           ::= / | \ | /\ | - | ! | #

structured       ::= list_director | pattern_director

list_director    ::= {director, director} | {director,
                      list_director}

pattern_director::= {atomic}
------------------------------------------------------------------
```

Structured directors can be linked to the meaning of the ß-reduction rules for pattern-binding lambda abstractions developed in section 4.2.2. A close scrutiny of these rules (ß1 - ß7) reveals that they are specifying the 'treatment' to be given (at run time) to the argument i.e. whether it should be broken into its constituents, or it should be discarded, or it should be presented to both parts of an application to decide further treatment, etc.. The nature of the treatment (or the ß-rule to be used for a redex) is solely governed by the type of lambda-abstraction or more appropriately by the correspondence of the body with the bound pattern. It does not depend at all on the argument. The assertion is based on the assumption that a reduction takes place after a pattern-matching process and hence the argument must have the same built as the bound pattern. Since a clause in a pattern-matching definition is like a lambda-abstract, a complete 'prescription' of the treatment can be prepared from the definition at the compile time itself. This saves the botheration of finding, at run time, which ß-rule to apply to reduce a redex.

It is a sizable saving considering that the search for applicable ß-rule is quite expensive and may have to be done recursively for the same redex.

The compilation aims at achieving the above goal so that the reduction process, at run time, becomes a purely mechanical exercise, a thing most suited to the temperament of a machine. It uses the language of SDS terms for prescribing the treatment. The body of each clause in a definition is compiled into a tree structure (SDS term). At each node in the tree, directors are attached indicating the treatment to be given to arguments taking a tree-walk during reduction.

It may be noted that the structured directors have all the necessary ingredients to indicate the argument treatment specified by the various ß-rules. They can specify not only the direction of movement at a node but also the decision whether to break it or not. A list director indicates breaking and the movement of the pieces whereas the pattern director guides a structured object without breaking. The atomic directors indicate the movement directions only, for simple objects. The next two sections develop pattern-abstraction needed for compiling an expression into its SDS term.

5.1.2.3 Pattern-abstraction

A SDS term for a definition body is obtained by abstracting out all the pattern parameters (occurring on the left-hand side of definition) from it. The abstraction is performed through

operators called the A-operators whose services are requested by a function *trans*. Thus for the first clause in the general pattern-matching definition (Eq. 4.1), the SDS term for E_1 is obtained as follows:

$$trans(f\ p_{11} \ldots p_{1m} = E_1) = A_p[E_1]$$
$$\text{where } A_p = A_{p11} \ldots A_{p1m} \qquad \ldots(5.3)$$

Here $A_{p_{ij}}$ are abstraction operators having right association property meaning that abstractions are performed from inside out i.e. the right-most parameter (p_{1m}) is abstracted out first. In this way, the top layer of abstraction corresponds to the first argument to be substituted. The expression on which an abstraction operator operates, is enclosed within square brackets. This abstraction is different from the variable-abstraction of DST in that it abstracts out a pattern (as a whole) rather than a variable, making SDST a 'pattern free' representation.

Before developing the abstraction rules for A-operators, the syntax of expression forming a definition body must be decided. To simplify the matter a little, we start with a lambda type syntax. It is assumed that the basic building blocks for an expression in the syntax are constants (including built-in operators), variables, function names and constructor functions; and the building mechanisms are Curried application and structuring through constructor functions. The syntax is summarised in Fig. 5.1 where c is a constructor function of arity r.

$E ::= A \mid$	an atom
$(E_1\ E_2) \mid$	application
$(c\ E_1\ E_2\ \dots\ E_r)$	structuring

Figure 5.1 Simple syntax of an expression forming a definition body

The syntax is later enriched by allowing guards (in section 5.3 of this chapter) and recursive and non-recursive local definitions (in the next chapter). Expressions in higher level functional language can be translated into the enriched syntax by an upper layer of compilation.

The simple syntax in Fig. 5.1 gives rise to two types of nodes in the tree representation of an expression - apply nodes and constructor nodes. The constructor nodes need not necessarily be binary. The expressions thus acquire a general r-ary tree structure, and a node is written as

node = (nt, node_1, node_2, ... , node_r)

where 'nt' stands for a node-type symbol. It may be $@_1$ indicating a leaf, $@_2$ indicating an application or a constructor indicating structuring of some particular type.

The intention of SDS representation is to attach director strings at the nodes in an expression tree through the process of abstraction. A term in this notation is either an atom or a tuple defined as

term ::= atom | (D_r nt, term_1, ... , term_r)

An atomic term can be a constant, a function identifier or a hole (created during abstraction for an occurrence of a parameter in the expression). D_r is a string of r-ary directors ($r \geq 1$). Tuple terms are written as (r+1)-tuples, such as pairs (r=1) for unary nodes, triples (r=2) for binary nodes, and so on. It may be noted that to indicate argument flow at a r-ary node, the set of atomic directors given in Def. 5.2 must be extended to include general r-ary directors also which although are difficult to show symbolically but can be easily implemented through bit codes.

Having decided on the syntax of expressions and the structure of SDS terms, we turn to the development of the abstraction rules for an operator A_p (Eq. 5.3) which operates on an expression to abstract out a pattern p. As there are three types of patterns (Def.4.1), the set of rules is divided into three cases and each case lists the rules for different syntactical forms of E (Fig. 5.1). In developing the rules, an idea of the variable-set of a pattern is needed because howsoever deep may be the nesting in a constructor-pattern, its ultimate components are variables (sometimes constants) by which it is referred in the body of a definition. Variable-set v(p) of a pattern p is a set consisting of all the variables occurring in it, and is defined [11] as

$$
\begin{aligned}
v(p) &= \{x\} && \text{, p is a variable x} \\
&= \{\ \} && \text{, p is a constant} \\
&= \bigcup_{i=1}^{r} v(p_i) && \text{, p is a constructor pattern of arity r}
\end{aligned}
\qquad ...(5.4)
$$

5.1.2.4 Pattern-abstraction rules

Let p be a pattern to be abstracted out from an expression. It would be referred as the 'abstractee'.

Case I: p is a constant

Referring a constant type parameter in an expression makes no sense (it is required for matching purposes only) and therefore the abstraction process can always assign a discarding director for such a parameter. Thus the rules for abstracting out a constant k from an expression are

(A1) $A_k[A] = (\#@_1, A)$

(A2) $A_k[E\ F] = (-@_2, E, F)$

(A3) $A_k[c\ E_1\ \dots\ E_r] = (\#@_1, c\ E_1\ \dots\ E_r)$

Rule A1 is for an atomic expression and it simply discards the argument at a unary node. Rule A2 translates an application into a triple, assigning '-' for discarding, and terminates the abstraction. Rule A3 deserves a closer look. Here we do not create the required node (as was done in rule A2), rather the decision is postponed by assigning a 'unary discard'. Hence the abstraction results in a pair term and the expression is left untouched for some later abstraction to explore. This has been purposely done to preserve laziness which, in this context, suggests - do not create a structural node unless essential.

Case II: p is a variable v

Abstraction of a variable leads to SDS terms with atomic directors

only. The rules are similar to those of DST [71]. In the context of the chosen syntax of an expression, the rules are

(A4) $A_v[A]$ = $(\#@_1, A)$, A is a constant, or a variable other than v (i)

= $(!@_1, \blacksquare)$, A is a variable same as v (ii)

(A5) $A_v[E\ F]$ = $(d_2@_2, A_v[E], A_v[F])$

(A6) $A_v[c\ E_1\ \ldots\ E_r]$

= $(\#@_1, c\ E_1\ \ldots\ E_r)$, v does not occur in any E_j, j = 1 to r (i)

= $(d_rc, A_v[E_1], \ldots, A_v[E_r])$, otherwise (ii)

Some new features of the abstraction scheme appear in these rules. Rule A4 discards the argument at a unary node if the atomic expression is different from the variable under abstraction (first equation). In case it is same, a pair, with second field as a hole, is created. A hole acts as an empty space to be filled by an argument at run time. It marks a successful end to the abstraction 'journey'.

In rule A5, the application gets translated into a triple and the operator A_v moves inside to operate on the components of application after assigning a director d_2. Leaving behind a director when made to move in, is a general characteristic of A-operators. It may be noted that this rule encodes the semantics of the reduction rule ß6 for an application. The rule is used with an optimisation as follows [71]:

$$A_v[E\ F] = (/\backslash@_2,\ A_v[E],\ A_v[F]), \quad v \text{ occurs in both E and F}$$

$$= (/@_2,\ A_v[E],\ F), \quad v \text{ occurs in E only}$$

$$= (\backslash@_2,\ E,\ A_v[F]), \quad v \text{ occurs in F only}$$

$$= (-@_2,\ E,\ F), \quad v \text{ does not occur in E or F}$$

The optimisation is that the abstraction operator is moved only to that part of the applicative expression where the abstractee occurs. It will henceforth be referred as **movement-as-per-need** optimisation.

Rule A6 deals with data construction in a manner similar to that for application except that it leaves the constructor expression as such if v does not occur in it (first equation). If it occurs then a construction node of type c, with a director d_r attached, is created and abstraction is made to move in. The movement-as-per-need optimisation is applicable here also.

<u>Case</u> <u>III</u>: **p is a constructor-pattern**

Let $p = s = c_s\ s_1\ s_2\ \ldots\ s_k$. This case makes use of structured directors. It may be recalled that in order to prevent unnecessary structure breaking only pattern directors should be used, as far as possible. Taking this into account, the rules are framed in such a way that the decision to break a structure (i.e. to use a list-director) is continuously postponed till a variable is encountered in the body. The strategy is sometimes able to completely eliminate the use of list directors. The abstraction rules are

(A7) $A_s[A]$ $= (\{\#\}@_1, A)$, A is a constant, or a variable $x | x \notin v(s)$ (i)

$= (\{dA_{s1}[A], dA_{s2}[A], \ldots, dA_{sk}[A]\}@_1, \blacksquare)$,

A is a variable $x | x \in v(s)$ (ii)

(A8) $A_s[E\ F] = (\{d_2\}@_2, A_s[E], A_s[F])$

(A9) $A_s[c\ E_1\ \ldots\ E_r]$

$= (\{!\}@_1, \blacksquare)$, $c\ E_1\ \ldots\ E_r = s$ (i)

$= (\{\#\}@_1, c\ E_1\ \ldots\ E_r)$, No variable of v(s) appears in any E_j (j=1 to r) (ii)

$= (\{\{\#\}^{i-1}, \{!\}, \{\#\}^{r-i}\}@_1, \blacksquare)$,

$c\ E_1\ \ldots\ E_r = s_i$, i=1 to k (iii)

$= (\{d_r\}c, A_s[E_1], \ldots, A_s[E_r])$, otherwise (iv)

In the above notation, d_r is a r-ary atomic director, $\{d\}^l$ stands for a l-list of the director {d}, and $dA_{s_i}[A]$ stands for the director resulting from the abstraction $A_{s_i}[A]$.

Rule A7, for atomic expressions, rejects the argument if it is unwanted (first equation), but if the expression is a variable x belonging to the variable-set v(s), the rule prepares to insert the relevant component of argument into a hole (second equation). The preparation is concerned with composing a list director whose elements are drawn from the process of abstracting out the components of the abstractee from the atomic expression. The translation can be seen to have correspondence with the β-reduction rule β5.

Rule A8, for application, remains as in other cases except that a pattern director is used. Movement-as-per-need optimisation is

applicable with the modification that now the need for argument in E or F is determined by the presence/absence of a member from v(s) in them.

Rule A9, for constructor expressions, emphasises that best efforts are made to avoid the creation of a constructor node. If the full expression matches the pattern under abstraction (first equation) or any of its components (third equation) then appropriate director is assigned to insert the full argument or its required component into a hole. These cases reflect the semantics of reduction rules ß1 and ß3 respectively. If no member from v(s) occurs in the constructor expression then the argument is unwanted and hence discarded (second equation). It corresponds to the situation of rule ß2. As a last resort when no such matching of the full expression is possible then a constructor node is created and abstraction operator is moved to the components of the expression (fourth equation). It corresponds to the general rule ß4. Further, the need for movement to a particular component is again determined by the presence/absence of a member from v(s) in that component.

It may be observed that the pattern-abstraction rules generate SDS terms in such a way that the list directors are used only in the end when an atom is encountered. At that time, no alternative other than to break the argument, for substitution, is left.

The above cases deal with the abstraction of a single parameter from an expression. However, a function definition may have

several parameters leading to multiple abstractions as in Eq. 5.3. In such a case, the abstractions are done one after the other starting with the right-most one (right associative manner). Each abstraction operator gets the result of the preceding abstraction as its input. These intermediate expressions are not strictly according to the syntax in Fig. 5.1 because of the introduction of director strings and holes. Hence, the abstraction rules A1-A9 cannot be used directly. Following are the rules for dealing with various situations of this type:

(A10) $A_p[(D_1@_1, E)] = (d_1D_1@_1, A_p[E])$

(A11) $A_p[(D_2@_2, E, F)] = (d_2D_2@_2, A_p[E], A_p[F])$

(A12) $A_p[(D_rc, E_1, ..., E_r)] = (d_rD_rc, A_p[E_1], ..., A_p[E_r])$

where D_1 and D_2 represent the already existing director strings, and d_1 and d_2 stand for the added directors (structured or atomic). These rules have a common approach - add an appropriate director to the left of the existing string and move the abstraction in according to the need. However, if a sub-expression is a hole then there is no need to apply further abstraction to it. The correspondence of these rules can be traced to rule β7.

5.2 AN EXAMPLE

The example here illustrates the compilation process and shows its utility in making the reduction process simple and purely mechanical. It is the example of the function *Add_head* introduced earlier in section 5.1.2.2. The function, defined as

Add_head CONS x y = CONS x (CONS x y),

has one parameter which is a constructor-pattern. The compilation produces a pair given as

$$P_1 = \langle S_1, t_1 \rangle, \quad \text{where } S_1 = \textit{pat_type} \text{ (CONS x y)}$$
$$t_1 = \textit{trans}(\text{Add_head} \ldots)$$

Using Def.5.1 for the function *pat_type* we get

$$S_1 = (\text{CONS}, \textit{pat_type}(x), \textit{pat_type}(y))$$
$$= (\text{CONS, Ir, Ir})$$

The sequence S_1 has only one element because the definition had one parameter but due to a constructor-pattern, the single element itself is a sequence of three elements. Actually, the example has only one clause meaning that no pattern-matching is involved, hence compilation will not produce the sequence of pattern-types needed for matching. It has been done here just for illustration. The other part of the pair P_1 which is the SDS term for the definition body, is obtained through pattern-abstraction using Eq. 5.3 as

$$t_1 = A_p[\text{CONS x (CONS x y)}]$$

where p = CONS x y. The constructor in p is a sum-constructor and v(p) = {x,y}. The input expression for A-operator is a constructor expression, and therefore rule A9 is applicable. By a trial to match the alternatives in that rule, fourth equation gets selected and its use gives

$$t_1 = (\{/\backslash\}:,\ A_p[x],\ A_p[\text{CONS } x\ y])$$

The both ways pattern director '{/\}' is assigned because members from v(p) are referred in both the sub-expressions. For abstracting further, we have to use rule A7 for left part and A9 for the right part of the created constructor node ':'. On the right side, the decision is straight forward for using first equation of rule A9. On the left, the second equation of A7 is to be used because $x \in v(p)$. Applying these rules, we get

$$t_1 = (\{/\backslash\}:,\ (\{dA_x[x],\ dA_y[x]\}@_1,\ ■),\ (\{!\}@_1,\ ■))$$

Now,

$$A_x[x] = (!@_1,\ ■),\ \text{and,}$$

$$A_y[x] = (\#@_1,\ x),\quad \text{by rule A4}$$

Therefore,

$$dA_x[x] = !,\ \text{and,}$$

$$dA_y[x] = \#$$

This leads to

$$t_1 = (\{/\backslash\}:,\ (\{!,\#\}@_1,\ ■),\ (\{!\}@_1,\ ■))$$

For a simpler and compact notation, we might write pair terms without comma and brackets. Also the symbol $@_1$ can be dropped whenever there is no confusion so that

$$t_1 = (\{/\backslash\}:,\ \{!,\#\}■,\ \{!\}■)$$

Combining the final results of the two parts of the compilation,

the total shape of the SDS field of the D-code for the given definition is

<(CONS, Ir, Ir), ({/\}:, {!,#}■, {!}■)>

As stated earlier, the SDS field in this example will not have the the sequence S_1 of pattern-types because there is no pattern-matching involved. Instead it has only the SDS term t_1.

Having done all the hard work of compilation, our function is ready to be applied to an argument in a mechanical way. The SDS tree of the function applied to a list (CONS 1 5) is shown in Fig. 5.2. The application is reduced by allowing the argument list to take a tree-walk over the SDS tree, and doing at each node exactly what the leading director of the director string prescribes (the question of selecting the appropriate ß-rule at each step and doing accordingly has been resolved at compile time and encoded into the language of SDS terms). The term reduction rules developed in chapter 7 is a formal representation of this reduction process.

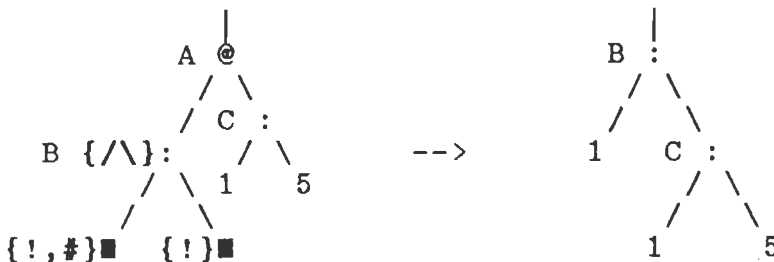


Figure.5.2 SDS tree for the body of *Add_head* and its reduction

The result of reduction is also shown in Fig 5.2. Had we followed the ß-reduction rules (without compiling to SDS terms), the reduction would have proceeded as follows (taking the given definition as a pattern-binding lambda abstraction):

```
(λ(CONS x y).CONS x (CONS x y))(CONS 1 5)
-->  CONS (λ(CONS x y).x)(CONS 1 5)
          (λ(CONS x y).CONS x y)(CONS 1 5)        ;by ß4
-->  CONS 1 (λ(CONS x y).CONS x y)(CONS 1 5)      ;by ß3
-->  CONS 1 (CONS 1 5)                            ;by ß1
```

In the above reduction, the rule selection procedure is invoked three times which is costly because selection involves trials of structure matching between the bound pattern and the body of abstraction.

5.3 GUARDED EQUATIONS

Some functional languages allow the use of guards [11] in both types of definitions viz. pattern-matching types or the simple ones. With this feature, a clause/definition can have several alternative right-hand sides, the choice among them being decided by guards. A guard is a boolean expression which involves the parameters of the definition. It is written after each right-hand side separated by a comma. An example of a guarded equation is the greatest common divisor function which is written as

gcd a b = *gcd* (a-b) b, a > b

= *gcd* a (b-a), a < b

= a, a = b

The definition has three right-hand sides distinguished by the conditions on the parameters 'a' and 'b'. The guards have nothing to do with the matching of patterns but they do appear often in pattern-matching definitions. A clause gets selected after the matching process, and if it has several right-hand sides then further selection is made by testing the guards. The guards are tested sequentially in the same order as given in the definition.

A general form of a guarded equation is written as

$$
\begin{aligned}
f\ p_1\ p_2\ \cdots\ p_m = \ & E_1\ ,\ G_1 \\
& \cdots \\
& E_n\ ,\ G_n
\end{aligned}
$$

where E_i is an expression and G_i is a boolean valued guard. The semantics of the body of this definition can be expressed as [11]

if G_1 **then** E_1 **else** (... (**if** G_n **then** E_n **else** FAIL)...)

Thus the body is a conditional expression and it can easily be compiled into a SDS term following the rules already given. In order to stick to the syntax for expressions (Fig. 5.1), the above semantics would be expressed as

IF G_1 E_1 (... (IF G_n E_n FAIL) ...)

where IF is a built-in operator and FAIL is a constant indicating the failure of a guard. Sometimes the last guard G_n is omitted giving a sense of 'otherwise'. In such a case, the inner-most conditional is replaced simply by E_n.

5.4 CONCLUDING REMARKS

The compilation of supercombinator definitions involving pattern-matching is divided into two parts : one for matching work where a list of pattern-types of the formal parameters is prepared, and the other for reduction of an application where the definition body is compiled into a SDS term. The two informations together form a pair for a clause in the definition. The complete definition gets compiled into a list of pairs whose length is equal to the number of clauses. The list constitutes the SDS field of a D-code record structure for the definition. Another field of the code called the header is introduced in next chapter.

The list of pattern-types, generated through a function *pat-type*, relieves some matching work by utilising the ideas of refutable and irrefutable patterns. This list will be used during reduction when the matching algorithm, given in section 4.2.1, is executed.

The SDS term representation for a definition body, obtained through a scheme of pattern-abstraction, serves as a means of encoding the intention of β-reduction specified in the last chapter (section 4.2.2). The structured directors (list director and pattern director) introduced in these terms extend the DST notation [71] so that directors acquire an additional ability to refer to the components of a structured argument. The laziness introduced by the use of pattern directors prevents unnecessary structure breaking.

The SDS terms are expected to simplify and mechanise the template-

instantiation of a supercombinator definition. The hope is based on the fact that the terms have fixed and simple reduction rules referred as term reduction rules (to be discussed in chapter 7). There is a possibility of coding these rules into hardware thus avoiding a level of software interpretation. The G-code of G-machine [88-89] or the imperative code of Flagship [95] do not seem to offer this possibility.

The pattern-abstraction scheme (abstraction rules A1 - A12), for obtaining SDS terms, has been coded into a Turbo-C software package. The software has been run successfully on IBM PC and it yielded correct codes for several example cases (including Quicksort function) tried during test runs. The laziness of the algorithm in assigning list directors has been confirmed.

The next chapter is a continuation of the compilation of definition bodies into SDS terms where the syntax of an expression used as a body has been extended to allow local definitions through let and letrec blocks.

CHAPTER 6

COMPILATION OF LET AND LETREC DEFINITIONS

In the previous chapter, a scheme for compiling pattern-matching definition bodies into a pattern-free SDS term representation for subsequent use in multiprocessor reduction has been developed. The syntax of expression forming the body was restricted to atoms, Curried applications and data structuring (Fig. 5.1). In the present chapter, the syntax is being extended to include local definitions through let and letrec blocks, and correspondingly, the SDS term structure also gets enriched. In the new structure, the local variables occurring in the main definition body are replaced by pointers instead of holes. Also, a concept of a context-list, forming an additional limb of the SDS term structure, has been introduced. The context-list contains compiled versions of the local definition bodies. Pointers and context-list together give the SDS term a graph structure. In the analysis presented, formal parameters of a definition have been restricted to simple variables only and not patterns so as to avoid mixing up of the issues of local definition and patterns. However, the compilation can be combined with that in the previous chapter to generalise it to pattern type parameters.

The chapter commences by an explanation of the issues involved in dealing with local definitions. Section 2 then gives some

additional abstraction schemes needed. Section 3 deals with the actual compilation process. In section 4, the D-code structure for a definition has been further expanded. Here methods of extracting some more useful information from a function definition, such as computability analysis, have been presented. Concluding remarks are finally given in section 5.

6.1 DEALING WITH LOCAL DEFINITIONS

Local definitions are introduced in a definition to express sharing of sub-expressions. They are written using **let** or **letrec** blocks. In the following definition for a function f

f v = (**let** x = E **in** F)

the body is a let-expression in which x is a local variable, E a local body, and F the Main. The variable v is a parameter of the definition.

A let-expression binds an occurrence of a local variable in the Main to the local body. In lambda calculus, the semantics of this expression is denoted as

$(\lambda x.F)\ E$

Under ß-reduction, it has the same meaning as the original let-expression. Using this semantics, the definition of f can be expressed as

$$f\ v = (\lambda x.F)\ E \qquad \ldots(6.1)$$

where the definition body is a ß-redex. Inspired by Eq. 6.1, the

abstraction process for getting SDS term of the definition could be expressed through the function *trans* as

$$trans(f\ v = (\lambda x.F)\ E) = A_v[(A_x[F])\ E] \qquad \ldots(6.2)$$

The operator A_x abstracts out the local variable from the Main F and thus generates directors and holes meant for the substitution of local body E in it. Actual substitution is done later at run time. In Eq. 6.2, the β-redex $((\lambda x.F)\ E)$, of Eq. 6.1, has been faithfully reproduced and preparation made, through A_x abstraction, for handling the redex at run time. We feel, this is unnecessary because the argument for β-reduction (local body E) is available at compile time itself, and hence there is no use letting reduction wait till run time nor is the preparation necessary for it. It is proposed that all occurrences of a local variable in the Main be replaced by pointers to local body, and all local bodies (there could be more than one local definition) be kept in a list named as Context-list. This, while achieving the effect of β-reduction, preserves the sharing of sub-expressions as well (one common pointer for all occurrences of local variable).

A Context-list (henceforth C-list) in our notation is written as

$$[x_1{:}E_1,\ \ldots,\ x_n{:}E_n]$$

where $x_1, \ldots, x_n$ are identifiers for the local bodies $E_1, \ldots, E_n$ respectively. The C-list and the Main, when put together, make the definition of f appear as

$$f\ v = F :: [x_1:E_1,\ \ldots,\ x_n:E_n] \quad \ldots(6.3)$$

where '::' separates Main from C-list. It may be seen that due to the presence of common pointers for local variables and the C-list, the Main F and therefore the SDS term also, acquire a graph structure (without them they had a tree structure).

Introduction of pointers and C-list has added the following two requirements to the abstraction scheme:

(1) conversion of occurrences of local variables in Main into pointers instead of holes.

(2) abstraction of definition parameters from Main and all local bodies.

The next section gives the details of additions required to the abstraction scheme for accommodating these requirements.

6.2 ADDITIONAL ABSTRACTION SCHEMES

In the last chapter, A-operators were introduced for pattern-abstraction. These operators leave behind a trail, in the form of directors, as they move inside to operate on the internal components of an expression. Ultimately, when the pattern itself or one of its constituents is encountered, it is replaced by an appropriate director and a hole. Attaching directors at the nodes as the expression structure is unfolded is a characteristic feature of these operators. In order to fulfil the first requirement for accommodating local definitions (section 6.1), a B-scheme is being introduced in which an abstraction operator is

completely relieved of the burden of creating the SDS term structure. It moves inside the expression simply to look for an occurrence of the variable under abstraction which, if encountered, is replaced by a pointer rather than a hole. No directors are attached by this type of operator. The local variables are abstracted out using B-scheme only and hence Eq. 6.2 gets modified as

$$trans(\text{f v} = (\lambda x.F)\ E) = A_v[B_x[F] :: E]$$

The following abstraction rules for an operator B_x bring out the methodology of this scheme:

(B1) $B_x[A]$ = ^x, atom A is a variable same as x

= A, atom A is a variable other than x, a constant or a pointer.

(B2) $B_x[E\ F]$ = $B_x[E]\ B_x[F]$

(B3) $B_x[c\ E_1\ \dots\ E_r]$ = $c\ B_x[E_1]\ \dots\ B_x[E_r]$

The only change that B-abstraction can bring to an expression is to replace local variables by pointers (the SDS term structure is created by A-abstraction).

For the second requirement (section 6.1), some more rules are to be added to A-scheme so as to be able to deal with pointers generated by B-operators. These rules are in addition to the rules A1 to A12, given in section 5.1.2.4 for pattern-abstraction. The additional rules cover the cases of (i) an atomic expression being a pointer, (ii) an application having pointer(s) as its component(s) and (iii) a constructor expression having pointer(s) as its constituent(s). With reference to the structure of an

expression arrived at in Eq. 6.3, the rules for abstracting a variable v are

(A13) $A_v[\hat{}x_i :: [x_1:E_1, \ldots, x_n:E_n]]$

$= \hat{}x_i :: [x_1:E_1, \ldots, x_i:A_v[E_i], \ldots, x_n:E_n]$,
v occurs in E_i, i = 1 to n

$= \hat{}x_i :: [x_1:E_1,\ldots,x_i:A'[E_i],\ldots,x_n:E_n]$, otherwise

(A14) $A_v[(\hat{}x_i\ G) :: [x_1:E_1, \ldots, x_n:E_n]]$

$= (/\backslash@_2, A_v[\hat{}x_i], A_v[G]) :: [x_1:E_1, \ldots, x_n:E_n]$, v occurs both in G and E_i - the expression pointed by $\hat{}x_i$

$= (/@_2, A_v[\hat{}x_i], G) :: [x_1:E_1, \ldots, x_n:E_n]$, v occurs in E_i only

$= (\backslash@_2, \hat{}x_i, A_v[G]) :: [x_1:E_1, \ldots, x_i:A'[E_i], \ldots, x_n:E_n]$,
v occurs in G only

$= (-@_2, \hat{}x_i, G) :: [x_1:E_1, \ldots, x_i:A'[E_i], \ldots, x_n:E_n]$,
v occurs neither in E_i nor in G

(A15) $A_v[(c\ G_1 \ldots \hat{}x_j \ldots G_r) :: [x_1:E_1, \ldots, x_n:E_n]]$

$= (\#@_1, c\ G_1 \ldots\hat{}x_j \ldots G_r) :: [x_1:E_1, \ldots,x_j:A'[E_j], \ldots,$
$x_n:E_n]$, v does not occur in any G_i or E_j, (i,j=1 to r)

$= (d_r c, A_v[G_1], \ldots, A_v[\hat{}x_j], \ldots, A_v[G_r] :: [x_1:E_1, \ldots,$
$x_n:E_n]$, otherwise

The rules show the behaviour of an A-operator if a pointer exists in the Main. Rule A13 is for a Main which is simply a pointer.

Central Library University of Roorkee

In this case, the abstraction operator moves to operate on the corresponding local body in C-list provided the variable v occurs in it (first equation). If the variable is not referred by it then a A'-operator is invoked to act on the local body (second equation). An A'-operator is not an abstraction operator. It simply creates a tree structure of a expression, without assigning any directors. Its rules are

(A'1)	A'[A]	$= (@_1, A)$, A is an atom
(A'2)	$A'[G_1\ G_2]$	$= (@_2, A'[G_1], A'[G_2])$
(A'3)	$A'[c\ G_1 \ldots G_r]$	$= (@_1, c\ G_1 \ldots G_r)$

An A'-operator serves to create term structure whenever the abstraction operator A_v is denied the opportunity to operate on a local body.

Rule A14 pertains to an applicative expression. The abstraction is done in the spirit of movement-as-per-need optimisation. Whenever a pointed local body does not refer to the variable under abstraction, the services of A'-operator are called to create term structure. Similarly, in rule A15, meant for a constructor expression, the director and movement of abstraction operator is decided according to need.

The Main expression may have several occurrences of a local variable which are all converted into a common pointer through B-abstraction. In such a case, the A-scheme would create several redundant paths (indicated by directors) leading to the same local body. This would require unnecessary copying of an argument at run

time during substitution. To avoid this multiplicity of paths, the A-scheme has been further optimised. While deciding a director at an application node or a structural node, if the reference of the abstractee is in two or more than two arms through the same pointer then only a left (in case of an application) or left-most (in case of data structuring) director is assigned and the abstraction operator moves only to the corresponding arm, e.g.

$$A_v[(\hat{}x_i\ \hat{}x_i) :: [x_1:E_1, \ldots, x_n:E_n]]$$
$$= (/@_2, A_v[\hat{}x_i], \hat{}x_i) :: [x_1:E_1, \ldots, x_n:E_n]$$

A bothways director (/\) is not assigned in this situation because that would simply provide another extra path leading to the same local body. This optimisation is being named as **keep-left** optimisation.

6.3 COMPILATION PROCESS

Equipped with the above abstraction tools, we now take the actual compilation of definitions having locally defined local variables into SDS terms. The SDS terms, here, have a graph structure due to pointers and C-list. We make use of the function *trans* which, while operating on a definition, invokes the services of A or B operators as appropriate for the definition.

In general, a user defined function definition D is expressed as

$$f\ v_1\ v_2\ \ldots\ v_m = E$$

which has m parameters (here all of them are assumed to be simple

variables and not patterns). E has let and letrec blocks also. The function *trans* operates on D as

$$trans(D) = A_{v_1} \ldots A_{v_m}[trans_local(E)] \quad \ldots(6.4)$$

There are several cases here depending on the kind of local definition in E. The new function *trans_local* acts accordingly.

Case I : E is a let-expression

The expression is written as E = (let x = E_1 in F). Using the ideas of pointers and C-list, the *trans_local* calls a B-operator for abstracting out the local variable x and creates a C-list as

$$trans_local(E) = B_x[F] :: [x : trans_local(E_1)] \quad \ldots(6.5)$$

The C-list has a single element here. The recursive call of *trans_local* in Eq. 6.5 is to take care of any nesting of local definitions in E_1. If E_1 contains no further local definition then *trans_local* terminates by returning E_1 itself and may be defined for a let block as

trans_local(let x=E_1 in F)

= if E_1 has no let-block

then B_x[F] :: [x:E_1]

else B_x[F] :: [x:*trans_local*(E_1)]

A let-expression may also have multiple definitions as

E = (let x_1 = E_1

...

x_n = E_n in F)

In that case (assuming no nesting of local definitions), the *trans_local* acts as

$$trans_local(E) = B_{x_1} \dots B_{x_n}[F] :: [x_1:E_1, \dots, x_n:E_n] \quad \dots(6.6)$$

Combining Eqs. 6.4 and 6.6, the complete translation of the definition D with n local definitions is given as

$$trans(D) = A_{v_1} \dots A_{v_m}[B_{x_1} \dots B_{x_n}[F] :: [x_1 : E_1, \dots, x_n : E_n]] \quad \dots(6.7)$$

The result has multiple abstraction whose rules are same as given earlier i.e. A10 - A12 (section 5.1.2.4).

<u>Case II</u> : **E is a letrec expression**

The expression is written as (E = **letrec** x = E_1 **in** F). It is different from case I because x is bound to E_1 not only in F but also in E_1. The recursive expression gets converted into a simple let-expression using the fixed point combinator Y [11] as

$$E = (\textbf{let } x = Y\ (\lambda x.E_1)\ \textbf{in } F)$$

Having expressed the letrec-expression as a let-expression, the function *trans_local* can be defined in a manner similar to that in case I giving

$$trans_local(E) = B_x[F] :: [x:Y\ (A_x[trans_local(E_1)])] \quad \dots(6.8)$$

The recursive invocation of *trans_local* takes care of the possible nesting of local definitions in E_1.

Recursion can be managed without Y combinator also, through cyclic pointers [11]. If cyclic pointers are used in our compilation scheme, it will lead to self-referencing elements in the C-list. In addition, if the global variables v_1, ..., v_n occur in the local body E_1 then the context is incomplete till, at run time, these parameters are available. This means that at run time, the updated context should replace the incomplete one for cyclic use. This approach does not seem to offer an efficient reduction mechanism in the model proposed and therefore, has not been explored further.

Case III : **E has mutual recursion in the letrec block**

The expression is given by

$$
\begin{aligned}
E = (\textbf{letrec}\ & x_1 = E_1 \\
& \ldots \\
& x_n = E_n\ \textbf{in}\ F)
\end{aligned}
\qquad \ldots(6.9)
$$

In this expression the scope of any x_i (i = 1 to n) is all E_i and F. This set of mutually recursive simultaneous equations can be combined into a single definition which defines a n-tuple type of structured local variable [11,31]. It leads to

$$E = (\textbf{letrec}\ (x_1, \ldots, x_n) = (E_1, \ldots, E_n)\ \textbf{in}\ F)$$

which, by using Y operator, gets converted into

$$
\begin{aligned}
E = (\textbf{let}\ & (x_1, \ldots, x_n) \\
& = Y\ (\lambda(x_1, \ldots, x_n).(E_1, \ldots, E_n))\ \textbf{in}\ F)
\end{aligned}
\qquad \ldots(6.10)
$$

The above two expressions for E correspond to letrec case and let case respectively. However, none of the definitions of *trans_local*, used in these cases, applies here because both the above representations of E define a structured variable rather than a simple one. We now apply lambda-calculus semantics to Eq. 6.10 giving

$$E = (\lambda(x_1, \ldots, x_n).F)\ (Y\ (\lambda(x_1, \ldots, x_n).(E_1, \ldots, E_n)))$$
$$= (\lambda\tilde{x}.F)\ (Y\ (\lambda\tilde{x}.(E_1, \ldots, E_n))) \qquad \ldots(6.11)$$

where $\tilde{x} = (x_1, \ldots, x_n)$. It may be noted that in Eq. 6.11 the 'λ' operators are binding a structured variable. Applying $\gamma 2$ rule (section 4.2) of the list manipulative lambda calculus [33,106], Eq. 6.11 becomes

$$E = (\lambda\tilde{x}.F)\ (Y\ (\lambda\tilde{x}.E_1, \ldots, \lambda\tilde{x}.E_n))$$

To simplify the notation, let $\lambda\tilde{x}.E_i = e_i$, $i = 1$ to n. E then simplifies to

$$E = (\lambda\tilde{x}.F)\ (Y\ (e_1, e_2, \ldots, e_n)) \qquad \ldots(6.12)$$

The above form of E is a ß-redex where the bound variable is a n-tuple but the argument is an applicative expression. In the argument expression, the Y combinator is applied to a n-tuple instead of a single function. Reduction for Y, in this case, results into a n-tuple given by

$$Y\ (e_1, \ldots, e_n) \dashrightarrow (e_1\ (Y(e_1, \ldots, e_n)), \ldots, e_n\ (Y(e_1, \ldots, e_n)))$$
$$= (e_1\ (Y\ \tilde{e}), \ldots, e_n\ (Y\ \tilde{e})) \qquad \ldots(6.13)$$

where $\tilde{e} = (e_1, e_2, \ldots, e_n)$. The reduction rule in Eq. 6.13 is being called as **Mutual-Y rule**, and it is an **extension** of the simple Y rule Y H --> H (Y H). Here an interesting analogy with plant life can be traced. Sowing a seed produces a plant which contains the seed also for future reproduction. It corresponds to simple Y rule. The analogy can be **extended** to mutual-Y through a hypothetical concept of a 'group seed'. A group seed is something capable of producing all the plants of the group. Sowing a group seed produces a plant of each variety and each plant carries the full group seed also for future reproduction. Thus any one plant can produce any other plant in that group. This corresponds to mutual recursion rule as is evident from Eq. 6.13 where a reduction gives rise to n 'plants', each carrying the 'group seed'.

Using Eq. 6.13 in Eq. 6.12, the expression E can be written as

$$E = (\lambda\tilde{x}.F)\ (e_1\ (Y\ \tilde{e}),\ \ldots,\ e_n\ (Y\ \tilde{e}))$$
$$= (\lambda\tilde{x}.F)\ (X_1,\ \ldots,\ X_n)$$

where $X_i = e_i\ (Y\ \tilde{e})$, i = 1 to n. Now E contains a β-redex where the bound variable and the argument both are n-tuples. However, it may be noticed from the original form (Eq. 6.9) that the body F is referring only to the individual components of the bound tuple and not to the complete structure. Hence it may be interpreted that the reduction of the β-redex requires a simple multiple substitution of the kind

$$[X_1/x_1,\ X_2/x_2,\ \ldots\ ,\ X_n/x_n]\ F$$

In view of this interpretation, the original form of E in Eq. 6.9 may alternatively be expressed as

$$E = \text{let } x_1 = X_1$$
$$\ldots$$
$$x_n = X_n \text{ in } F \qquad \ldots(6.14)$$

In Eq. 6.14, the set of mutually recursive definitions of Eq. 6.9 has finally been replaced by a set of simple let-definitions. The mutual recursion now works through $(Y\ (\lambda\tilde{x}.E_1,\ \ldots,\ \lambda\tilde{x}.E_n)$, the expression present in each X_i.

Based on the above discussion, the function *trans_local* may be defined on the lines of Eq. 6.6 of case I as

$$trans_local(E) = B_{x_1} \ldots B_{x_n}[F] :: [x_1{:}X_1,\ \ldots,\ x_n{:}X_n]$$

Each $X_i = (\lambda\tilde{x}.E_i)\ Y\ (\lambda\tilde{x}.E_1,\ \ldots,\ \lambda\tilde{x}.E_n)$ can be expressed in terms of abstraction operators as

$$(A_{\tilde{x}}[E_i])\ Y\ (A_{\tilde{x}}[E_1],\ \ldots,\ A_{\tilde{x}}[E_n]),\ i = 1 \text{ to } n$$

Hence *trans_local* is finally given as

$$trans_local(E) = B_{x_1} \ldots B_{x_n}[F] :: [x_1{:}(A_{\tilde{x}}[E_1]\ S),\ \ldots,$$
$$x_n{:}(A_{\tilde{x}}[E_n]\ S)] \qquad \ldots(6.15)$$

where $S = Y\ (A_{\tilde{x}}[E_1],\ \ldots,\ A_{\tilde{x}}[E_n])$. The quantity S represents the group seed present in each element of the C-list. The abstraction operator $A_{\tilde{x}}$ represents a pattern abstraction with respect to the tuple $(x_1,\ \ldots,\ x_n)$. This abstraction, governed by rules A7-A9 (chapter 5), will replace an occurrence of a x_i (i= 1 to n) by a

list director and a hole through which the relevant element of the tuple S can be selected for use. Eq. 6.15, combined with Eq. 6.4, gives the complete SDS term for a definition having mutual recursion in its local definitions.

The above description of the various cases completes the compilation of local definitions. An example here illustrates some aspects of the process. It corresponds to the definition set of a program for summing first m integers. It has a letrec block in one of the definitions.

<u>Example</u>

```
$Count count m n = IF (GT n m) NIL (Cons n (count (+ 1 n)))
$Sum ns = IF (EQ NIL ns) 0 (+ HD ns) ($Sum (TL ns)))
$Sumints m = (letrec count = $Count count m in $Sum (count 1))
```

The '$' sign, before an identifier, indicates a supercombinator. First two definitions have no local definitions hence A-scheme is used. The detailed steps of abstraction for them are being skipped.

$$\text{(i)}\quad \textit{trans}(\$\text{Count} \ldots) = A_{\text{count}}\ A_{\text{m}}\ A_{\text{n}}[\text{IF (GT n m)} \ldots]$$

$$= (\backslash\ /\ /\backslash @_2,\ (//@_2,\ (\backslash\backslash @_2,\ \text{IF},\ (\backslash\ /@_2,\ (\backslash @_2,\ \text{GT},\ !\blacksquare)),\ \text{NIL}),$$
$$(\backslash\ /\backslash :,\ !\blacksquare,\ (/\ \backslash @_2,\ !\blacksquare,\ (\backslash @_2,\ +1,\ !\blacksquare))))))$$

$$\text{(ii)}\quad \textit{trans}(\$\text{Sum ns} \ldots) = A_{\text{ns}}[\text{IF (EQ NIL ns)} \ldots]$$

$$= (/\backslash @_2,\ (/@_2,\ (\backslash @_2,\ \text{IF},\ (\backslash @_2,\ \text{EQ NIL},\ !\blacksquare)),\ 0),\ (/\backslash @_2,$$
$$(\backslash @_2,\ +,\ (\backslash @_2,\ \text{HD},\ !\blacksquare)),\ (\backslash @_2,\ \$\text{Sum},\ (\backslash @_2,\ \text{TL},\ !\blacksquare)))))$$

The third definition has a letrec-expression. We go through complete steps of its translation.

```
(iii) trans($Sumints m ...) = A_m[trans_local(letrec count ...)]
                                                        ;by Eq. 6.4

      = A_m[B_count[$Sum (count 1)] ::
            [count:Y (A_count[$Count count m])]]          ;by Eq. 6.8

      = A_m[$Sum (^count 1)::[count:Y (A_count[$Count count m])]]
                                                   ;by B-scheme rules

      = A_m[$Sum (^count 1) ::
           [count:Y (/@_2, A_count[$Count count], m)]]   ;by rule A5

      = A_m[$Sum (^count 1) ::
           [count:Y (/@_2, (\@_2, $Count, !■), m)]]    ;by A5 and A1

      = (\@_2, $Sum, A_m[^count 1]) ::
           [count:Y (/@_2, (\@_2, $Count, !■), m)]      ;by A5 for A_m

      = (\@_2, $Sum, (/@_2, A_m[^count], 1)) ::
           [count:Y (/@_2, (\@_2, $Count, !■), m)]    ;by A14

      = (\@_2, $Sum, (/@_2, ^count, 1)) ::
           [count:A_m[Y (/@_2, (\@_2, $Count, !■), m)]]  ;by A13

      = (\@_2, $Sum, (/@_2, ^count, 1)) ::
           [count:(\@_2, Y, A_m[(/@_2, (\@_2, $Count, !■), m)])] ;by A5

      = (\@_2, $Sum, (/@_2, ^count, 1)) ::
           [count:(\@_2, Y, (\ /@_2, (\@_2, $Count, !■), A_m[m]))]
                                                             ;by A11

      = (\@_2, $Sum, (/@_2, ^count, 1)) ::
           [count:(\@_2, Y, (\ /@_2, (\@_2, $Count, !■), !■))]  ;by A2
```

The graph corresponding to this SDS term is shown in Fig. 6.1. In the appendix, the example has been taken up further to illustrate the use of this SDS term in computation through task reduction discussed in next chapter.

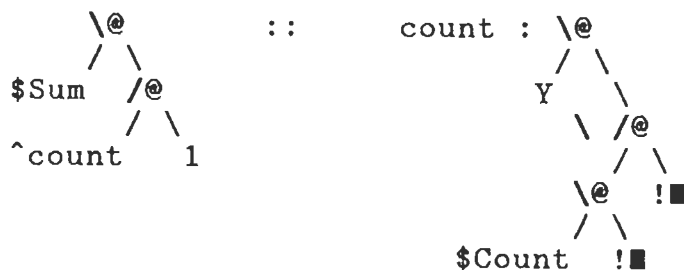


Figure 6.1 Annotated tree corresponding to the SDS term for the supercombinator $Sumints.

6.4 D-CODE STRUCTURE

The compilation discussed in this chapter and the previous one is basically concerned with abstracting out the formal parameters (patterns or variables) from a definition body, and thus yielding a SDS term. In addition, in the previous chapter the SDS terms corresponding to various clauses of a definition were paired with pattern-type sequences derived from the formal parameters of each clause. The list of pairs, so generated for a pattern-matching definition, has been termed as the SDS-field of the D-code structure. In the case of a single definition, the field simply has a SDS term.

Another field, termed as header, is being added to complete the compiled code structure. The header field contains following sub-fields :

Identifier : a name for the function defined.

m : an integer giving the number of arguments needed by the function i.e. its arity.

n : an integer giving the number of clauses in the definition.

UL : a list of integers giving the parameter numbers which the definition body discards (unwanted list).

SL : a list of integers giving the parameter numbers which have multiple occurrences in the definition body (sharing list).

CN : an integer indicating that the function body has a computable sub-expression if a minimum of CN arguments are available (computability number).

The identifier gives a name to the D-code of each function by which it can be referred. Integer m is helpful at the time of reduction in determining whether a function application is saturated or not. The number n is useful in deciding the requirement for pattern-matching. If n > 1 then matching is required and the SDS field correspondingly contains a list of pairs, otherwise the field has only a SDS term and no matching is required.

UL is a list of integers indicating the unwanted argument numbers. (if a director string has m directors then the director number k from left corresponds to the argument number k, k = 1 to m). The UL list is utilised at run time in deciding to discard an unwanted argument in its raw form. It saves computation time and provides some safety in cases where the unwanted argument is non-terminating type. The list is generated from the director string at the root node of the SDS term. A discardable argument has a discarding director (- or #) at the root node itself. Such argument numbers are included in the list. If there is no discardable argument then the list is empty. A pattern-matching

definition has n such lists corresponding to the n clauses.

SL is again a list of integers listing the argument numbers which may be shared in the definition body due to multiple occurrences of the variables corresponding to them. The reduction strategy, discussed in chapter 7, uses this information to preserve laziness in computation of shared arguments. The list is generated by following the directors for each argument in the SDS term. Whenever a bothways (/\) director (meaning sharing) is found, the argument number is included in the list. A pattern-matching definition has n lists, one for each clause.

The last sub-field CN needs maximum description. The integer contained in this field gives the minimum number of arguments required for having a computable sub-expression in the function body. The information is used during reduction to decide whether to go ahead with the reduction of a partially applied function or to wait. It helps in avoiding repeated computations for the same sub-expression. The matter is discussed in full detail in section 7.5. At present, we are concerned with how to generate CN from a given function definition. The compile time operation of finding this number for a given SDS term is being called computability analysis.

<u>Computability analysis</u>

Let t be a SDS term for the body of a definition under analysis, and m be the number of parameters. Let the symbol $t^{(-k)}$ represent the term t with (i) directors for first k arguments removed, (ii) holes corresponding to the removed directors, and all constants

replaced by C (meaning computable), and (iii) the remaining holes and data constructor nodes replaced by XC (meaning not computable). The formation of $t^{(-k)}$ is with the speculative idea that the available arguments are computable. The remaining holes are naturally not computable. Thus $t^{(-k)}$ has a binary tree structure with leaves as C, XC or a pointer.

Using the above notation, Fig. 6.2 gives a program in functional style for finding the computability number for a term t. If m = 0 then there is no need to find CN, otherwise *compute* is called to start checking t with zero directors removed. Starting with zero takes care of any constant sub-expression such as (square 4). The function *compute* checks the computability of $t^{(-k)}$. If it is computable then k is returned as CN, otherwise the computability of $t^{(-(k+1))}$ is checked. The process stops when k becomes equal to m because there is no point in going further. Computability of a node or a leaf is tested through the function *tree_computable* and *leaf_computable* respectively. As long as there are directors at a node, it cannot be computable hence the test passes on to left and right sub-trees or the C-list terms. A node free from directors can be computable if both its sub-trees are computable. Function *leaf_computable* returns 'True' only for a leaf having C. For a node input, it calls back the function *tree_computable.* The *leaf_computable* cannot get a node with directors as input.

```
find_CN (t, m) = 0,   m = 0
               = compute (t, m, 0)

compute (t, m, k) = k,  k = m
                  = if (tree_computable (t^(-k))) then k
                    else compute (t, m, k+1)

tree_computable (leaf)          = False

tree_computable ((D2@2, t1, t2) :: [x1:E1, ..., xn:En])
                                = tree_computable (t1) OR
                                  tree_computable (t2) OR
                                  tree_computable (E1) OR
                                  ...
                                  tree_computable (En)

tree_computable (@2, t1, t2)    = if leaf_computable (t1)
                                  then leaf_computable (t2)
                                  else tree_computable (t2)

leaf_computable (^xi)           = False

leaf_computable (C)             = True

leaf_computable (XC)            = False

leaf_computable (■)             = False

leaf_computable (@2, t1, t2)    = tree_computable (@2, t1, t2)
```

Figure 6.2 Program for finding CN

6.5 CONCLUDING REMARKS

Function definitions often have let and letrec blocks where some variables are defined locally. The feature allows sharing of sub-expressions which otherwise would require separate computation for each occurrence in the definition body. The abstraction scheme, discussed in chapter 5, is not suitable for abstracting out local

variables as it leads to an inefficient code. The code has directors and holes for reducing a β-redex at run time which could easily be resolved at compile time itself.

To overcome this, the abstraction scheme has been suitably appended by incorporating pointers and introducing the concept of context-list (C-list) in the SDS term structure. This gives the terms a graph structure where the SDS term of a local body, forming an element of C-list, is connected to the main body (at several places) through a common pointer. The use of pointers and C-list retains sharing besides avoiding substitution of local bodies at run time.

The route from supercombinator definitions to SDS terms consists of an interpretation of different types of local definitions into lambda calculus followed by an encoding of their meaning into the modified SDS term structure. An elegant solution to mutual recursion has been obtained where a set of mutually recursive definitions is converted into a set of independent let-definitions.

Giving final touches to the compilation, a D-code structure, consisting of a SDS field and a header field is introduced for each function definition. The SDS terms form part of the SDS field whose other part is the sequences of pattern-types meant for pattern matching work. The header field contains some miscellaneous information for house-keeping jobs. In particular, it includes a computability number (CN) which gives the minimum number of arguments that will generate a computable sub-expression

in a body. An algorithm for generating this number from a given SDS term has been devised. CN is used in the reduction process (next chapter) for avoiding repeated computations whenever a partially applied (unsaturated) supercombinator is shared.

The next chapter deals with the final aspect of the computation model i.e. reduction. It develops a supercombinator reduction mechanism where the SDS terms are used as a means of template instantiation in a simple and mechanical way.

CHAPTER 7

MULTIPROCESSOR REDUCTION

The computation model, proposed in this work, starts with an assumption that a program is available in supercombinator form obtained through lambda-lifting transformation of the high level program. Such a program has the following main features:

- It is made up of a set S of supercombinator definitions and an expression E for evaluation. Each definition expresses a function of arity m ($m \geq 0$) where the body has no free variables, but may have other supercombinators and built-in functions. Functions may be defined through pattern-matching also.
- Arbitrary let/letrec blocks, defining some local variables, may be embedded in a definition body. Prior lambda-lifting ensures that function definitions occur only at the top level, and not within let/letrec blocks.
- Data objects are built through constructor functions.

Through the compilation scheme discussed in previous two chapters, each function definition in the set S gets compiled into a D-code which, besides a header field, consists of either a SDS term for the definition body, if a simple function is defined, or a list of pairs of a pattern-type sequence and a SDS term, one for each clause, if a pattern-matching function is defined.

The current chapter presents the development of a reduction scheme for evaluating the program expression E in the light of definition set S while making use of the D-code. Before reduction, the

expression is organised (a compile time operation) into a graph of variable sized supercombinator redexes of the kind (F Q_1 Q_2 ... Q_m) where F is a function of arity m. A redex of this kind, named as task, applies the function to all its arguments under one computation process and thus constitutes the smallest indivisible unit of work. It is similar to current context stack in G-machine [88,89] or a packet in Flagship [95]. Reduction involves template instantiation of F (if it is user defined) using its SDS term. However, in case of a pattern-matching function, matching precedes instantiation. F could be a primitive operator also and then the reduction follows the operator rule. The complete task-graph reduces in a message passing multiprocessor environment making use of the opportunities of parallelism, wherever possible.

The chapter is organised as follows: Section 1 is preparatory to the reduction process. It gives the reduction rules for SDS terms applied to some arguments. Section 2 describes the task structure. Section 3 discusses an algorithm for organising a program expression into a task-graph. Section 4 deals with the actual task reduction mechanism including descriptions of messages and their handling. In section 5, some modifications to the simple reduction strategy so as to accommodate the features of laziness, recursion etc., are discussed. Finally section 6 gives a summary and concluding remarks on the reduction scheme.

7.1 SDS TERM REDUCTION

The SDS term structure resulting from the compilation scheme, given in previous chapters, is summarised in the following

definition of a term:

Definition 7.1 : SDS term

term	::= atom | (r+1)-tuple | (r+1)-tuple :: C-list
atom	::= value | hole | pointer
value	::= constant | operator | function_identifier
(r+1)-tuple	::= ($d_r D_r$nt, term_1, ..., term_r)
C-list	::= Nil | [P_1 : term, C-list]

From the above definition, the general form of a (r+1)-tuple type term, representing a r-ary node, having a C-list is written as

$$((d_r D_r nt,\ t_1,\ \ldots,\ t_r)::[P_1:t_{c1},\ \ldots,\ P_n:t_{cn}]),\ r \geq 1,\ n \geq 0$$

where t_1, t_2 ... stand for arbitrary SDS terms; P_1, P_2 ... are identifier names for corresponding terms t_{c1}, t_{c2} ... in the C-list; d_r is a r-ary director; D_r is a string of r-ary directors; and 'nt' is a symbol for node-type. It may be recalled that a SDS term represents the body of a user defined function in the form of a graph with nodes having director string annotations for guiding the substitution of actual parameters at run time. The reduction rules for these terms are meant for dealing with an application of a term t to an argument Q. Such a situation arises when the function F associated with a redex (F Q_1 Q_2 ... Q_m) is a user defined one. The function identifier F, in that case, is replaced by its SDS term, and argument substitution proceeds according to the term reduction rules. These rules, by induction over the syntax of term (Definition 7.1), are

(R1) $\blacksquare\ Q \quad \text{-->} \quad Q$

(R2) $\hat{}P_i\ Q \quad \text{-->} \quad t_{ci}\ Q, \quad 1 \le i \le n$

(R3) $(!@_1,\ \blacksquare)Q \quad \text{-->} \quad (@_1,\ \blacksquare\ Q)$

(R4) $(!D_1@_1,\ t_1)Q \quad \text{-->} \quad (D_1@_1,\ t_1Q)$

(R5) $(\#D_1@_1,\ t_1)Q \quad \text{-->} \quad (D_1@_1,\ t_1)$

(R6) $(/\backslash D_2@_2,\ t_1,\ t_2)Q \quad \text{-->} \quad (D_2@_2,\ t_1Q,\ t_2Q)$

(R7) $(/D_2@_2,\ t_1,\ t_2)Q \quad \text{-->} \quad (D_2@_2,\ t_1Q,\ t_2)$

(R8) $(\backslash D_2@_2,\ t_1,\ t_2)Q \quad \text{-->} \quad (D_2@_2,\ t_1,\ t_2Q)$

(R9) $(-D_2@_2,\ t_1,\ t_2)Q \quad \text{-->} \quad (D_2@_2,\ t_1,\ t_2)$

(R10) $(d_rD_rnt,\ t_1,\ \ldots,\ t_r)Q \quad \text{-->} \quad (D_rnt,\ t_1Q,\ \ldots,\ t_rQ)$

A general characteristics of the above reduction rules is to transport the argument to a sub-term depending on the leading director. Rule R1 corresponds to the terminal situation of an argument reaching a hole. Successive applications of various other rules may ultimately terminate in the application of this rule. Rule R2 refers to a pointer being applied to an argument. It says that the argument should be passed to the corresponding term in the C-list. There is no rule for a value type term because it does not form part of term-reduction. It is dealt at task reduction level. Moreover, the abstraction rules (A1-A15), for producing SDS terms, do not allow this kind of situation to occur. Rules R3 - R5 are for pair terms, and R6 - R9 for triple terms. In rule R10, which is for a general (r+1)-tuple term, the argument Q is actually passed to the sub-terms indicated by d_r.

The above rules have been written with an atomic director in the leading position. The same rules apply if the leading director is a pattern director. However, if it is a list director then a

breaking of the argument structure is suggested. Let the structure of Q be given by

$$Q = c\ q_1\ q_2\ \cdots\ q_k$$

The reduction rule then, with a list director in the front, is

$$\text{(R11)} \qquad (\{\#^{j-1},\ !,\ \#^{k-j}\}D_1@_1,\ t_1)Q \text{ --> } (D_1@_1,\ t_1q_j), \quad 1 \le j \le k$$

A list director is assigned, during abstraction, only when a leaf is encountered in the expression tree. Hence the director occurs in pair terms only.

In the reduction of a task, the SDS term of a function definition acts as a template, and the term reduction rules govern the instantiation process.

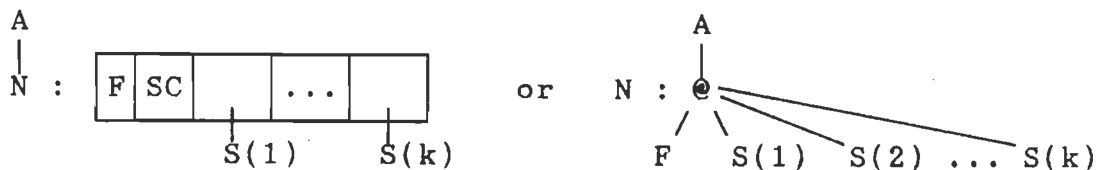
7.2 TASK STRUCTURE

An important design parameter in a parallel reduction machine is grain size of a computation unit. The whole program can be divided into such units and distributed over the parallel hardware, hoping to achieve better throughput due to concurrent action. But a necessary evil associated with all such arrangements is the 'administrative' overheads (e.g. communication between processors). Too fine a grain size may provide better utilisation of parallelism, but a major part of the advantage gained could be offset by increased overheads. On the other hand, too coarse a grain size may lead to loss in parallelism, but the communication costs are less (a single processor machine executing

an imperative program is an extreme of coarse grain - the whole program is a single indivisible grain, hence no communication but no parallelism too). Although the issue of optimal grain size is difficult to resolve, but the idea of bigger grain size has found more favor [15,83] because the ratio of inter-processor communication time to CPU instruction time is generally quite high.

In our model, a supercombinator definition body is compiled into a SDS term which may look like going from the coarse grain size of a supercombinator to the fine grain of director string model. However, this is true only if the reduction is done in an 'incremental manner' as in, e.g. CTDNet [58,59]. The proposed reduction model treats a supercombinator/operator applied to *all its arguments* as a single indivisible unit of computation named as **task**. In other words, the task reduction represents a single large step instantiation of a supercombinator body rather than a collection of small steps with communication intervals in between them. G-machine [88,89] and Flagship [95] architectures have similar grain sizes.

For task reduction, the program expression, under evaluation, is organised into a task-graph where the nodes are tasks. Each node maintains graph links in the form of ancestors and successors. Tasks are of variable size because different functions require different number of arguments. The structure of a task in terms of its various fields is represented as follows:



where

N = Name field

A = Ancestor field

F = Function field

SC = Successor count field

S(1), ..., S(k) = k Successor fields, $k \geq 0$

The N-field provides an identification for a task. Besides self address, the field contains two more sub-fields called type and count. Task types are defined according to the number and nature of arguments available for a function. In a program expression (f e_1 e_2 ... e_k), let the arity of the function f be m. Depending on the values of m and k, following three task types are possible from this expression (the process of converting a program expression into a task-graph is termed as <u>organisation</u> and is discussed in next section):

(1) **Complete** : $m = k$, i.e. the function requirement matches the availability.

(2) **Partial** : $m > k$, i.e. the function requires more than the available arguments. The situation corresponds to partially applied functions [84].

(3) **Dummy** : $m < k$, i.e. the function requirement is less than the availability. Dummy task holds the surplus arguments (k-m) in its successor fields and the complete task (organised from m arguments) in its F-field.

The 'Complete' type is further divided into two sub-types called

Executable and **Waiting** depending on the nature of arguments. In an executable task, each successor is a data value/pointer but not a task whereas a waiting task has some successors as tasks.

The count sub-field in N-field is just an integer which has different meaning for different type of tasks. It indicates the number of successors in task form if the task is W type; the shortage of arguments if task is P type; and the number of surplus arguments if task is D type.

The type and count sub-fields have been purposely included in the N-field. A task refers to its neighbours by their N-fields. The presence of type and count information in this field provides a task all the necessary knowledge about its neighbours which otherwise has to be obtained through communication.

The ancestor field is a linkage field. It holds N-field of the ancestor task and a number 's' indicating that the task is sth successor of its ancestor. If a task is shared then its A-field contains a list of A-fields indicating all those who share it. A root task has A-field as nul.

The F-field contains the function identifier which the task is supposed to apply to the arguments available in the successor fields. It could have a built-in operator, a user defined function or the N-field of another task. Sometimes, F-field may house a pointer to an unorganised expression graph also whose implication is discussed later.

The successor fields correspond to the arguments. The fields are

ordered from left to right. A successor could be a task or some data item, a pointer to a data node or an unorganised graph. For a task type successor, the field carries the name field of the pointed task. For other data type successors, it becomes necessary to indicate the type i.e. whether it is a constant or some operator or some identifier etc.. Hence, the field holds a tagged value consisting of sub-fields 'tag' and 'qty'.

The SC field simply gives the number of successors in a task. It may be noted that in a dummy task, the count in N-field and this SC field will have the same value because a dummy holds the surplus arguments as successors.

The above description is summarised in the following definition of task fields:

Definition 7.2 : Task fields

N ::= (Identifier, type, count)

type ::= E | W | P | D

count ::= positive integer

A ::= nul | $(N_A ; s)$ | $(N_{A1};s_1, N_{A2};s_2, \ldots, N_{Ap};s_p)$

$s, s_1, \ldots, s_p$ ::= +ve integers indicating ancestor's successor no.

F ::= <u>op</u>operator | <u>$</u>function_identifier | N | <u>ug</u>^unorg_graph

SC ::= positive integer giving the no. of successors

S(I) ::= N | tagged_value, I = 1 to k

tagged_value ::= (tag, qty)

tag ::= <u>val</u> | <u>op</u> | <u>$</u> | <u>ug</u> | <u>ds</u>

qty ::= constant | operator | function_identifier | ^unorg_graph | ^data

---

A function_identifier refers to a user defined function and is identified by a $ prefix. The pointer types ds and ug refer to a data structure and an unorganised graph respectively. As an example, the expression (Twice Twice Suc 0), when organised, will give rise to two tasks shown in Fig. 7.1. Here X and Y are task identifiers. Task X is dummy and task Y is executable. F-field of X points to Y task. In the A-field of Y task, s = 0 means that

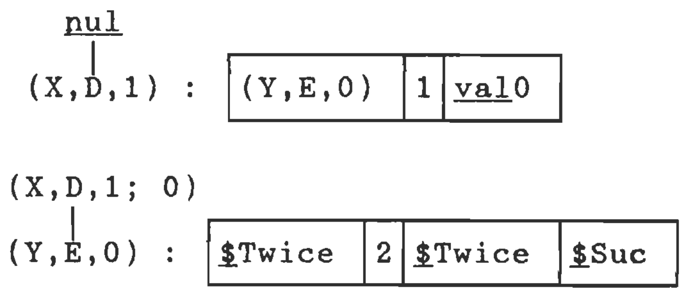


Figure 7.1 Task-graph for the expression (Twice Twice Suc 0)

it is 0th successor of its ancestor (the F-field is treated as 0th successor for link purposes). The Y task knows, through its A-field, that its ancestor is a dummy task having one argument. Similarly, the X task knows, from its F-field, that its 0th successor is an executable task. This has been made possible by including type and count sub-fields in the N-field. The above task-graph is obtained by organising the expression (Twice Twice Suc 0) and the next section gives the details of this organisation process.

7.3 ORGANISATION OF PROGRAM EXPRESSION

The process of organisation is a compile time operation for converting a program expression into a task-graph. It is used at run time also, for organising the intermediate results produced by task reduction, but in a slightly different form.

An expression in a high level functional language builds up in size mainly through left-associative binary application and through data structuring using constructor functions. Its atoms consist of constants, built-in operators, and function identifiers. Besides these, sharing of sub-expressions may also exist. An expression may thus be viewed as an expression graph defined as follows:

Definition 7.3 : exp_graph

```
------------------------------------------------------------------
exp_graph ::= node | leaf

node      ::= @ (exp_graph, exp_graph) | c (exp_graph_1, ...,
              exp_graph_r) | identifier : node

leaf      ::= constant | operator | function_identifier |
              ^identifier
------------------------------------------------------------------
```

The productions for node correspond, in order, to binary application, data construction through a constructor function c of arity r, and identified sub-expression (to show sharing). The productions for leaf include constants, built-in operators, function names, and pointers used for shared nodes. As an example, the graph of the expression (let x = SQ 5 in + x x) is written in textual form as (@ (@ (+, (x : @ (SQ, 5))), ^x)). In graphical form, it is shown in Fig. 7.2.

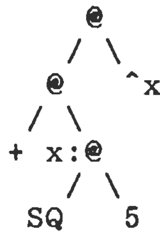


Figure 7.2 Graphical representation of (let x = SQ 5 in + x x)

The above form of expression graph is assumed as input to the organisation module. In the task model, organisation serves two purposes. The first is to frame proper tasks from the given expression graph i.e. to group a function of arity m along with m arguments (if available) under one computation process. The second purpose is to act as a pre-condition for any reductions to take place. An unorganised piece of graph is 'dead' i.e. it cannot initiate any action and hence cannot reduce. Organisation puts 'life' into it. The reduction mechanism (discussed in next section) makes extensive use of this idea in bringing selective laziness to an otherwise applicative strategy.

A conceptual picture of the organisation process is given by Def. 7.4 for a function *Org*. Here e stands for an expression, f for a function of arity m, C for a constant value, d for a data structure node, and w for a task (work) resulting from the application of *Org* to an expression.

Definition 7.4 : ***Org***

Org(C) = tag C ;tag depends on the type of C.

Org(d) = <u>ds</u>^d ;data constructor nodes are simply replaced by a pointer without organising further.

Org(^x) = ^*Org*(x:e_1) ;^x points to the expression e_1, and hence organisation results in a pointer to the
= ^*Org*(e_1) organised form of e_1.

Org (f e_1 e_2 ... e_k)

= [*Org*(f) (*Org*(e_1), ..., *Org*(e_k)],

= [w_0 (w_1, ..., w_k)], m = k ;Complete task

= [w_0 (w_1, ..., w_k, ...], m > k ;Partial task

= [[w_0 (w_1, ..., w_m)] (e_{m+1}, ..., e_k)], m < k ;Dummy task

Org(IF e_1 e_2 e_3)

= [*Org*(IF) (*Org*(e_1), <u>ug</u>^e_2, <u>ug</u>^e_3)] ;then and else arms are not organised

= [<u>op</u>IF (w_1, <u>ug</u>^e_2, <u>ug</u>^e_3)]

Org (f e_1 ...e_i ... e_m)

= [w_0 (w_1, ..., K_i, ..., w_m)] ;ith argument is unwanted.

The result of organising an expression (f e_1 e_2 ... e_k) is a task which, in the notation introduced, is written within square brackets. It consists of a function-task w_0 applied to a list of argument tasks w_1 ... w_k. An unfinished list of argument tasks (indicated by a list without a closing bracket) represents a partial task. As an example, [<u>op</u>+ (<u>val</u>5, ...] is a partial task because the '+' operator does not have a complete list of arguments.

A general characteristic of *Org* is to avoid organisation of an argument into a task if its need is doubtful. The feature imparts some safety to the reduction because an unorganised expression remains in a dormant state. The most obvious use of this feature is in the organisation of a conditional expression where initially only the condition part is organised. The other two parts are left unorganised. Later when need is known, through the result of condition, the relevant part is organised. On the same lines, whenever m < k, only the required m expressions are organised into a complete task. The dummy task, created for this situation, holds the rest k-m arguments in unorganised form because it does not know at present whether these expressions would be needed or not. Similarly, while organising the arguments of a function f, the organiser looks for its need in the definition of f (indicated in the sub-field UL of the header of D-code) and does not organise, if unwanted, thus saving the computation of an unneeded argument. In the last case of *Org* definition, ith argument is assumed as unwanted, hence a constant K_i (not meaning anything) is substituted for it and e_i is rejected.

Let us take an example of organising the expression (Twice SQ (+ (* 3 5) (* 5 7))) into tasks. It is given that the arity of Twice is 2. The organisation will proceed through following steps:

Org(Twice SQ (+ (* 3 5) (* 5 7))))

= [(*Org*(Twice)) (*Org*(SQ), *Org*(+ (* 3 5) (* 5 7)))]

= [$Twice (<u>op</u>SQ, [(*Org*(+)) (*Org*(* 3 5), *Org*(* 5 7))])]

= [$Twice (opSQ, [(*Org*(+)) ([*Org*(*) (*Org*(3), *Org*(5))],
[*Org*(*) (*Org*(5), *Org*(7))])])]

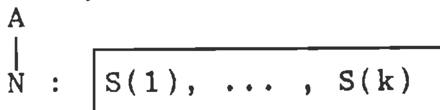
= [$Twice (opSQ, [op+ ([op* (val3, val5)], [op* (val5, val7)])])]

7.3.1 Organisation Algorithm

A formal description of the organisation algorithm is being given now. The input to this algorithm is an exp_graph defined in Def. 7.3. For convenience of reference, a structure for nodes and leaves in exp_graph is being assumed as given below:

Definition 7.5 : Node and leaf structure

Node Structure



where

N ::= (identifier, type, ref_cnt)

type ::= @ | c

ref_cnt ::= positive integer

S(1),...,S(k) ::= N | Leaf

A ::= nul | N

Leaf Structure

where

tag ::= val | op | $

qty ::= constant | operator | function_identifier

Here the ref_cnt of a node indicates the number of nodes referring

to it. Shared nodes have a ref_cnt greater than 1. The information is used by organisation module to avoid repeated organisations of a shared sub-graph. Once a shared sub-graph is organised, the ref_cnt of its root node is set to zero to indicate that the graph is already organised. Taking the above structure for nodes and leaves in an exp_graph as input, the organisation process delivers a task-graph which is defined as follows:

Definition 7.6 : task-graph

---

task-graph ::= task | tagged_value | unorg_graph

unorg_graph ::= exp_graph

The structures of task and tagged_value have been given earlier in Def. 7.2. Based on these input and output structures, the organisation algorithm is specified formally in a Pascal-like notation in Fig. 7.3. Here the main module *Org* calls *Org_exp* and then *Mark_type* to assign task-types in the task-graph created by *Org_exp*. The *Org_exp*, depending on the form of input exp_graph, calls *Org_leaf*, *Org_ds* or *Org_node*. These modules organise a leaf, a sub-graph at a data structure node and an application node respectively. Some of the procedures such as *get_root_task*, *get_count* etc., have been taken as primitive ones and have not been specified in detail.

```
program Org;

begin

  org_exp (exp_graph, task-graph);

  mark_type (task-graph);

  task:= get_root_task (task-graph);

  task.A:= nul    {set A-field of root task as nul}

end.

procedure org_exp (exp_graph, task-graph);

begin

  case exp_graph of

    leaf : org_leaf (leaf, tagged_value);

    node : if (node.N.type = 'c') then org_ds (node, tagged_value)

           else if (node.N.ref_cnt = 0)       {node is already
                                                           organised}
                then task-graph:= get_task (node.S(1))

                else begin

                       E_task:= new(task);  {create a new task for
                                                 the expression}
                       org_node (node, E_task)

                     end

  end {case}

end;

procedure org_leaf (leaf, tagged_value);

begin

  tagged_value.tag:= leaf.tag;

  tagged_value.qty:= leaf.qty

end;
```

```
procedure org_ds (node, tagged_value);

begin

  name:= node.N;

  tagged_value.tag:= ds;

  tagged_value.qty:= ^name

end;


procedure org-node (graph, task);

begin

  c_node:= get_root_node (graph);  {set current node to the root
                                     of the input graph}
  nc:= 0;       {initialise node-count}

  repeat    {repeat until the end of spine is reached}

    c_node:= c_node.S(1);    {move down left on the graph}

    nc:= nc + 1;

  until ((c_node.N.type = '@') AND (c_node.N.ref_cnt = 1));

  if (c_node.N.type = 'c') OR

       ((type(c_node) = 'leaf') AND (node.tag = val))
            {type procedure checks whether it is a leaf or a node}

  then print "error";    {corresponds to a constant or data
                          structure in the operator position}

  if (type(c_node) = 'leaf') AND (c_node.tag = op OR $)
                          {an operator or function at the spine tip}
  then begin

         tip:= c_node;

         task.F:= tip;    {set the F-field of task}

         if tip = opIF     {IF operator at the tip}

         then begin

                fpc:= 1;  {formal parameter count for IF = 1}

                ul:= NIL;  {list of unwanted arguments is empty}

                make_task (ul, fpc, fpc, task, c_node);
```

```
            task.S(2).tag:= ug;            {then and else arms
                                            are attached to the
            task.S(2).qty:= ^c_node.S(2);   task in unorganised
                                            form}
            c_node:= c_node.A;  {move up}

            task.S(3).tag:= ug;

            task.S(3).qty:= ^c_node.S(2);

            task.SC:= 3;   {set successor count to 3}

            c_node:= c_node.A;

            leftovers:= nc - 3;  {extra arguments that could
                                  not be used by IF task}
            if (leftovers > 0)

            then begin

                   d_task:= new(task); {create a new task to
                                        act as dummy}

                   dummy (leftovers, task, d_task)

                 end

          end

       else begin   {operator is other than IF}

            fpc:= get_count (tip);  {get parameter count of
                                     the function on tip}

            ul:= get_unwanted_list (tip);

            make_task (ul, fpc, nc, task, c_node);

          end

     end;

if (c_node.N.type = '@') AND (c_node.N.ref_cnt > 1)

then begin      {a shared node in the operator position}

       task.N.type:= 'D';  {declare the task as dummy}

       task.N.count:= nc; {set surplus count of dummy = nc}

       task.SC:= nc;

       save:= c_node;   {save the current node}
```

```
        F_task:= new(task);

        org_node (c_node, F_task);

        c_node:= get_node (save);  {restore node}

        c_node.N.ref_cnt:= 0;       {set ref_cnt = 0 to avoid
                                     repeated organisations}

        c_node.S(1):= F_task.N;    {keep F_task name for
                                    future use}

        link (task.F, node.S(1));  {link F-field of task to
                                     F_task}
        arg:= 1;

        while (arg ≤ nc) do

        begin

          c_node:= c_node.A;

          task.S(arg).tag:= ug;

          task.S(arg).qty:= ^c_node.S(2);

          arg:= arg + 1

        end  {while ... do begin}

      end

end;
```

procedure make_task (list, pc, aa, task, node);

```
begin

  if (aa ≠ pc)   {available arguments ≠ parameter count}

  then if (aa < pc)

       then begin

              group_args (list, aa, task, node);

              task.N.type:= 'P';

              task.N.count:= pc - aa  {set shortage count}

            end

       else begin
```

```
            group_args (list, pc, task, node);

            leftovers:= aa - pc;

            d_task:= new(task);  {create a new task to act as
                                  dummy}
            dummy (leftovers, task, d_task);

          end

  else group_args (list, pc, task, node)

end;
```

procedure group_args (list, pc, task, node);

```
begin

  save:= node;

  arg:= 1;  {initialise argument number counter}

  task.SC:= pc;

  while (arg ≤ pc) do

    begin

      if (in_list (arg, list))   {argument number is in the
                                  unwanted-list}
      then begin

             mark_for_GC (node.S(2));  {mark the unwanted argument
                                        for garbage collection}
             task.S(arg):= 'UNWANTED'

           end

      else begin

             Arg_task:= new(task);

             org_exp (node.S(2), Arg_task);

             link (task.S(arg), Arg_task) {link current successor
                                           and organised argument}
           end;

      node:= node.A;   {next argument}

      arg:= arg + 1    {increment argument counter}

    end;  {while ... do begin}
```

```
   node:= get_node (save);
   if (node.N.ref_cnt > 1) then
   begin
     node.N.ref_cnt:= 0;  {to avoid repeated organisations}
     node.S(1):= Arg_task.N   {save organised task in S(1) field
                               of the shared node for future use}
   end
end;

procedure dummy (unused_args, task, d_task);
begin
  d_task.N.type:= 'D';
  d_task.N.count:= unused_args;
  d_task.SC:= unused_args;
  link (d_task.F, task);   {dummy-task holds task in its F-field}
  arg:= 1;
  while (arg ≤ unused_args) do
    begin
      d_task.S(arg).tag:= ug;   {a dummy keeps its arguments in
                                       unorganised form}
      d_task.S(arg).qty:= ^node.S(2);
      node:= node.A;   {next argument}
      arg:= arg + 1
    end {while ... do begin}
end;

procedure mark_type (task-graph); {procedure to set N.type of all
                                   tasks in a task-graph}
begin
  root:= get_root_task (task-graph);
  if (type(task-graph) = 'task') AND (root.N.type ≠ 'P')) then
    if (root.N.type ≠ 'D')  {task is other than partial or dummy}
```

```
    then begin
           I:= 1;
           k:= root.SC; {store successor count in k}
           wc:= 0;    {initialise wait-count}
           while (I ≤ k) do
             begin
               if type(root.S(I)) = 'task')
               then begin
                      wc:= wc + 1;  {increment wait-count because
                                     successor is a task}
                      mark_type (root.S(I))
                    end;
               I:= I + 1;
             end; {while ... do begin}
           if (wc = 0) then root.N.type:= 'E'
           else begin
                  root.N.type:= 'W';
                  root.N.count:= wc  {set wait-count of root task}
                end
        end
    else mark_type (root.F)  {if task is dummy then mark the task
                              contained in its F-field}
end;
```

Figure 7.3 Specification of the Organisation Algorithm

7.4 TASK REDUCTION

The compile time *Org*, specified in the last section, converts an expression under evaluation into a task-graph. Besides expression

for evaluation, a program has definitions which get converted into SDS terms through the earlier compilation. These may be called a set of definition graphs. The two graphs, put together, constitute a computation graph which is defined as

Definition 7.7 : Comp_graph

```
-----------------------------------------------------------------
comp_graph      ::= task-graph + def_graph_set
def_graph_set   ::= { } | {def_graph_1 + ... + def_graph_p}
-----------------------------------------------------------------
```

Here p is the number of definitions in the definition set of a program. Task-graph has already been defined earlier in Def. 7.6 and a def_graph is a function body in the form of a SDS term whose definition is given in Def. 7.1. The productions for def_graph_set state that the set may be empty or may have p unconnected def_graphs. The task-graph and def_graph are also unconnected graphs. The purpose of task reduction is to reduce the task-graph to normal form i.e. to a form when it has no executable tasks left. In the process, it may use a def_graph whenever a task involving application of a user defined function is encountered.

During task-graph reduction, the tasks do not live for ever. As soon as a task has reduced, it kills itself after passing on the result in the form of a message to its ancestor(s). During the tenure of its existence, it either reduces or handles messages received from other tasks. Fig. 7.4 defines this 'life style' of a task formally, in a Pascal-like notation. The various procedure names introduced here are discussed later in this section.

```
task ::=
begin
  while task ≠ null do      {while alive, keep on doing}
    begin
      if reducible
      then reduce
      else begin
             * ? M;    {receive a message M from some task}
             case M.type of
               result : handle_result_message(M);
               link : handle_link_message(M);
               else handle_arg_message(M)
             end  {case}
           end
    end  {while ... do begin}
end.
```

Figure 7.4 Formal definition of a task life.

7.4.1 Task Reducibility

Task reduction has been designed as a graph reduction activity with potential to exploit parallelism. The reduction strategy is therefore basically applicative (eager) so as to allow simultaneous action on all reducible tasks at any given time. Based on this strategy, the conditions of reducibility for a task in the model are

(1) The task is complete i.e., its successor count is equal to the arity of the function in F-field, and the F-field function is either a built-in operator or user defined.

(2) All the successors are either in tagged_value form, or partial type tasks.

These two conditions are in the spirit of data flow approach [48] that all instructions, for which input tokens are available, can be fired. The execution of the program is thus data-driven. A Complete task which has some of its successors as further tasks (other than partial type), waits till results in tagged_value/ partial-task form are received from such successors through result/link messages. Later (section 7.5), we discuss situations where the purely eager strategy, stated above, is made selectively lazy to accommodate some features which are otherwise not possible with the eager order. In that case, the treatment for partial tasks also gets modified. At present, we continue with the details of the applicative reduction mechanism.

7.4.2 Reduction Mechanism

A complete reduction cycle for a reducible task is divided into following steps:

(1) Application : The function in F-field is applied to the arguments in the successor fields.

(2) Organisation : The result of step 1 is organised into a task-graph.

(3) Communication : The organised task-graph of step 2 is communicated to the ancestor task(s).

(4) Removal : The reducing task is removed from the computation graph.

These steps are now explained further in detail.

7.4.2.1 Application

The function to be applied may be either a user defined function (identified by a $ sign) or a built-in operator. In the former case, the application is done through what we call a $-reduction and in the latter through primitive reduction.

$-reduction : A copy of SDS term of the F-field function is obtained from the def_graph_set and the arguments (successors) are substituted in it. The process follows the term reduction rules (R1-R11) discussed in section 7.1. The successors in a task are ordered in the sense that S(1) represents the first argument for substitution, S(2) the second and so on. A function of arity m has m directors in the director string at the root node of its SDS term. S(1) undertakes a tree-walk guided by a leading director at a node till it reaches the hole where it is meant. All the leading directors which have been used in the process are deleted. On the way, if a both ways director (/\) is encountered, a copy of the argument is made. This copying is not costly because the arguments are in tagged_value form representing simple data values or pointers to data structures. In case an argument is a partial task, its name-field is used for all substitutions. At the end of the journey i.e. at a hole, the director could be a list director meant for a structured argument. In that case, the appropriate component of the argument structure is extracted, and if the component is a simple data item then it is directly substituted in

the hole otherwise a pointer to the component is substituted.

After finishing with S(1), the director string at root node would consist of (m-1) directors. The leading director now corresponds to S(2), the substitution of which is taken up in a similar manner. The process is repeated for all the m successors. In the end no directors are left in the copied template and it represents an instance of the function being applied.

Primitive Reduction : In this case, the F-field contains a built-in operator. It could be a simple operator (arithmetical, logical or relational), a conditional operator, or the fixed point operator Y. For a simple operator, the application is done following the operator rule.

The handling of IF operator requires an explanation. Left to itself, an IF-task may start action on all the three arms viz. the condition arm, **then** arm and the **else** arm. Although in a parallel machine, this should be encouraged, but some conditional expressions are specifically used to avoid non-termination through a check on a parameter value, and in such cases, the parallel action may lead to non-termination. In CTDNet [58,59], it is avoided by making the **then** and **else** sub-graphs inactive (tardy) till the condition is evaluated, after which the selected graph is activated through a message communication. In the present computation model, all the three arms belong to the same process (task), hence communication is not needed. Here the key to suspension or activation of a work is organisation. When a IF-task is created, its **then** and **else** parts are left unorganised

(Def. 7.4 of *Org*). The task, handling condition part, will eventually send a result message depending on which the IF-task will activate, through organisation, either then or the else part and link it to its ancestor. In this way, the unneeded arm of a conditional is neither organised nor computed.

The reduction mechanism for the Y operator is discussed in section 7.5 while discussing modifications to the scheme.

7.4.2.2 Organisation

The result of application step is either a value or an exp-graph. In either case, the result is organised into a task-graph using an organisation algorithm similar to that of section 7.3. This run time algorithm for organising intermediate results is being called *Org_result* which is slightly different from *Org*, used at compile time for the program expression. The *Org* program, after calling *org_exp* and *mark_type*, gets the root task and sets its ancestor to be <u>nul</u>. The *Org_result*, on the other hand, sets this ancestor equal to the A-field of the task under reduction. This action transfers the parentage of the reducing task to the newly created one and helps in maintaining the links in the dynamic graph without any extra communication. It is part of preparation done by the reducing task to ultimately get away from the scene.

7.4.2.3 Communication

The action, till now, has resulted in a task-graph (say G_r) which is either a task or a tagged_value. The result is to be reported

to the ancestor task whose address is available in the A-field of the task under reduction say, w_r. However, if the ancestor is nul then no communication is needed and the reduction cycle can proceed to the next step.

The ancestor of the reducing task can be a Waiting type, waiting for the result from this successor. Depending on the type of result, the reducing task w_r prepares a message M whose type is given by

M.type = result, if G_r is a tagged_value
= link, if G_r is a task

The prepared message is sent to the waiting ancestor, and if the A-field contains a list then a copy of the message is sent to each ancestor. The structure of messages and their handling is discussed in section 7.4.3.

7.4.2.4 Removal

At this stage, w_r has finished its work. The result of its computation has been communicated to the ancestor(s). It has no relevance left now, and therefore it gets away from the computation graph by killing itself.

The complete reduction cycle is specified as procedure 'reduce' in Fig. 7.5 in terms of the symbols used above. The *org_result* referred here is same as *Org* except for a minor difference mentioned earlier. Some additional features of *org_result* are discussed later in section 7.5.

```
procedure reduce;
begin
  if (w_r.F = opoperator)      {beginning of step 1}
  then gr:= primitive_reduce (w_r)
  else gr:= $_reduce (w_r);
  Gr:= org_result (gr);        {step 2}
  if (w_r.A ≠ nul)    {step 3, communicate if ancestor is a task}
  then begin
         if (type (Gr) = 'task')
         then begin
                Gr.A:= w_r.A;  {pass on the A-field of w_r to the
                                                result task G_r}
                M:= (link, Gr.N, w_r.A.s)
              end
         else M:= (result, Gr, w_r.A.s);
         M ! w_r.A  {send the message M to the ancestor(s)}
       end
  else Gr.A:= nul;
  w_r:= null    {step 4, kill self}
end;
```

Figure 7.5 Procedure for reducing an executable task

7.4.3 Message Handling

In the reduction procedure, discussed in the last section, two messages of result and link type have been introduced. In addition, another message of type arg is required while dealing

with a link message. On receipt of a message, a task takes some actions which we call as message handling. These messages and the corresponding actions taken by a recipient task are now being described.

7.4.3.1 Result message

This message is generated by a reducing task w_r whenever the result of its reduction is a tagged_value and not a task. The message structure consists of three fields - type, data and link_no, and is given by

M ::= (type, data, link_no)

type ::= result

data ::= (tag, qty)

link_no ::= w_r.A.s

The type field in any message is necessary for the recipient task to decide its course of action. The data field carries the value type result of reduction where the tag indicates whether the value is a constant, an operator, a function identifier etc.. The link_no field carries a successor number so that the recipient task can know that the required data has come from which of its successors.

A result message may be received by a W, D or P type of task only, and depending on its type, a task takes an appropriate course of action from the following:

W-type: The task simply overwrites its appropriate successor field by the data field of the message and decrements its wait-count in the N-field. As a result, it may become executable if this was the last successor, it was waiting for. In a special case when it has an IF operator in its F-field, the message must have come from the condition part. Depending on the message data (True or False), it initiates a fresh reduction cycle from step 2 (organisation) for the selected successor. The other successor is marked for garbage collection.

P-type: The appropriate successor field is overwritten.

D-type: The message is of use to a dummy task only if the received data is an operator or a function identifier. In that case, it initiates action to organise its successors according to the needs of the function received. Depending on the arity (say m) of this function and the number of successors (say k) available with it, the organisation may result into a Complete (if m=k), Partial (if m>k) or a Complete plus a Dummy (if m<k) task. However, if the received data is a constant, the organiser would indicate an error.

The details of the actions are formally specified in a Pascal-like procedure, named handle_result_message, in Fig. 7.6.

```
procedure handle_result_message (M);
begin
  I:= M.link_no;
  dtag:= M.data.tag;
  case r_task.N.type of    {depending on the type of recipient
                            task (r_task) take one of the actions}
    W : if (r_task.F ≠ opIF)
        then begin
               r_task.N.count:= r_task.N.count - 1; {decrement
                                                   wait-count}
               r_task.S(I):= M.data  {put data in corresponding
                                      successor field}
             end
        else begin  {operator in F-field is an IF operator}
```

```
            c_task:= new(task);

            if M.data.qty = True  {if data is TRUE, organise
                                   then part into c_task}

            then org_exp (r_task.S(2), c_task)

            else org_exp (r_task.S(3), c_task);

            mark_type (c_task);

            if type(c_task) = 'task'  {check if the c_task is a
                                        task or a tagged_value}
            then begin

                   Mc:= (link, c_task.N, r_task.A.s);

                   c_task.A:= r_task.A  {pass on self parentage
                                         to c_task}
                 end

            else Mc:= (result, c_task, r_task.A.s);

            Mc ! r_task.A; {send framed message to ancestor(s)}

            r_task:= null  {kill self}

          end;

    P : r_task.S(I):= M.data;

    else if (dtag = val OR ds) then indicate_error  {constant in
                                                   operator position}
         else convert_dummy (r_task, M)

    end   {case}

end;


procedure convert_dummy (task,M);

begin

  fpc:= get_count(M.data);  {get parameter  count of the function
                              received in the message}
  ul:= get_unwanted_list(M.data);

  s:= task.SC;  {save the no. of successors in the dummy under
                                                     conversion}
  if (fpc > s) then begin

                 p:= s;

                 task.N.type:= 'P'
```

```
                    end
                else p:= fpc;
k:= 1;
while (k ≤ p) do
  begin
    org_exp(task.S(k), task-graph);  {organise a successor}
    link(task.S(k), task-graph); {link the organised task-graph
                                  to the appropriate successor}
    k:= k + 1  {next successor}
  end;
if (fpc = s) then mark_type(task);
if (fpc < s)    {a new dummy task will have to be made}
then begin
       leftovers:= s - fpc;  {no. of unabsorbed successors}
       k:= fpc + 1; {set successor counter next to the organised
                     successor}
       d_task:= new(task);  {create a new task to act as dummy}
       d_task.A:= task.A;  {set its ancestor same as that of the
                            task under conversion}
       task.A:= (d_task.N, 0)  {make d_task the ancestor of the
                                task under conversion}
       task.SC:= fpc;
       d_task.N.count:= leftovers;
       d_task.SC:= leftovers;
       I:= 1;
       while (k ≤ s) do   {make unabsorbed successors as
                           successors of the d_task}
         begin
           link(d_task.S(I), task.S(k));
           I:= I + 1;
           k:= k + 1
```

```
        end;
      link(d_task.F, task);
      mark_type(d_task)
    end
end;
```

Figure 7.6 Specification of handle_result_message procedure

7.4.3.2 Link message

This message is generated by a reducing task w_r whenever the result of its reduction is another task, say w_t, instead of a tagged_value. Its structure is given by

M ::= (type, data, link_no)

type ::= <u>link</u>

data ::= w_t.N

link_no ::= w_r.A.s

Data field contains the name of the task w_t, created during reduction, and the link_no field, like in <u>result</u> message, contains the successor number for proper linking. A recipient task, depending on its own type, takes a corresponding course of action described as follows:

<u>W or P type</u> : The appropriate successor field is adjusted to point to the task w_t mentioned in the data field of the message. In addition, the W type decrements its wait-count if the type of w_t is P.

<u>D type</u> : The task-name, contained in the data field of the received message, would indicate whether w_t is of W, E, P or D type. If it is W or E type, the recipient dummy cannot do anything with it. It simply adjusts its F-field to point to

w_t. If it is P type, the dummy may get rid of some or all of its successors depending on its own surplus and the shortage with the P type. Similarly, if it is D type, there is no point in keeping two dummies in series and the recipient dummy passes on all its successors to w_t and gets away from the scene. However, before killing itself, it sends a link message to its ancestor for linking him to w_t. In the last two cases, the surplus successors, with the recipient dummy, are passed down through an arg_message discussed in the next section.

A detailed formal description of handling a link message is given in Fig. 7.7.

```
procedure handle_link_message (M);
begin
  I:= M.link_no;
  tag:= M.data.type;   {data received is the N-field of a task}
  r_task.S(I):= M.data;
  rcount:= r_task.N.count;
  rtype:= r_task.N.type;
  if rtype = 'W' then if tag = 'P' then rcount:= rcount - 1;
  if (rtype = 'D')
  then begin
         es:= rcount; {set es to surplus count with recipient
                       dummy}
         s:= M.data.count;
         if es > s
         then if tag = 'P' {received task-name indicates P type}
              then begin
                     rcount:= es - s;  {update self count}
                     list:= (r_task.S(1), ..., r_task.S(s));
                     Ma:= (arg, list, no_change); {prepare message
                                                   to send s successors}
```

```
                Ma ! r_task.S(I);   {send arg message to the
                                     task whose name appears
                                     in the link message}

                shift_left (r_task, s) {shift the remaining
                                        successors to the
                                        left by s}
              end

       else if (tag = 'P' OR 'D')

            then begin

                list:= (r_task.S(1), ..., r_task.S(es));

                Ma:= (arg, list, r_task.A); {prepare message
                                          to send es successors}
                Ma ! r_task.S(I);

                if tag = 'P' then M.data.count:= s - es

                              else M.data.count:= s + es;
            {modify the count field of the received task-name}

                Ml:= (link, M.data, r_task.A.s);

                Ml ! r_task.A; {link ancestor with the
                                 received task-name}
                r_task:= null

              end

      end

end;
```

Figure 7.7 Handling of link message

7.4.3.3 Argument (arg) message

This message is generated by a dummy task while handling a link message. It is sent to the task whose name is received in the link message. Through this message, a dummy gets relieved of some or all of its successors whenever the name of a partial or dummy task is received in link message. The structure of the message,

when generated by a dummy task w_d, is

M ::= (type, data, anc)

type ::= <u>arg</u>

data ::= (w_d.S(1), ..., w_d.S(n)), n = no. of successor fields to be sent

anc ::= w_d.A | no-change

Its data field contains an ordered list of the dummy's successor fields whose length is equal to the number of successor names to be passed down. Through the 'anc' field, the sender dummy communicates its A-field (for linking) when it is planning to kill itself otherwise an indication, meaning that no change is required in the recipient's A-field, is sent.

The recipient of an <u>arg</u> type message can be a P or D type only. On receipt of the message, the two types take appropriate action as given below:

<u>P type</u> : The ancestor field is adjusted according to the 'anc' field of the message. The unorganised successors, received through the 'data' field, are organised and added at the end of the existing array of successors.

<u>D type</u> : A dummy recipient performs the same actions as above except organising the received successors.

A formal description of the handling of argument message is given in Fig. 7.8.

```
procedure handle_arg_message (M);
begin
  n:= length (M.data);    {set n to number of arguments received}
```

```
k:= r_task.SC;    {initialise successor counter to the existing
                   number in the recipient task}

I:= 0;   {initialise argument counter}

count:= r_task.N.count;

if r_task.N.type = 'D' then count:= count + n

else count:= count - n;  {adjust the surplus/shortage count}

r_task.N.count:= count;  {set the count to the adjusted count}

while I < n do

  begin

    I:= I + 1;

    k:= k + 1;

    r_task.S(k):= M.data(I)

  end;  {while ... do begin}

r_task.SC:= k;

if (M.anc ≠ 'no-change') then r_task.A:= M.anc;

if r_task.N.type = 'P'

then begin

       k:= k - n + 1;  {set the successor counter at the first
                         new successor}
       I:= 1;

       ul:= get_unwanted_list (r_task.F);

       while I ≤ n do

         begin

           if in_list (k, ul) {check if k is in unwanted list}

           then begin

                  mark_for_GC (r_task.S(k));

                  r_task.S(k):= 'UNWANTED'

                end

           else begin
```

```
                arg_task:= new(task);  {create a new task for
                                         a new successor}
                org_exp (r_task.S(k), arg_task);

                link (r_task.S(k), arg_task)

              end;

          I:= I + 1;  {next argument}

          k:= k + 1;  {next successor}

        end;  {while ... do begin}

      if count = 0  {partial task has become complete type}

      then begin

            r_task.N.type:= ' ';  {prepare to mark the type
                                    afresh}
            mark_type (r_task)

          end

    end

end;
```

Figure 7.8 Specification of the procedure for handling arg message

7.5 MODIFICATIONS

The task reduction mechanism, discussed in the previous section, is based on a purely applicative reduction strategy. Only those tasks are selected for execution whose F-field function is an operator or a user defined function and the successors are in tagged_value/partial-task form. This corresponds to an eager order where the arguments are first reduced to value form before being substituted in a function body. Issues such as recursion, laziness in the computation of shared sub-expressions, safety of computation etc. have been avoided in that discussion in order to

keep the initial explanations simple. This section takes up these issues and appropriate modifications to the model, to accommodate these features, have been suggested.

7.5.1 Laziness

Laziness is a useful property for avoiding wasteful work. It advocates that an expression be computed only when needed and that too only once. Thus one of its requirements is that the decision to undertake a computation be based on need rather than the availability of data and thus requires the ability to postpone a computation and activate it again when the need is more clearly established. In the task model, the process of organisation, acting as a necessary (but not a sufficient) condition for starting a computation, serves this purpose. It has already been used in conditional expressions for avoiding the computation of a discardable work. Similarly, the successors of a dummy task are left in an unorganised form because their need is not clear at the time of task creation. To accommodate this, provision for a successor to point to an unorganised graph (Def. 7.2 for task fields) has been kept.

The other requirement of laziness is that a needed expression should not be computed more than once. This aspect of laziness is connected with sharing of sub-expressions. In task structure, sharing is allowed through multiple ancestors (a list in the ancestor field). There is no risk of repeated computations when a complete task is shared because the applicative strategy will reduce the shared task to value form and communicate the result to

all those sharing it.

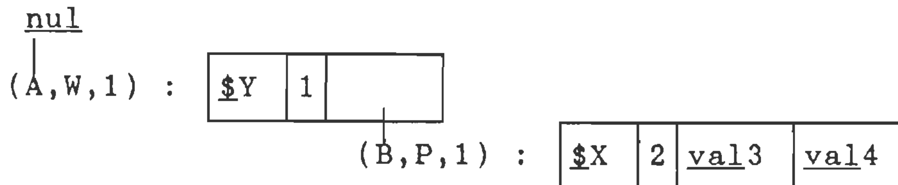
A situation of particular interest is the sharing of partial application [84] of a function i.e. a partial task. In the task reduction mechanism discussed so far, partial tasks were allowed to be passed down as arguments to other complete tasks. This approach has the risk of repeating some computation enclosed in the partial task if the function, to whom it is passed as argument, has several references for it. This is so because the reduction procedure copies an argument for multiple requirements. The situation is similar to a conventional language program loop having some computation not involving the loop variables. Let us consider the following program to illustrate the risk of repeated computations:

```
------------------------------
$X a b c = * (+ (SQ a) b) c
$Y h     = * (h 5) (h 6)
------------------------------
$Y ($X 3 4)
```

In this example, the program expression contains a partial application ($X 3 4). Looking at the definition of $X, it can be seen that the partial application has a computable part (+ (SQ 3) 4) for the two available arguments. Further in the definition of $Y, the parameter h occurs twice. Thus if the partial application ($X 3 4) is passed as argument to $Y then the computable part (+ (SQ 3) 4) will be computed twice.

In the task reduction, the above problem is solved by detecting whether a partial task has a computable part, and if it has one

then it is allowed to reduce before being passed as argument to another task. Continuing with the example, the program expression, put in task-graph form, is written textually as [$Y ([$X (val3, val4, ...])], and in graph form as



Allowing partial task B to reduce (because it has a computable part) leads to an 'incomplete' instance of the function $X where one director, corresponding to the argument not available, is left in the SDS term of $X. This incomplete instance is being called a **residual definition.** A residual definition is like a def_graph having directors for the arguments that could not be provided by the partial task. For the example, the residual definition resulting from the reduction of task B is shown in Fig. 7.9(a). This residual definition, after organisation (through an *Org_def* module), is shown in Fig. 7.9(b), and after reduction of the two complete tasks, it takes the form shown in Fig. 7.9(c). The single director on root node is for the third argument which was not available. A pointer to this residual definition is passed by the partial task B, as result, to its ancestor. There is no possible computation left in the residual definition at this stage and hence there is no risk of losing laziness in copying it for multiple references.

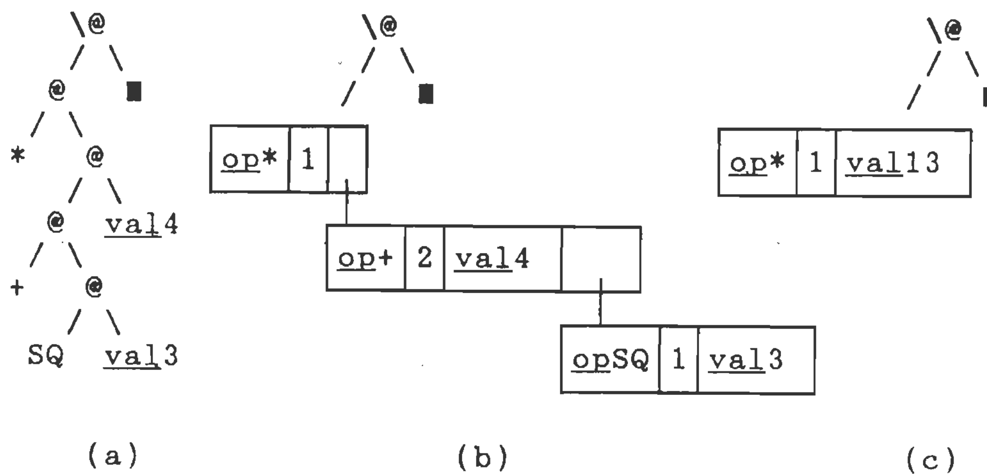


Figure 7.9 Treatment of residual definition in the example - (a) initial residual definition (b) after organisation (c) after reduction

The question of detecting whether a partial application has a computable sub-expression or not, has been resolved at compile time for each definition through computability analysis (section 6.4). The CN sub-field of the header in each D-code has the relevant information. When a partial task is created, it knows that it has a computable part if the number of successors available is greater than or equal to the computability number CN contained in the D-code of its F-field function. Partial tasks with built-in operator in the F-field are assumed to have no computability.

In view of the above discussion, the conditions of reducibility of a task (section 7.4.1) may be modified as follows:

(1) The task is complete and its F-field function is either a built-in operator or user defined or a residual definition.

(2) All the successors are either in tagged_value form, or partial type tasks having no computability.

OR

(1) The task is partial type having some computability ($SC \geq CN$) and has more than one ancestor, or has one ancestor only but is asked to reduce.

Thus if a successor S(I) of a W type task is a partial type with some computability then the task checks for multiple occurrences of the Ith argument in the D-code of its function (indicated by the presence of the number I in sharing list SL). If sharing is found then S(I) is asked to reduce. For a smooth working of this idea (without any extra communication), it is proposed to qualify the computable partial type as Computable-partial (CP) and non-computable as simply partial. These types can be indicated in the N-field during organisation so that a waiting type parent need not find, through communication, about the nature of a partial successor.

The *Org_def* module, introduced above for organising a residual definition, is similar to *Org-exp* except that the former does not organise holes or the nodes having some directors. The idea has been illustrated in Fig. 7.9(b) in the example.

In view of the modified clauses for reducibility, the organisation process for intermediate results needs to take care of residual definitions, in addition to partial tasks that are passed as arguments without reduction. As mentioned earlier in section 7.4.2.2, the intermediate results are organised through a *Org_result* module which is similar to *Org* but with some additional features which are now given in Def. 7.8. Here, 'rd' stands for a residual definition, <u>rd</u> is a tag for indicating pointer-type, r is

the number of directors at the root node in a residual definition, and s is the shortage count of a partial task.

Definition 7.8 : ***Org_result*** **- extra features as compared to** ***Org***

Org_result(rd) = rd^*Org_def*(rd)

Org_result(rd e_1 ... e_k)

  = [rd^rd (w_1, ..., w_k, ...], $r > k$

  = [rd^rd (w_1, ..., w_k)], $r = k$

  = [[rd^rd (w_1, ..., w_r)] (e_{r+1}, ..., e_k)], $r < k$

Org_result([w_0 (w_1, ..., w_k, ...]) = [w_0 (w_1, ..., w_k, ...]

Org_result([w_0 (w_1, ..., w_k, ...] e_1 ... e_j)

  = [w_0 (w_1, ..., w_k, w_{k+1}, ...,w_{k+j}, ...], $j < s$

  = [w_0 (w_1, ..., w_k, w_{k+1}, ...,w_{k+j})], $j = s$

  = [[w_0 (w_1, ..., w_k, w_{k+1}, ...,w_{k+s})] (e_{s+1}, ..., e_j], $j > s$

---

where w_{k+1} is a task-graph coming from the organisation of e_1, w_{k+j} from e_j etc.. While organising a residual definition in an operand position, the *Org_result* yields a rd type pointer to a task-graph generated by organising the residual definition through *Org_def* which has been described earlier qualitatively. In the operator position, a residual definition is like any other function requiring as many arguments as the number of directors at its root node. The *Org_result*, in this case, can take any of the three courses of action depending on the values of r and k. Similarly, for a partial task in operand position, the *Org_result* returns the task without any change, but when it occurs in operator position, applied to j expressions, the result is a

complete task (if j=s), a partial task (if j<s), or a dummy task (if j>s). Definition 7.4 for *Org*, combined with this definition, forms a complete definition for *Org_result*.

Continuing the example further, the task A reduces using the residual definition of Fig. 7.9(c), to the form shown in Fig. 7.10(c) while passing through the intermediate steps shown in part (a) and (b) of Fig. 7.10. It may be noted that *Org_result* has been used in arriving at the final task-graph (Fig. 7.10(c)) from part (b) of the figure. At this stage, the tasks D and E are both executable and the reduction proceeds in the usual manner.

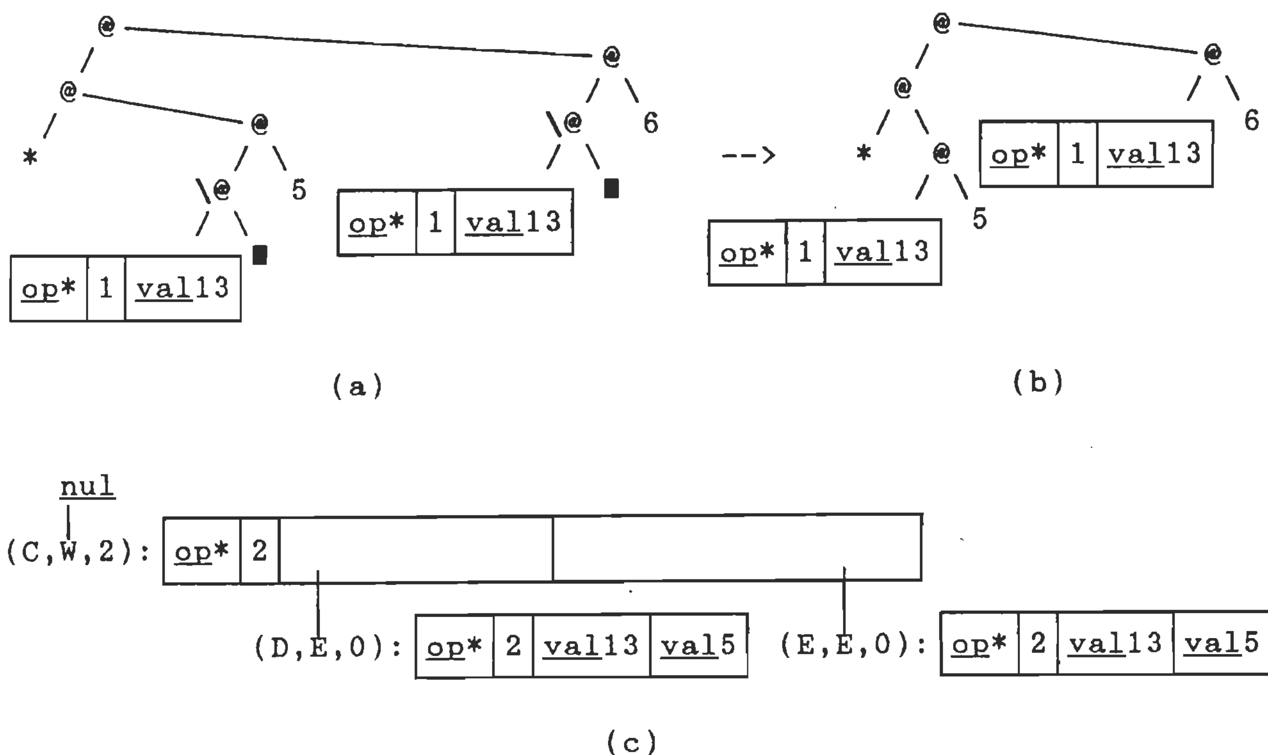


Figure 7.10 Use of residual definition in the example - (a) body of $Y with residual definition substituted in it, (b) the body with arguments substituted into holes, and (c) the body after organisation through *Org_result*

Another area which requires laziness, is the handling of data structures. Representation of infinite data structures is not possible unless the data constructors are handled lazily. In Fig. 7.3, for *Org* algorithm, care has been taken for this laziness in the form that the *org_ds* procedure does not organise the successors of a data structure node. This way, action on reducing the components of a structure is not initiated and the reduction is saved from getting stuck up in computing the components of an infinite data structure. If at any time, the structure is broken apart then the action on a component may start, but it again stops at any inner structural node. Example 3 in the Appendix illustrates the utility of this facility.

7.5.2 Simple Recursion

In functional style of programming, recursion is the main ingredient. It can be handled through the fixed point combinator Y treated as a built-in operator. It has already been explained in Chapter 2 that the applicative order cannot handle recursion because in its eagerness to compute everything, it gets entangled in the non-terminating task of reducing the application (Y H) to normal form. Thus, if recursion is to be accommodated then it is necessary to modify the reduction strategy so as to be able to postpone the reduction of (Y H) till some appropriate time.

Revesz [106] has suggested a controlled recursion through an operator Y' which can fire only once. With this combinator, the reduction rule is

$$Y' \; H = H \; (Y \; H)$$

Here, Y is a disabled combinator which cannot initiate further reduction. Thus the expression (Y H) is in normal form. In order to restart the blocked reduction, Revesz suggests a modification to the ß-reduction rule which states that during a ß-reduction an occurrence of Y be changed to Y'.

Using the above idea and preserving the spirit of coarse grain architecture, the Y operator in task model fires only on the availability of as many arguments as required by H (and not of a single argument). Y, here, is a built-in operator with no fixed arity of its own. It acquires the arity of the function to which it is applied. Thus, if H is a function of two parameters then (Y H) is a partial task whereas (Y H A) is a complete one. The reduction rule for Y then is

$$Y \; H \; A \; \text{-->} \; H \; (Y \; H) \; A \qquad \ldots(7.1)$$

The presence of the argument A, on the left-hand side, acts as a 'catalytic agent' which does not take part in the actual reduction but without which the reduction does not proceed further whereas, on the right-hand side, the application (Y H) is inhibited from further reduction because Y is short of one argument. In general, if H requires m arguments then the expression $(Y \; H \; a_1 \; a_2 \; \ldots \; a_k)$ is reducible only if $k \geq m-1$.

A Y-task is organised as partial unless the required number of arguments is available. The procedure to organise an expression into a Y-task (*Org_Y*) is defined in Fig. 7.11. It would be called

by *Org_exp* procedure whenever the tip of a spine has a Y operator. The *Org_Y* procedure assigns to Y an arity equal to that of H function. It is assumed that the built-in operator Y has an arity sub-field. A Y-operator with arity r is written as Y_r (Y becomes Y_r during organisation).

```
procedure Org_Y (node,nc,task);
begin
  save:= node;
  node:= node.A;
  h:= node.S(2);     {save the right successor which is assumed to
                      be a $-function or a residual definition}
  Y:= task.F;
  if Y.arity ≠' ') then fpc:= Y.arity    {Y is already having an
                                           arity, take it as fpc}
  else begin
         if h.tag = '$'       {h is user defined}
         then begin
                fpc:= get_parameter_count (h);
                Y.arity:= fpc;
                ul:= get_unwanted_list (h)
              end;
         if h.tag = 'rd'   {h is a residual definition}
         then begin
                node:= get_root_node(rd^rd); {get root node of
                                              residual definition}
                dir:= directors(node);  {find the no. of directors
                                         at root node}
                nc:= 0;
                repeat
                  node:= node.S(1);
```

```
                nc:= nc + 1
            until type(node) = 'leaf' AND node.tag = '$';
            fpc:= get_parameter_count(node);
            ul:= get_unwanted_list(node);
            short:= fpc - nc; {shortage of the function at the
                                tip of residual definition}
            Y.arity:= short + dir
          end
      end;
  task.F:= Y;
  node:= get_node (save);
  make_task (ul, fpc, nc, task, node) {nc and task are available
                                       from Org_exp who has
end;                                   called this procedure}
```

Figure 7.11 Specification of procedure Org_Y

A complete Y-task is of the form $[Y_r\ (H,\ w_1,\ \ldots,\ w_{r-1})]$. If reducible, it undergoes Y-reduction (a special case of primitive reduction) which is expressed as

$$[Y_r\ (H,\ w_1,\ \ldots,\ w_{r-1})] \\ \to [H\ ([Y_r\ (H,\ \ldots],\ w_1,\ \ldots,\ w_{r-1})] \qquad \ldots(7.2)$$

The rule specifies a general case of the function H requiring r arguments. The reduction results in a complete task where operator position is occupied by H, and the first member in the argument list is always a partial Y_r-task having only one argument. This partial task contains the seed of recursion from which a fresh cycle of recursion can be initiated, whenever

required. The reduction now proceeds with the \$-reduction of the main H-task. The partial Y_r-task has no computability and is used as such in the \$-reduction. Thus the total reduction sequence for a complete Y_r-task may be expressed as

\$-reduce (Y-reduce); Org_result; communicate; kill self

It is different from the usual reduction cycle, given in section 7.4.2, in the first step only which is a composite one here, consisting of \$-reduction of the result of Y-reduction. Rest of the steps are same. Example 2 in Appendix illustrates this reduction.

7.5.3 Mutual Recursion

A set of definitions becomes mutually recursive when there are cross references for each other in the definition bodies. In this case, the fixed point operator Y, instead of operating on a single function, operates on a n-tuple of functions expressed as an application $Y(h_1, \ldots, h_n)$. Reduction rule for this kind of expression was given in Eq. 6.13 in chapter 6. The situation is again handled through partial tasks.

In the simple recursion case, an expression (Y H) gets organised into a partial task with no computability. Here the expression $(Y(h_1, \ldots, h_n))$ is organised into a partial task given as

$$P_task = [Y_m\ ((h_1, \ldots, h_n), \ldots]$$

where Y_m is the mutual version of Y. The P_task has a

computability and if asked by an ancestor (through a message reduce), it will reduce, following Eq. 6.13, into a tuple structure δ given by

$$\delta = (A_1, A_2, \ldots, A_n)$$
$$\text{where } A_1 = [h_1 ([Y_m ((h_1, \ldots, h_n), \ldots], \ldots]$$
$$\ldots$$
$$A_n = [h_n ([Y_m ((h_1, \ldots, h_n), \ldots], \ldots]$$

Each of the components of δ is a partial task and in each partial task, the first successor is the P_task. The P_task contains the 'group seed' of mutual recursion from which a fresh recursion cycle can be initiated whenever required. The structure δ acts as an argument to the ancestor who asked for the reduction of P_task and the choice of a component A_i ($i = 1$ to n) for substitution is decided by a list director in the SDS term being used by the ancestor for instantiation.

The ancestor now organises the instance and in the process, the chosen partial task A_i may get organised into a complete task W_i of the form

$$W_i = [A_i (w_1, \ldots, w_{r_i-1})]$$
$$= [h_i ([Y_m ((h_1, \ldots, h_n), \ldots], w_1, \ldots, w_{r_i-1})]$$

where r_i is the arity of the function h_i. If task W_i is reducible, it may start reducing using the SDS term for h_i. However, before doing so, it asks the first successor (the P-task) to reduce who obliges by returning the structure δ again. The task W_i, after receiving the structure δ as result, appears as

$$W_i = [h_1 ((A_1, \ldots, A_n), w_1, \ldots, w_{r_i - 1})]$$

It now reduces in the usual manner and the process continues till recursion terminates. If h_i refers to an h_j ($j = 1$ to n) then it is available from the first argument (the group seed of mutual recursion) by choosing the corresponding A_j.

The recursion has been controlled here by making $Y(h_1, \ldots, h_n)$ a partial task which reduces to the structure δ only when asked. The structure contains the full seed of recursion for future use.

7.5.4 Pattern-matching

A pattern-matching definition consists of n clauses out of which the one applicable in a particular task is selected after matching the arguments with the parameter structures in each clause. We are, at present, concerned with the clause selection procedure. A function f of m arguments and having n clauses gets compiled into a D-code which, besides some information in the the header field, has a n-list of pairs in the SDS field. The list is expressed as

$$[\langle (T(p_{11}), \ldots, T(p_{1m})), SDS(E_1)\rangle, \ldots, \\ \langle (T(p_{n1}), \ldots, T(p_{nm})), SDS(E_n)\rangle]$$

Each pair in the list has a sequence of pattern-types corresponding to the parameters involved in the clause, and a SDS term. Whenever a function has more than one clause, a process of matching is necessary before the function can be applied. Therefore, a task referring to a pattern-matching definition in its F-field, has to undergo an additional step of matching before

starting the reduction cycle of section 7.4.2. The complete process of evaluation starts with the matching step and when the proper clause has been selected, the process continues with the four steps of reduction cycle as if the function had only one definition. The implementation of the matching step is being described now.

A pattern-matching task has to match the arguments according to the algorithm discussed in section 4.2.1. The task is written as

$$pm_task = [f\ (a_1,\ \ldots,\ a_m)].$$

In order to make full use of the multiprocessor environment, while doing the bulky work of matching, the pm_task creates n first level **slave tasks** expressed as

$$S_1 = [Match_list\ ((\tau_{11},\ \ldots,\ \tau_{1m}),\ (a_1,\ \ldots,\ a_m))]$$

$$\ldots$$

$$S_n = [Match_list\ ((\tau_{n1},\ \ldots,\ \tau_{nm}),\ (a_1,\ \ldots,\ a_m))]$$

where $\tau_{ij} = T(p_{ij})$, $i = 1$ to n, $j = 1$ to m. Each slave is entrusted with the task of matching the sequence of pattern-types of a clause against the arguments. A slave returns a boolean reply. For more parallelism, each first level slave further creates m second level slaves of the kind

$$S_{i1} = [Match\ (\tau_{i1},\ [Tag\ (a_1)])]$$

$$\ldots$$

$$S_{im} = [Match\ (\tau_{im},\ [Tag\ (a_m)])],\quad i = 1 \text{ to } n$$

Thus the initial n slaves create nxm slaves. The situation is

shown in Fig. 7.12. One set of m second level slaves is answerable to one first level slave. If the reply from any one of them is 'False' then the first level slave sends a 'False' reply to the master without waiting for other replies. It rather stops further execution of its slaves by killing them because the result is not required. A clause fails to match if any one pattern-matching fails.

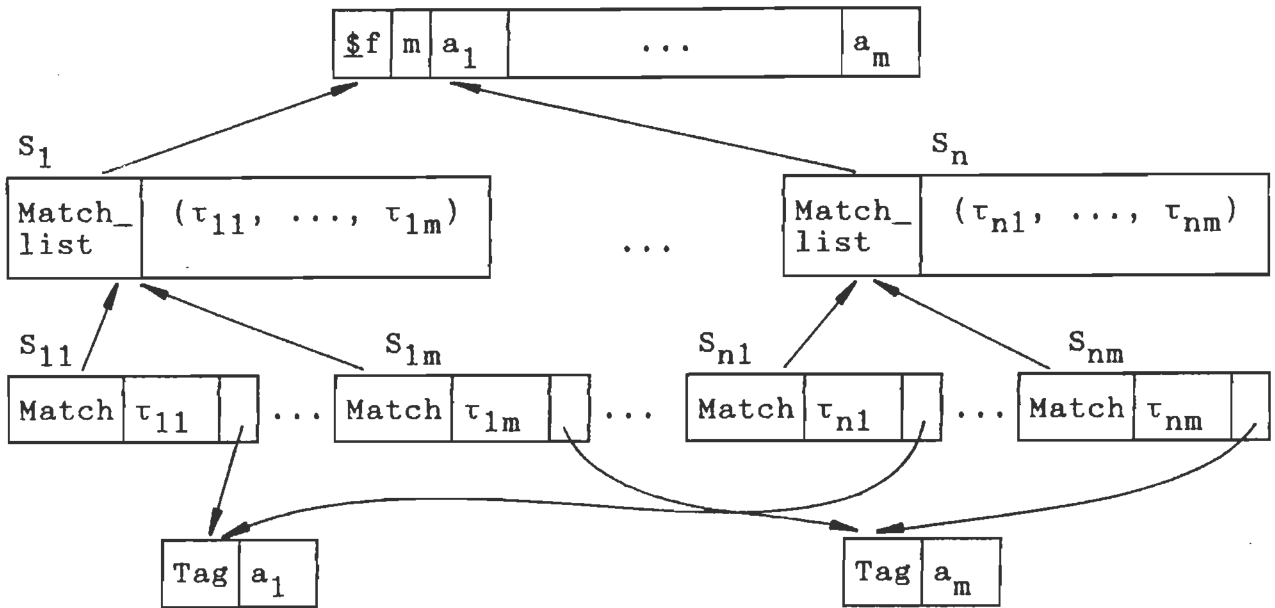


Figure 7.12 Hierarchy of slave tasks created for matching

The Tag-tasks are shared by second level slaves. A tag computation benefits all because a copy of the computed structure tag is supplied to each slave. This is better than copying the whole argument for each second level slave to do its matching. Whenever a set of second level slaves is killed, the Tag tasks remove their names from their ancestor list. It is possible that

the argument in a Tag task is a structured one. A Tag task kills itself after one tag computation if the argument was a simple one, otherwise it waits for a request from a slave to compute the inner tags of the structure. This is necessary in order to be able to handle infinite data structures.

After the matching process is over, the slaves no longer exist and the pm_task is through with the clause selection. The reduction then proceeds in the usual manner. The parallel activity in matching is at the cost of an additional communication overhead. The finer details of the matching step have not been worked out but they seem to pose no special problems.

7.6 CONCLUDING REMARKS

Supercombinator reduction is a coarse-grain model aimed at decreasing the amount of communication overheads in a multi-processor reduction machine. SDS terms, introduced to represent a supercombinator body as a variable-free structured director string graph, have simple and easy-to-follow-by-machine reduction rules. A task reduction model has been proposed which utilises SDS terms for efficient template-instantiation of supercombinators.

A task is expressed as a record structure and when reducible, it represents either a supercombinator redex or a redex involving a built-in operator. An algorithm to organise a given expression into a task-graph has been developed and specified in a Pascal-like notation. In a task-graph, the nodes are tasks and the links are maintained through ancestor and successor fields of the task

structure. Two neighbouring tasks refer to each other by name field which includes, besides an identifier, some more information about task type. The extra informative name field, though bigger in size, decreases the need for inter-task communication.

The task-graph, combined with definition graphs (SDS terms), forms the input to task reduction model. A task, during reduction, applies a function to all the arguments, organises the result into task-graph, communicates the organised result to its ancestor(s), and kills itself. The methodology draws some ideas from CTDNet [58,59] but the design leads to much less communication overhead. The bigger size of task compared to a process in CTDNet is also responsible for this. The task model has only three types of messages (<u>result</u>, <u>link</u> and <u>arg</u>) compared to eight in CTDNet.

The model exploits parallelism by reducing tasks in an applicative style. However, selective laziness has been introduced wherever useful. Delaying of the computation of an expression is achieved by postponing its organisation. This has been utilised in the handling of conditional expressions and data structure nodes.

The concept of a partial task, introduced to represent a partially applied function, is usefully employed in handling shared partial applications lazily. A partial task knows, through the information provided by the compile time computability analysis, whether it has a computable part or not. If it has, and the task is shared then it is allowed to reduce with the available arguments before being passed as argument to another task.

The reluctance of a partial tasks to reduce on its own, has been

utilised in realising a controlled recursion through Y combinator in a model which is basically applicative in its reduction strategy. The use has been extended to mutual recursion also, and is based on the treatment of mutual recursion developed in chapter 6.

A parallel implementation of the pattern-matching algorithm, developed earlier, has been suggested in the model by introducing slave tasks. A task required to do matching creates slave tasks to get the matching of pattern-types and argument tags done in parallel. Slaves are special purpose tasks created to do a specific job and are killed subsequently.

CHAPTER 8

CONCLUSIONS

The work presented in this thesis is an attempt towards designing a coarse-grain computation model for executing functional programs in a multiprocessor environment. In the model designed, a grain of computation is a supercombinator redex, and the pre-processing of supercombinator definitions consists of compilation into a variable-free annotated-graph structured code (SDS terms). The reduction of a program expression is done using the graphical code as a template for function instantiation. The order of reductions is basically applicative so as to allow more parallelism, but selective laziness has been incorporated to handle conditionals, recursion and infinite data structures. The model has some safeguards against non-termination of the applicative order.

The design considers definition styles which allow pattern-matching, and the use of arbitrary nesting of simple/recursive local definitions in the bodies. The expressions are allowed to have data structuring through constructor functions.

8.1 CONCLUSIONS

The principal features/conclusions of the research work, reported in the thesis, are as follows:

(1) Lambda calculus is a powerful mathematical tool for expressing the operational semantics of functional programs. To support pattern-matching, recursion etc., enriched notations of lambda calculus are available where abstractions may bind patterns besides variables [11], and list manipulations are permissible [33,106]. An interpretation of pattern-matching into such an extended lambda notation has been developed. The approach takes an isolated view of the two aspects of implementation of pattern-matching viz. matching and reduction. Through isolation, we are able to handle the matching work in parallel without activating the clause bodies. After matching is over, the reduction is done for the selected clause only. This is in contrast to other interpretations [11,109] which take a combined view of the two processes and carry them out together, with the result that the dynamic graph size may become large because all the clause bodies are kept alive till a match failure occurs.

(2) The isolated view has led us to rename the pattern-matching lambda abstractions as pattern-binding lambda abstractions. An application of such an abstraction to a matched structured object cannot be reduced through the ordinary β-reduction rule of lambda calculus. Accordingly, a set of modified β-rewrite rules has been proposed. The modified rules allow substitution of a *component* of the structured argument for a free occurrence of a *component* of the bound pattern, and thus form a basis for handling data structures through lambda calculus.

(3) An algorithm for the matching work has been developed which is utilised at run time for performing the matching operations in parallel. The algorithm has been specified through functions defined in functional programming style.

(4) As supercombinators do not have any fixed reduction rules (they are user defined), a pre-processing of their definitions is essential for efficient reduction. The proposed model aims at supercombinator reduction through template instantiation, and for this, a pre-processing scheme has been suggested where each definition body is compiled, through a process of abstraction, into a variable-free form named as SDS (Structured Director String) term. The idea of SDS terms is a generalisation of DS terms [71] by allowing the parameters of the abstraction process to be patterns besides variables. The DS terms use simple atomic directors which imitate the ordinary ß-rules, whereas the structured directors (list and pattern types), used in SDS terms, are designed to express the intention of the proposed modified ß-rules. For instance, a list director indicates the breaking of a structured argument into components for substitution into a definition body. The breaking process is made lazy, through the use of pattern director, so that it is done only when essential and that too just upto the necessary extent.

(5) An algorithm for generating SDS terms, by abstracting out the parameters of a pattern-matching definition from its various clause bodies, has been designed. It consists of pattern-

abstraction rules for different types of patterns and various syntactical forms of expression forming a body. The rules show a preference to use pattern directors, as far as possible, so that unnecessary structure breaking is avoided. The complete pattern-abstraction algorithm has been coded into Turbo-C and tested successfully on IBM PC. The test runs for compiling different types of definitions have confirmed that the algorithm is lazy in assigning a list director responsible for structure breaking.

(6) To allow the use of **let** and **letrec** blocks in a definition body, the notation of SDS terms has been enriched by incorporating pointers and a concept of context-list (C-list). These additions to SDS term syntax allow them to express sharing of sub-expressions which is the purpose of local definitions. Their use yields smaller sized terms as compared to the alternative of using the lambda calculus meaning of local definitions, directly, for generating SDS terms. By having pointers, the occurrences of local variables get connected to the relevant expressions in the C-list at compile time only instead of waiting for a β-reduction to do it at run time.

(7) Another abstraction algorithm has been developed for taking care of local definitions. The algorithm takes into account simple let-definitions, a single recursive definition in a **letrec** block and mutually recursive definitions. The mutual recursion case has been interpreted into the SDS term notation using the concepts of pattern-abstraction because a

set of n mutually recursive definitions is seen as one definition for a n-tuple structured pattern [11,31]. Here an intuitive reduction rule for the Y operator applied to a n-tuple of functions, named as mutual Y rule, has been suggested. Our only support to this is an extension of an analogy of simple recursion to plant life.

(8) The results of pre-processing of supercombinator definitions are packed into a D-code structure of which SDS term is one component. Another component is a sequence of pattern-types of the parameters of a clause which is used for the matching purposes. These two components together constitute the SDS field of D-code. The code has a header field also which encloses miscellaneous house-keeping information used for some optimisations in the reduction process. A special information in it is the computability number (CN) generated through a computability analysis of a definition body. The number helps the reduction process in avoiding repeated computations whenever a partial application of a function is shared. The algorithm for computation of CN has been specified in functional style.

(9) For reducing the program expression using D-code, a task structure has been designed which represents a super-combinator or a built-in operator applied to all its arguments as an indivisible work packet. The idea is similar to a *current context stack* in G-machine [89] or the *packet* in Flagship model [95], but the supporting code is SDS term

rather than G-code or an imperative code. The SDS terms have simple and fixed reduction rules which can be coded into hardware. It is felt that the fixed nature of these rules eliminates the need for an extra level of software interpretation which does not seem to be true for the G-code or the imperative code of Flagship.

(10) The structure of task-fields has been designed to keep the requirements for inter-task communication low. Specifically, the name-field of a task carries all the information about task-type so that a neighbouring task in the graph need not communicate for finding neighbour-type. It is like assigning types to the linking arcs in a directed graph.

(11) Two special task types, namely the partial and dummy, have been introduced to take care of a function applied to too few or too many arguments. These tasks generally do not take any action on their own and wait for messages from other tasks to act accordingly. However, using the CN information from D-code, a partial task is allowed to reduce whenever it has a computable part and is shared. This makes the reduction lazy in such cases because repeated evaluations of the computable part, due to sharing, are eliminated.

(12) In the model, a program expression becomes ready for reduction only after it has been organised into a task-graph. Correspondingly, an unorganised expression remains a inactive part of the overall graph. The idea has been utilised in delaying a computation, whenever useful, such as in a

conditional expression - for postponing action on 'then' and 'else' parts, and in a data structure expression - for delaying computation of the arguments of a constructor function. The latter provides implementation of infinite data structures. The Organise procedure has been specified formally in a *Pascal-like* notation.

(13) The reduction strategy employed in the model is data-driven or applicative so as to exploit parallelism. Although normal order is safe yet it allows less parallelism because of the restriction that only top left-most redex is allowed to be reduced at any time. On the other hand, the applicative order, while allowing several redexes to reduce concurrently, has a tendency to fall into the trap of non-termination. This tendency has been checked to some extent by using the organisation tool as a means of blocking a computation. For example, the unwanted arguments of a function (identified at compile time) are left unorganised. Similarly, a dummy task keeps its successors in an unorganised state because it is not sure about their need. The technique brings some limited safety to the applicative reduction.

(14) The problem of garbage collection is an important one in the implementation of functional models. In our model, a task lives only as long as it is needed. Self killing is a part of its reduction cycle. It is felt that this idea, taken from CTDNet [58,59] - a model dealing with the reduction of expressions represented as lambda graphs, should greatly relieve the burden of garbage collection.

(15) Communication is a necessary evil in a multiprocessor model. Bigger grain size of computation leads to lesser communication. The fact is demonstrated by the simpler and lesser number of messages in the model as compared to CTDNet which is a fine-grain machine. The model has only three types of messages as compared to eight in CTDNet. The actions taken for handling the messages have been formally specified through Pascal-like procedures. These procedures are called by a task whenever it receives a message from some other task.

(16) Recursion through Y combinator is not possible in a purely applicative order machine. However in the task model, the 'inertia' of a partial task has been utilised in realising a controlled recursion. A modified rule for Y operator has been given which permits a Y to fire only in the presence of as many arguments as required by the function to which it is applied. An application (Y H) is cast as a partial task with no computability and hence it is unable to proceed further, unless more arguments are available. A scheme for dealing with mutual recursion has also been proposed using the idea of partial tasks, though at a conceptual level only.

(17) In the application of a pattern-matching function, the work concerned with matching may be quite heavy. Keeping this in mind, the matching is proposed to be done in parallel through creation of large number of slave tasks who carry out work according to the developed matching-algorithm. The idea has

not been fully integrated into the task structure, and its implications from communication requirement point of view are not very clear yet.

8.2 RECOMMENDATIONS FOR FUTURE WORK

(1) The model has been specified to a great extent through formal algorithms. Based on this design, the work can be further extended by carrying out simulation studies. This will provide a quantitative performance analysis and an opportunity to incorporate optimisations, and have comparison with other existing models. Similarly, some work in the direction of designing a physical architecture for the model may be carried out.

(2) The safety aspect of the model can be further improved by adding strictness analysis [111] information in the header field of the D-code of a definition. Work by Mycroft [112] and Burn [113] on abstract interpretation, which gives the definedness of a function in terms of the definedness of its arguments, is relevant here. Evaluation transformers, designed by Burn [114,115], tell that, given how much evaluation is allowed by a function application, how much evaluation of the arguments can be allowed without jeopardising safety. It is expected that integrating the information extracted through this kind of analysis into the D-code structure could make the applicative order task reduction perfectly safe and capable of exploiting parallelism also at the same time.

(3) List comprehensions, analogous to set comprehensions in Zermelo-Frankel set theory, were first used by Turner in KRC [23] as ZF expressions. They have since been included in several other functional languages such as Miranda [26], SASL [21] and Orwell [27]. It is recommended that these constructs be translated into SDS terms to enrich the notation further and enhance their utility as an intermediate form for machine interpretation.

[1] Burks A.W., Goldstine H.H., and von Neumann J., "Preliminary discussion of the design of an electronic computing instrument", *Computer Design Development: Principal Papers*, by E.E.Schwartzlander Jr (ed.), pp.221-59, Hayden Book Co., 1976.

[2] Backus J., "Can programming be liberated from the von Neumann style ? A functional style and its algebra of programs", Comm. ACM, vol.21, no.8, pp.613-641, Aug. 1978.

[3] Paker Y., *Multi-Microprocessor Systems*, APIC Studies in Data Processing, vol.18, Academic Press, 1983.

[4] Flynn M.J., "Very high speed computing systems", Proc. IEEE, vol.54, no.12, pp.1901-9, Dec. 1966.

[5] Cray Research, "CRAY-1", Computer System Reference Manual, Cray Research Inc. Minneapolis, Minn., 1976.

[6] Reddaway S.F., "DAP - a distributed array processor", 1st Annual Symp. on Computer Architecture, Florida, pp.61-65, Dec. 1973.

[7] Hockney R.W., and Jesshope C.R., *Parallel Computers*, Adam Hilger Ltd, Bristol, 1981.

[8] Dennis J.B., "First version of a data flow procedure language", LNCS vol.19, pp.362-376, Springer-verlag, 1974.

[9] Dennis J.B., and Misunas D.P., "A preliminary architecture for a basic data flow processor", in Proc. 2nd IEEE Symposium on Computer Architecture, p.126, Jan. 1975.

[10] Berkling K.J., "A computing machine based on tree structures", IEEE Trans. Comp., vol.C-20, pp.404-418, Jan. 1974.

[11] Peyton Jones S.L., *The Implementation of Functional Programming Languages*, PHI series in Computer Science, PHI(UK), 1987.

[12] Church A., *The Calculi of Lambda Conversion*, Princeton University press, Princeton, N.J., 1941.

[13] Turner D.A., "A new implementation technique for applicative languages", Software - Practice and Experience, vol.9, pp.31-49, Sept. 1979.

[14] Kennaway J.R., and Sleep M.R., "Director strings as combinators", Department of Computer Science, University of East Anglia, 1982.

[15] Hughes R.J.M., "Supercombinators, a new implementation method for applicative languages", in Proc. ACM Symposium on Lisp and Functional Programming, Pitsburgh, pp.1-10, Aug. 1982.

[16] Hudak P., and Goldberg B., "Serial combinators : "optimal" grains of parallelism", in Conf. on Functional Programming and Computer Architecture, Nancy, LNCS vol.201, pp.382-388, Springer-Verlag, 1985.

[17] Fleck A.C., "A case study comparison of four declarative programming languages", Software - Practice and Experience, vol.20, pp.49-65, Jan. 1990.

[18] Turner D.A., "The semantic elegance of applicative languages", in Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, pp.85-92, 1981.

[19] McCarthy J. et al, "Lisp 1.5 programmers Manual", MIT press, Cambridge, Mass., 1962.

[20] Backus J., "The algebra of functional programs : functional level reasoning, linear equations, and extended definitions", in Proc. International Colloqium on Formalisation of programming concepts, LNCS, vol. 107, pp.1-43, Springer-Verlag, 1981.

[21] Turner D.A., *The SASL language manual*, University of St. Andrews, Dec. 1976.

[22] Burstall R.M., MacQueen D.B., and Sanella D.T., "Hope : An experimental applicative language", in Proc. of the Lisp Conference, pp.136-143, Stanford, California, Aug. 1980.

[23] Turner D.A., "Recursion equations as a programming language", in *Functional Programming and its Applications*, by Darlington, Henderson, Turner (ed.), pp.1-28, Cambridge University press, 1982.

[24] Fairbairn J., "Design and implementation of a simple typed language based on the lambda calculus", PhD thesis, Tech. Report 75, University Of Cambridge, May 1985.

[25] Augustsson L., "A compiler for lazy ML", in Proc. of the ACM Symposium on LISP and Functional programming, pp.218-227, Austin, Aug. 1984.

[26] Turner D.A., "Miranda - a non-strict functional language with polymorphic types", in Conf. on Functional Programming and Computer Architecture, Nancy, LNCS vol.201, pp.1-16, Springer-Verlag, 1985.

[27] Wadler P., *Introduction to Orwell*, Programming Research group, University of Oxford, 1985.

[28] Hudak P., and Wadler P. et al, "Report on the functional programming language Haskell", Dept. of Computer Science, Yale University, Dec. 1988.

[29] Henderson P., *Functional Programming Application and Implementation*, Prentice Hall Inc., 1980.

[30] Henson M.C., *Elements of Functional Languages*, Blackwell Scientific Publications, 1987.

[31] Burge W.H., *Recursive Programming Techniques*, Addison-Wesley, 1975.

[32] Glaser H., Hankin C., and Till D., *Principles of Functional Programming*, Prentice-Hall (U.K.), 1984.

[33] Revesz G.E., *Lambda-calculus, Combinators and Functional Programming*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1988.

[34] Schönfinkel M., "Über die Bausteine der mathematischen logik", Math. Ann. vol.92 pp.305-316, 1924.

[35] Curry H.B.,and Feys R., *Combinatory Logic*, vol.1, North Holland, Amsterdam, 1958.

[36] Barendregt H.P., *The Lambda-calculus: its Syntax and Semantics*, North Holland, 1984.

[37] Hindley J.R., and Seldin J.P., *Introduction to Combinators and Lambda Calculus*, Cambridge University press, 1986.

[38] Stoy J.E., *Denotational Semantics*, MIT press, 1981.

[39] Landin P.J., "A correspondence between Algol-60 and Church's lambda notation", Comm. of the ACM, vol.8, pp.89-101, 158-165, 1965.

[40] Church A., and Rosser J.B., "Some properties of conversion", Trans. Amer. Math. Soc., vol.39, pp.472-482, 1936.

[41] Wadsworth C.P., "Semantics and pragmatics of the lambda calculus", PhD thesis, Oxford, 1971.

[42] DeBruijn N.G., "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation with application to the Church-Rosser theorem", Indag. Math., vol.34, pp.381-392, 1972.

[43] Curien P-L., *Categorical Combinators, Sequential Algorithms and Functional Programming*, Research Notes in Theoretical Computer Science, Pitman Publishing Ltd., 1986.

[44] Treleaven P.C., "Computer architecture for functional programming", in *Functional Programming and its Applications*, by Darlington, Henderson, Turner (ed.), Cambridge University press, 1982.

[45] Treleaven P.C., Brownbridge D.R., and Hopkins R.P., "Data-driven and demand-driven computer architecture", ACM Comput. Surveys, vol.14, no.1, pp.93-143, March 1982.

[46] Ackerman W.B., "Data flow languages", IEEE Computer, vol.15, no.2, pp.15-25, Feb. 1982.

[47] Davis A.L., and Keller R.M., "Data flow program graphs", IEEE Computer, vol.15, no.2, pp.26-41, Feb. 1982.

[48] Gurd J.R., "Fundamentals of data flow", in *Distributed Computing*, by F.B. Chambers et al (ed.), Academic Press, 1984.

[49] Watson I., and Gurd J.R., "A practical data flow computer", IEEE Computer, vol.15, no.2, pp.51-57, Feb. 1982.

[50] Gurd J.R., Kirkham C.C., and Watson I., "The Manchester prototype dataflow machine", Comm. ACM, vol.28, no.1, pp.34-52, 1985.

[51] Patnaik L.M., Govindarajan R., and Ramadoss N.S., "Design and performance evaluation of EXMAN: an EXtended MANchester data flow computer", IEEE Trans. Comp., vol.C-35, no.3, pp.229-243, March 1986.

[52] Gurd J.R. et al, "Fine-grain parallel computing: the Dataflow approach", in Proc. of An Advanced Course on Future Parallel Computers, Pisa, Italy, LNCS vol.272, pp.82-152, Springer-Verlag, June 1986.

[53] Gaudiot J-L., "Structure handling in data-flow systems", IEEE Trans. Comp., vol.C-35, no.6, pp.489-502, June 1986.

[54] Böhm A.P.W., and Gurd J.R., "Iterative instructions in the Manchester dataflow computer", IEEE Trans. PDS, vol.1, no.2, pp.129-139, April 1990.

[55] Berkling K.J., "Reduction languages for reduction machines", in Proc. 2nd Int. Symp. Computer Architecture, Houston, pp.133-140, Jan. 1975.

[56] Mago G.A., "A network of microprocessors to execute reduction languages", part 1 and part 2, Int. journal of Computer and Information Sciences, vol.8, pp.349-385 and 435-471, 1979.

[57] Mago G.A., "A cellular computer architecture for functional programming", in Proc. IEEE COMPCON, 1980, pp.179-187.

[58] Gupta J.P., "The realisation of a CTD multiprocessor architecture", PhD thesis, The Polytechnic of Central London, London, UK, 1985.

[59] Gupta J.P., Winter S.C., and Wilson D.R., "CTDNet - A mechanism for the concurrent execution of lambda graphs", IEEE Trans. Software Eng., Vol.15, No.11, pp.1357-1367, Nov. 1989.

[60] Berkling K.J. and Fehr E., "A consistent extension of the lambda calculus as a base for functional programming languages", Information and Control, vol.55, nos.1-3, Oct./Nov./Dec. 1982.

[61] Henderson P., and Morris J.H., "A lazy evaluator", in Proc. ACM symposium on Principles of Programming Languages, pp.95-103, 1976.

[62] Landin P.J., "The mechanical evaluation of expressions", Computer Journal, vol.6, pp.308-320, 1964.

[63] Turner D.A., "Another algorithm for bracket abstraction", J. of Symb. Logic, vol.44, no.2, pp.267-270, June 1979.

[64] Clarke T.J.W., Gladstone P.J.S., Maclean C.D., and Norman A.C., "SKIM - The SKI reduction machine", in Proc. of the ACM Lisp Conf., Stanford, CA, pp.128-135, 1980.

[65] Scheevel M., "NORMA: A graph reduction processor", in Proc. ACM conf. on Lisp and Functional Programming, Cambridge, Mass., pp.212-219, Aug. 1986.

[66] Stoye W.R., Clarke T.J.W., and Norman A.C., "Some practical methods for rapid combinator reduction", ACM Symposium on LISP and Functional Programming, pp.159-166, 1984.

[67] Hudak P., and Kranz D., "A combinator-based compiler for a functional language", in Proc. 11th annual ACM Symp. on Principles of Programming Languages, pp.121-132, Jan. 1984.

[68] Oberhauser H.G., "A fully lazy lambda style graph reducer", SFB 124-C1, 06/1986, Universität des Saarlandes, West Germany, 1986.

[69] Oberhauser H.G., "On the correspondence of lambda style reduction and combinator style reduction", in Proc. of Workshop on Graph Reduction at Santa Fé, New Mexico, USA, LNCS vol.279, pp.1-25, Springer-Verlag, Oct. 1986.

[70] Kennaway J.R., and Sleep M.R., "Novel architectures of declarative languages", Software and Microsystems, vol.2, no.3, pp.59-70, June 1983.

[71] Kennaway J.R., and Sleep M.R., "Director strings as combinators", Internal report no. SYS-C87-06, University of East Anglia UK, Jan. 1987.

[72] Kennaway J.R., and Sleep M.R., "Variable abstraction in O(nlogn) space", Internal report no. SYS-C87-04, University of East Anglia UK, Jan. 1987.

[73] Noshita K., "Translation of Turner combinators in O(nlogn) space", Inf. Proc. Letters, vol.20, pp.71-74, 1985.

[74] Stoye W.R., "The implementation of functional languages using custom hardware", PhD Thesis, Computer Lab., University of Cambridge, May 1985.

[75] Anderson P. et al, "COBWEB-2: Structured specification of a wafer-scale supercomputer", PARLE I, LNCS Vol.258, pp.51-67, Springer-verlag, 1987.

[76] Johnsson T., "Lambda-lifting - transforming programs to recursive equations" in Conf. on Functional Programming and Computer Architecture, Nancy, LNCS vol.201, pp.190-203, Springer-Verlag, 1985.

[77] Hughes R.J.M., "The design and implementation of programming languages", PhD Thesis, PRG-40, Programming Research Group, Oxford, Sept. 1984.

[78] Hoffman C.M., and O'Donell M.J., "Programming with equations", ACM TOPLAS, vol.4, no.1, pp.83-112, 1982.

[79] Barendregt H.P. et al, "Term graph rewriting", Internal report no. SYS-C87-01, University of East Anglia, Norwitch, UK, 1987.

[80] Klop J.W., "Term rewriting systems", in Proc. of the 1st Autumn Workshop on Reduction Machines, Ustica, Italy, Sept. 1985.

[81] Goguen J., Kirchner C., and Meseguer J., "Concurrent term rewriting as a model of computation", in Proc. of Workshop on Graph Reduction at Santa Fé, New Mexico, USA, LNCS vol.279, pp.53-93, Springer-Verlag, Oct. 1986.

[82] van Eekelen M.C.J.D., and Plasmeijer M.J., "Specification of reduction strategies in term rewriting systems", in Proc. of Workshop on Graph Reduction at Santa Fé, New Mexico, USA, LNCS vol.279, pp.215-239, Springer-Verlag, Oct. 1986.

[83] Hudak P., and Goldberg B., "Distributed execution of functional programs using serial combinators", IEEE Trans. Comp., vol.C-34, no.10, pp.881-891, Oct. 1985.

[84] Goldberg B., "Detecting sharing of partial applications in functional programs", in Proc. 3rd Conf. on Functional Programming and Computer Architecture, Portland, Orgeon, USA, LNCS Vol. 274, pp.408-425, Springer-Verlag, Sept. 1987.

[85] Lins R.D., "Categorical multi-combinators", in Proc. 3rd Conf. on Functional Programming and Computer Architecture, Portland, Orgeon, USA, LNCS Vol. 274, pp.60-79, Springer-Verlag, Sept. 1987.

[86] Coussineau G., Curien P-L., and Mauny M., "The categorical abstract machine", in Conf. on Functional Programming and Computer Architecture, Nancy, LNCS vol. 201, pp.50-64, Springer-Verlag, 1985.

[87] Lins R.D., "A new formula for the execution of categorical combinators", in Proc. 8th Int. Conf. on Automated Deduction, Oxford, LNCS vol. 230, pp.89-98, Springer-Verlag, July 1986.

[88] Johnsson T., "The G-machine: an abstract machine for graph reduction", in Proc. Declarative Programming Workshop, University College London, pp.1-19, April 1983.

[89] Kieburtz R.B., "The G-machine: a fast graph reduction evaluator", in Proc. IFIP Conf. on Functional Programming and Computer Architecture, Nancy, LNCS vol.201, pp.400-413, Springer-Verlag, 1985.

[90] Augustsson L., and Johnsson T., "The Chalmers lazy - ML compiler", The Computer Journal, Vol.32, No.2, pp.127-141, 1989.

[91] Peyton Jones S.L., Clack C., and Salkild J., "GRIP : A parallel graph reduction machine", ICL Technical Journal, pp.595-599, May, 1987.

[92] Peyton Jones S.L., et al, "GRIP - A high performance architecture for parallel graph reduction", in Proc. 3rd Conf. on Functional Programming and Computer Architecture, Portland, Orgeon, USA, LNCS Vol. 274, pp.98-112, Springer Verlag, Sept. 1987.

[93] Darlington J., and Reeve M., "ALICE - a multiprocessor reduction machine for the parallel evaluation of applicative languages", in Proc. ACM Conf. on Functional Programming Languages and Comp. Architecture, New Hampshire, pp.65-75, Oct. 1981.

[94] Harrison P.G., and Reeve M.J., "The parallel graph reduction machine, ALICE", in Proc. of Workshop on Graph Reduction at Santa Fé, New Mexico, USA, LNCS vol.279, pp.181-202, Springer-Verlag, Oct. 1986.

[95] Watson I. et al, "Flagship computational model and machine architecture", ICL Technical Journal pp.555-574, May 1987.

[96] Townsend P., "Flagship hardware and implementation", ICL Technical Journal pp.575-594, May 1987.

[97] Watson P., and Watson I., "Evaluating functional programs on the FLAGSHIP machine", in Proc. 3rd Conf. on Functional Programming and Computer Architecture, Portland, Orgeon, USA, LNCS Vol. 274, pp.80-97, Springer-Verlag, Sept. 1987.

[98] Greenberg M., and Woods V., "Flagship - a parallel reduction machine for declarative programming", Computing and Control Eng. Journal, pp.81-86, March 1990.

[99] Greenberg M., Woods V., and Sargeant J., "Work distribution and scheduling in the Flagship parallel reduction machine", in Proc. of the Workshop on Architectural support for Declarative Programming, Eilat, Israel, pp.1-8, May 1989.

[100] Fairbairn J., and Wray S., "TIM: A simple, lazy abstract machine to execute supercombinators", in Proc. 3rd Conf. on Functional Programming and Computer Architecture, Portland, Orgeon, USA, LNCS Vol. 274, pp.34-45, Springer-Verlag, Sept. 1987.

[101] Raber M. et al, "Compiled graph reduction on a processor network", *Informatik-Berichte*, GI/ITG Tagung paderborn, 1988.

[102] Augustsson L., Nilsson C., and Truvé S., "The PG - machine: a machine for parallel graph reduction", Tech. Report Memo 57, Department of Comp. Sciences, Chalmers University of Technology, S-412 96 Göteborg, 1988.

[103] Burn G.L., "A shared memory parallel G-machine based on the evaluation transformer model of computation", in Proc. of the Workshop on the Implementation of Lazy Functional Languages, Programming Methodology Group, Göteborg, Sept. 1988.

[104] Augustsson L., and Johnsson T., "Parallel graph reduction with the <nu, G> machine", in Proc. IFIP Conf. on Functional Programming Languages and Computer Architectures, London, pp.202-213, Sept. 1989.

[105] Peyton Jones S.L., and Salkild J., "The spineless tagless G-machine", in *Functional Programming Languages and Computer Architecture*, by MacQueen D. (ed.), Addison-Wesley, pp.184-201, Sept. 1989.

[106] Revesz G.E., "Rule-based semantics for an extended lambda calculus", in Proc. 3rd Workshop on Mathematical Foundations of Programming Language Semantics, New Orleans, Louisiana, USA, LNCS vol.298, pp.43-56, Springer-Verlag, April 1987.

[107] Revesz G.E., "Axioms for the theory of lambda conversion", SIAM Journal on Computing, vol.14, no.2, pp.373-382, 1985.

[108] Revesz G.E., "An extension of lambda calculus for functional programming", The Journal of Logic Programming, vol.1, no.3, pp.241-251, 1984.

[109] Wadler P., "Efficient compilation of pattern-matching", in *The implementation of functional programming languages*, by S.L.Peyton Jones, PHI series in Computer Science, PHI(UK), 1987.

[110] Augustsson L., "Compiling pattern-matching", in Proc. IFIP Conf. on Functional Programming and Computer Architecture, Nancy, LNCS vol.201, pp.368-381, Springer-Verlag, 1985.

[111] Clack C., and Peyton Jones S.L., "Strictness analysis - a practical approach", in Proc. IFIP Conf. on Functional Programming and Computer Architecture, Nancy, LNCS vol.201, pp.35-49, Springer-Verlag, 1985.

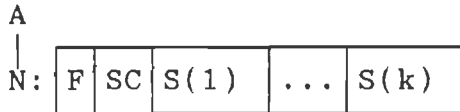
[112] Mycroft A., "Abstract interpretation and optimising transformations for applicative programs", PhD Thesis, University of Edinburgh, 1981.

[113] Burn G.L., "Abstract interpretation and the parallel evaluation of functional languages", PhD Thesis, Dept. of Computing, Imperial College of Science and Technology, University of London, 1987.

[114] Burn G.L., "Evaluation transformers - a model for the parallel evaluation of functional languages (extended abstract)", in Proc. 3rd Conf. on Functional Programming and Computer Architecture, Portland, Orgeon, USA, LNCS Vol. 274, pp.446-470, Springer Verlag, Sept. 1987.

[115] Bevan D.I., Burn G.L., Karia R.J., and Robson J.D., "Principles for the design of a distributed memory architecture for parallel graph reduction", The Computer Journal, vol.32, no.5, pp.461-469, 1989.

The appendix contains some examples that illustrate task reduction mechanism through snap-shots of task-graph at various stages of reduction. The figures used for task-graphs show a task as



where the A-field either explicitly shows a task name or an arc link to some task implies that the field contains name of the pointed task.

<u>**Example 1**</u> : The example illustrates the basic reduction strategy. The program is

```
---------------------------------------------
$f x = let y = SQ x in
       + (+ (* (- 9 4) y) y) (TAN (+ 22 23))
---------------------------------------------
$f 2
```

The definition of $f gets compiled into a D-code given as

```
D-code = (Header, SDS)
       = (($f,  1, 1, [], [], 0), (/@_2, (\@_2, +, (/@_2,  (\@_2,  +,
         (\@_2, (@_2, *, (@_2, (@_2, -, 4), 9)), ^y)), ^y)), (@_2, TAN,
         (@_2, (@_2, +, 22), 23)) :: [y:(\@_2, SQ, !■)]))
```

Organising the program expression, through *Org* procedure, yields the following task-graph:

nul

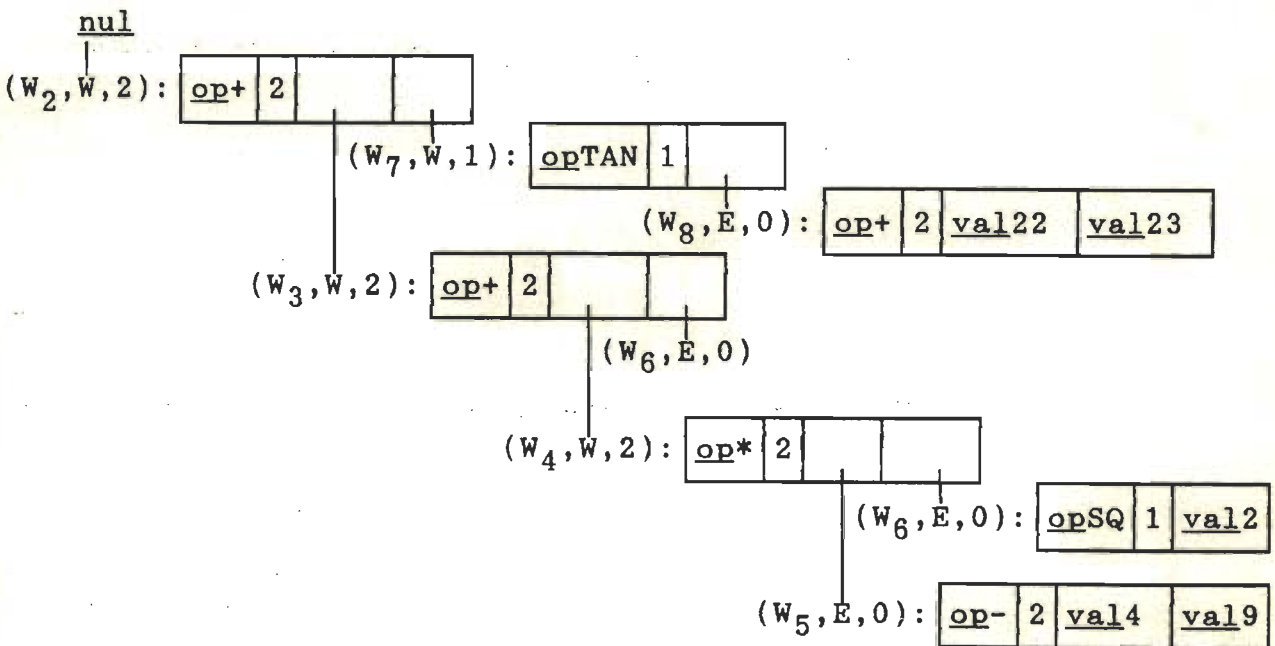
(W_1,E,0): | $f | 1 | val2 |

Task W_1 starts reducing, using the D-code of $f. Finding that the function $f involves no pattern matching (number of clauses = 1), the four step reduction cycle is initiated, using SDS term in the D-code as a template for instantiation. The reduction proceeds through following steps:

Step 1 : Application - a copy of the template is applied, following term reduction rules, to the argument (val2) resulting into the term

($@_2$, ($@_2$, +, ($@_2$, ($@_2$, +, ($@_2$, ($@_2$, *, ($@_2$, ($@_2$, -, 4), 9)), ^y)), ^y)), ($@_2$, TAN, ($@_2$, ($@_2$, +, 22), 23)) :: [y:($@_2$, SQ, val2)])

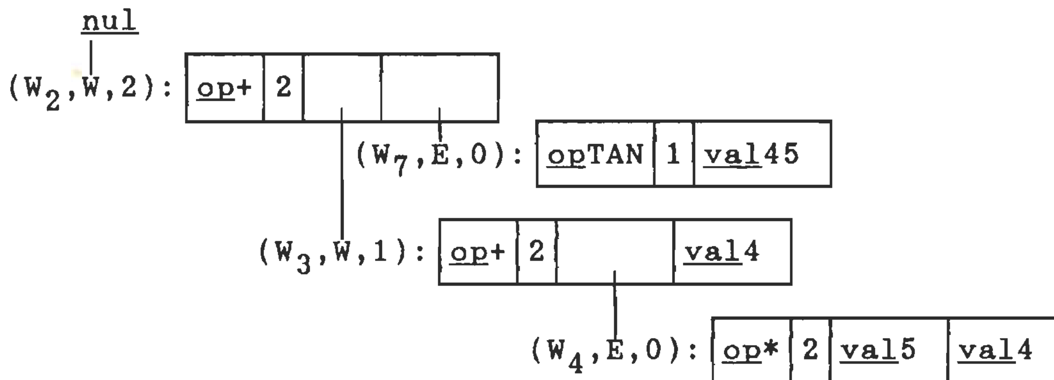
Step 2 : Organisation - the SDS term is now an exp_graph which is organised using *Org_result*. It gives the following task-graph:



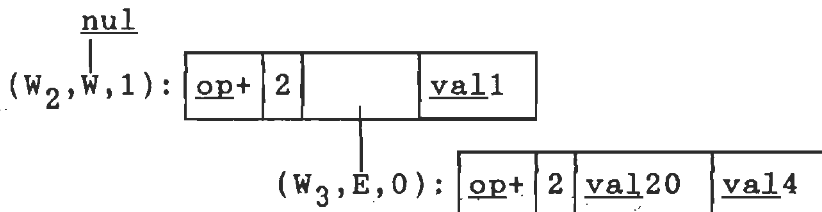
Step 3 : Communication - ancestor of the reducing task W_1 is nul hence no communication is required.

Step 4 : Removal - task W_1 kills itself.

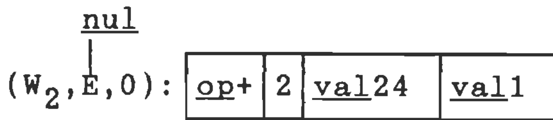
Now the dynamic graph is the one shown in step 2. Tasks W_5, W_6 and W_8 can execute in parallel. It may be noted that W_6 has two ancestors - (W_4,W,2;2) and (W_3,W,2;2). The three tasks reduce simultaneously (cases of primitive reduction) and communicate the results to their respective ancestor(s). The task-graph after these reductions becomes



Now W_4 and W_7 reduce in parallel. After these reductions the graph appears as



Reduction of W_3 at this stage transforms the task-graph into

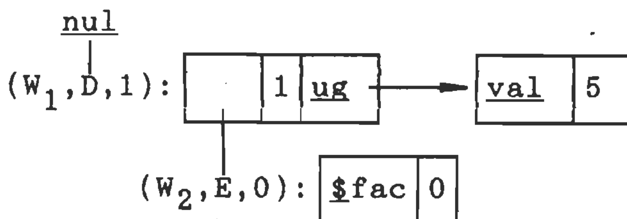


Finally task W_2 reduces to val25 so that the task-graph becomes a tagged_value lying at the root as

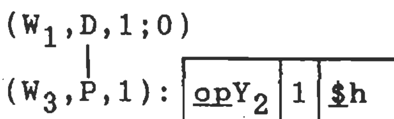
Example 2 : This example illustrates recursion in factorial function using concepts of partial tasks and Y operator. The program is

```
--------------------------------------
$h f n = if (= 0 n) 1 (* n f(- 1 n)
$fac = Y $h
--------------------------------------
$fac 5
```

The program expression gets organised into a dummy and an executable task as shown in the following task-graph:



Task W_2 reduces and the result of its reduction is a task given as



It prepares a link message M = (link, (W_3,P,1), 0) and communicates it to W_1. W_2 then kills itself. W_1, while handling the link message, generates an arg message M = (arg, (ug^val5), nul), communicates it to W_3 and kills itself. W_3, while handling the arg message, organises itself into the following task-graph:

```
          nul
           |
(W3,E,0): | opY2 | 2 | $h | val5 |
```

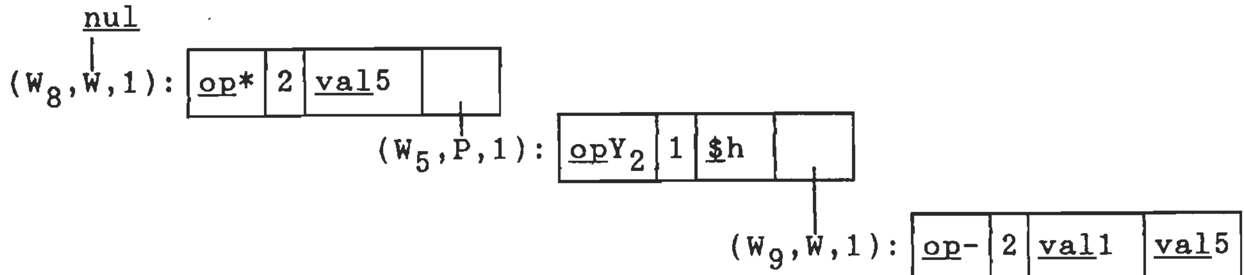
Now task W_3 undergoes a Y reduction resulting into the following task-graph:

```
          nul
           |
(W4,E,0): | $h | 2 |      | val 5 |
                       |
             (W5,P,1): | opY2 | 1 | $h |
```

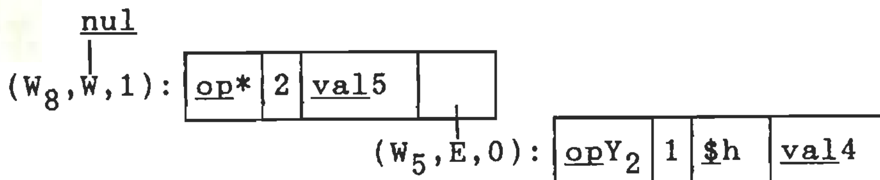
It may be noted that W_5 is a non-computable partial task and hence W_4 is executable. Reduction of W_4, while using the SDS term of $h, gives

```
                                  | val | 5 |
          nul                        ^
           |                         |
(W6,W,1): | opIF | 3 |       | ug | ug | ---> @ ------------- @
                         |                  /               / \
                         |                 @      (W5,P,1)   @
                         |                / \               / \
                         |               *  val5           @   va5
                         |                                / \
                         |                               -  val1
                         |
              (W7,E,0): | op= | 2 | val0 | val5 |
```

Task W_7 returns a result 'False' to W_6 through the message M = (result, valFalse, 1). On receipt of this message W_6 orders organisation of S(3) (through *Org_result*) so that the graph changes to



Task W_9 now reduces and communicates its result to W_5 through the message M = (result, val4, 2), and kills itself. After taking action on this message the task W_5 becomes reducible and the graph appears as



The situation of task W_5 now is similar to that of task (W_3,E,0) and the reduction proceeds through repetitions of reduction cycles described above.

Example 3 :This is the example of a program which computes the sum of first m integers. The example was taken up in chapter 6 for illustrating the compilation of supercombinator definitions with recursive local definitions into SDS terms. Here we proceed further to look at its execution. The program is

```
-----------------------------------------------------------------
$Count count m n = IF (> n m) NIL (CONS n (count (+ n 1)))
$Sum ns = IF (= ns NIL) 0 (+ (HD ns) ($Sum (TL ns)))
$Sumints m = letrec count = $Count count m in $Sum (count 1)
-----------------------------------------------------------------
$Sumints 100
```

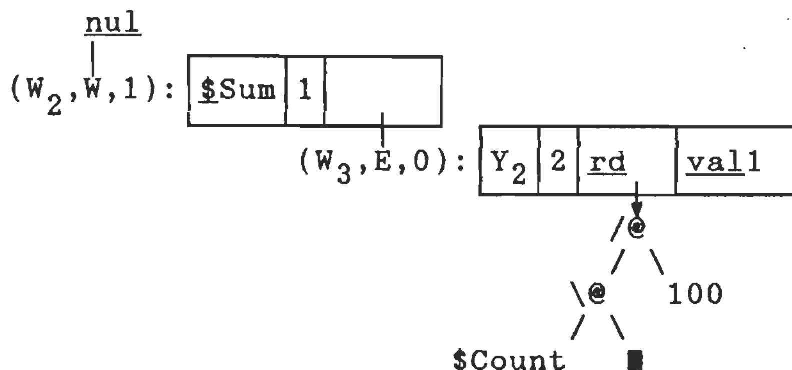
The program expression gets compiled through *Org* into a task-graph given as

```
  nul
   |
(W1,E,0): | $Sumints | 1 | val100 |
```

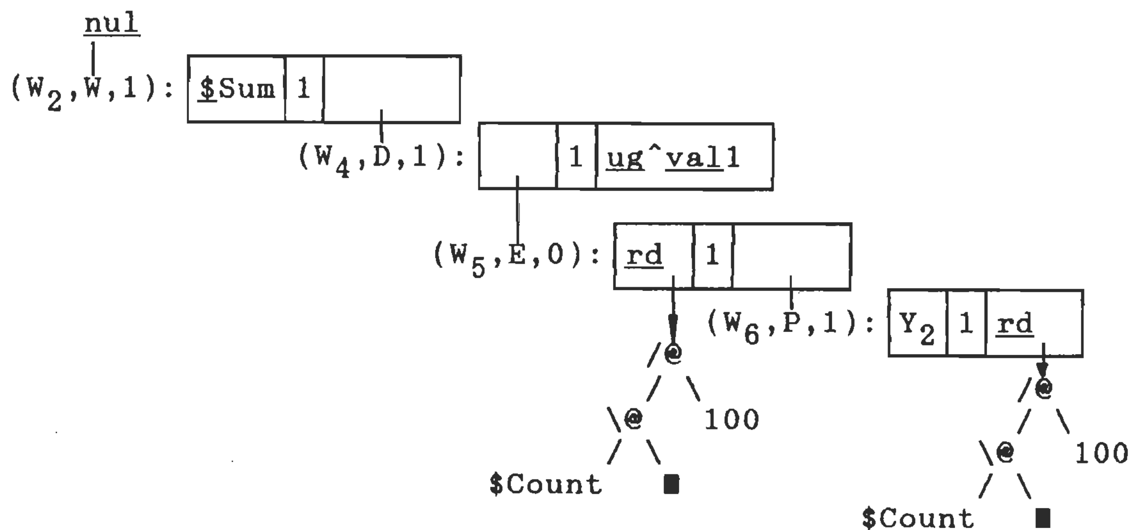
The only task in this graph is executable and it initiates a reduction cycle using a copy of the SDS term of $Sumints. During the first step of reduction cycle the SDS term becomes

```
(@2, $Sum, (@2, ^count, 1)) :: [count:(@2, Y, (/@2, (\@2,
                                        $Count, !■), 100))]
```

In step 2 the resulting SDS term is organised into a task-graph. Here the *Org_exp* module will call the *Org_Y* module while organising the expression in the C-list. The arity of Y will come out to be 2 because Y is applied to the expression (/$@_2$, (\$@_2$, $Count, !■), 100). In this expression, one argument is required due to a director at the root node and one more due to the function $Count at the tip being short of one argument, hence the total requirement of Y is 2. Thus the organisation in step 2 results into following task-graph:



The ancestor of W_1 is nul hence no communication is required, and the task kills itself. Now task W_3 can reduce because the Y operator in it has the necessary number of arguments. It stands for an expression (Y_2 (rd^) val1) which reduces to ((rd^) (Y_2 (rd^)) val1) by the reduction rule for Y. After this Y-reduction the task-graph will look as

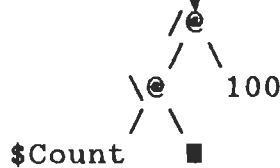


Now task W_5 reduces. It has a residual definition in its F-field which makes it executable according to the modified conditions of reducibility. Its only successor is a non-computable partial. The result of the reduction of W_5 is the task-graph

$(W_4,D,1;0)$

$(W_7,P,1)$: | <u>$</u>Count | 2 | | <u>val</u>100 |

$(W_6,P,1)$: | Y_2 | 1 | <u>rd</u> |



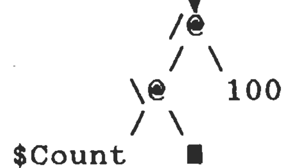
W_5 prepares a message M = (<u>link</u>, $(W_7,P,1)$, 0) and sends it to the ancestor W_4, who on receipt, prepares two messages, M_1 = (<u>arg</u>, (<u>ug</u>^<u>val</u>1), $(W_2,W,1;1)$) for W_7 and M_2 = (<u>link</u>, $(W_7,P,0)$, 1) for the ancestor W_2. Task W_4 then kills itself. The complete task-graph after exchange of these messages and corresponding actions becomes

<u>nul</u>

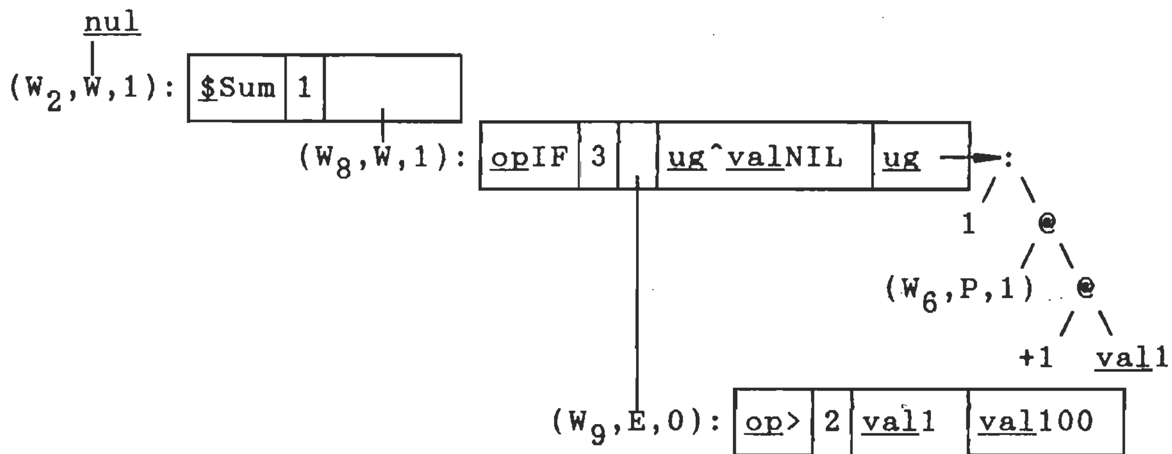
$(W_2,W,1)$: | <u>$</u>Sum | 1 | |

$(W_7,E,0)$: | <u>$</u>Count | 3 | | <u>val</u>100 | <u>val</u>1 |

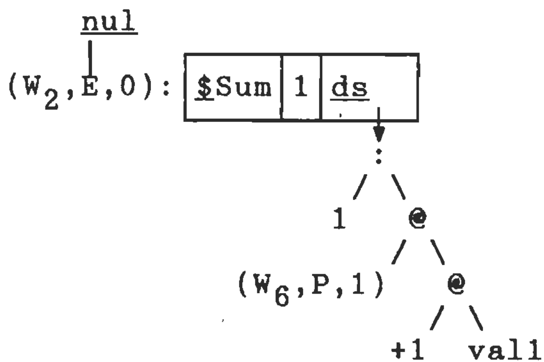
$(W_6,P,1)$: | Y_2 | 1 | <u>rd</u> |



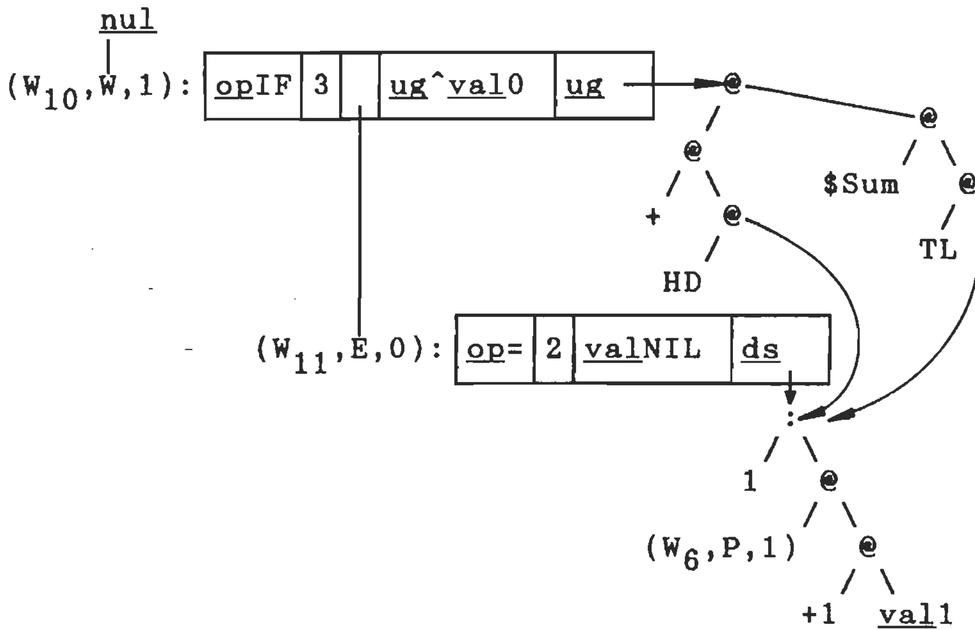
Now task W_7 finding a non-computable partial at S(1) position, goes ahead with reduction, using the SDS term of $Count as template. The graph then becomes



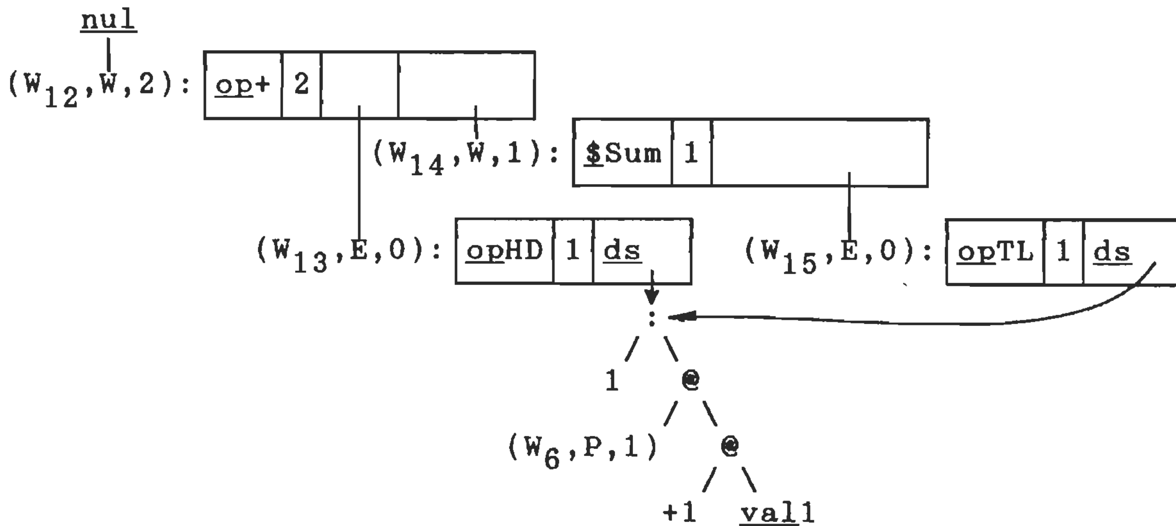
Task W_9 sends a result message containing the result 'False' to its ancestor and kills itself. As a result, task W_8 organises its S(3) and links to its ancestor. The graph then becomes



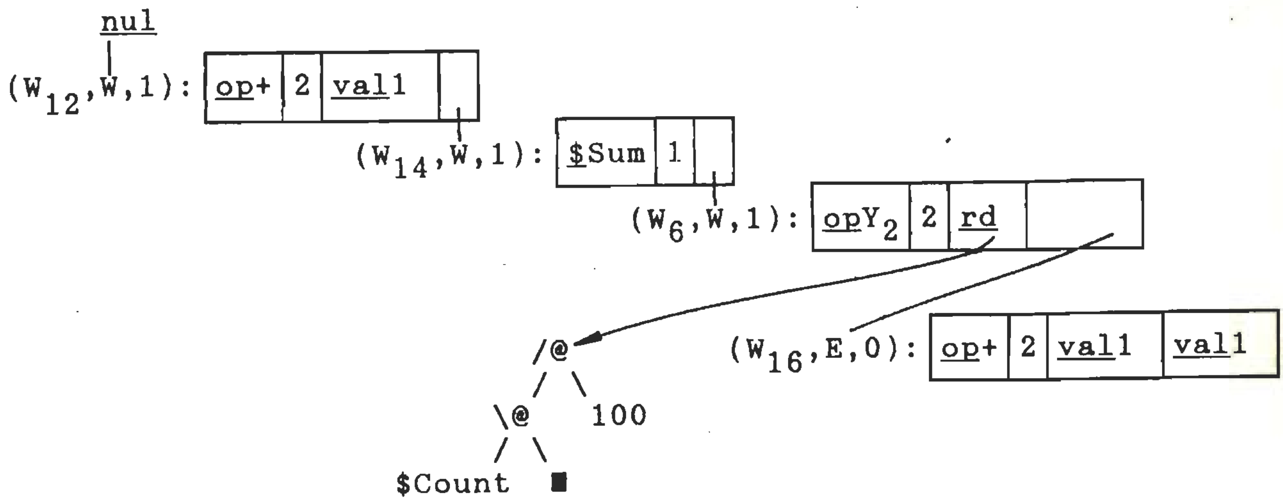
Task W_2 has now become reducible. Its reduction uses the SDS term of $Sum. After reduction the graph will appear as



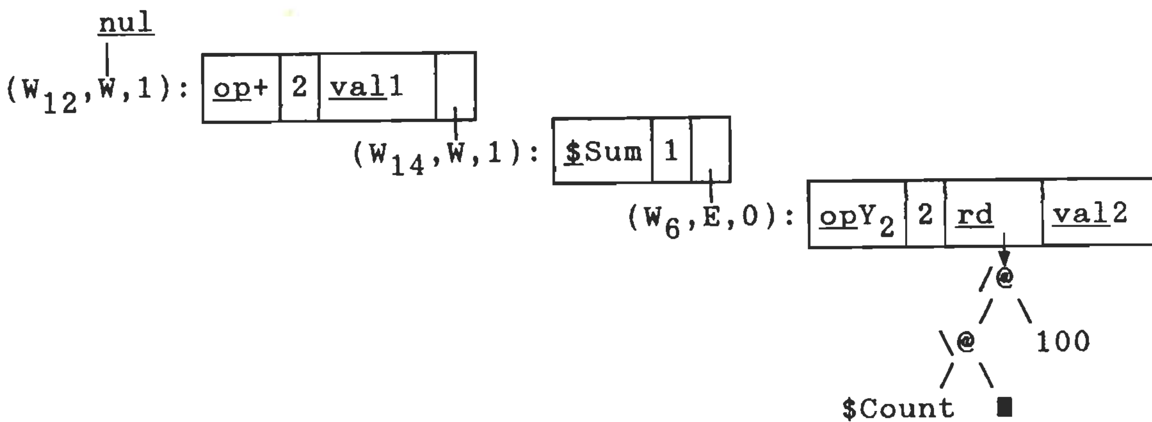
Task W_{11} returns a result 'False' to its ancestor who then organises its S(3) giving it the ancestor nul. The graph then becomes



Tasks W_{13} and W_{15} reduce simultaneously producing the graph



The partial task W_6 got organised into a complete task through *Org-result*. Now task W_{16} reduces and the graph takes the shape



At this stage, the recursion has run through one cycle. From here onwards the reduction will be repetition of the previous reductions. Task W_{14} is in the same situation as was the task W_2 at the time of its creation in the first snapshot of the task-graph.

RESEARCH PAPERS OUT OF THE WORK

I. Communicated

1. "Compiling Pattern-Matching Definitions into Structured Director String Terms", Communicated to *IEEE Trans. Computers*, April 1990.

2. "Supercombinator Compilation for Multiprocessor Architecture Implementation", Communicated to *IEEE Trans. Software Engineering*, July 1989.

II. Accepted/Published

3. "CTDNet II - A Coarse Grain Multiprocessor Architecture for Functional Programs", accepted for presentation in *UNESCO Conference on Parallel Computing in Engineering and Engineering Education*, to be held at Paris, Oct. 8-12, 1990.

4. "Implementing Pattern Matching Definitions in CTDNet-A Multiprocessor Architecture", *Proc. Fifteenth Euromicro Symposium on Microprocessing & Microprogramming*, Cologne, West Germany, pp. 151-155, Sept. 1989.

5. "Implementation of Recursion in CTDNet - An Applicative Order Machine", *Proc. Second Int. Conf. on Software Engineering for Real-time Systems*, Cirencester, U.K., pp. 224-227, Sept. 1989.