Chapter 2

REVIEW OF FUNCTIONAL PROGRAMMING

2.1 INTRODUCTION

Advocates of functional programming contend that the imperative programming languages (e.g. FORTRAN, Pascal) are simply 'high-level versions' of the von-Neumann machine [6]. Principal operations, in the imperative languages, involve changing the state of computation in much the same way a machine language program does. 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. The underlying computational model (von-Neumann) has been playing a steering role in the language design technology and thus becoming a constraint on the language development.

The concept of functional languages has developed more or less independently of an underlying computational model. Realising their importance as a highly expressive and elegant programming medium, the research on designing suitable models for them (or adopting 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.
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

Three main models that are being considered for implementation of functional languages are: Control Flow, Data flow and Reduction [68,69]. 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 GO TO). In control flow instructions are active agents that manipulate passive data. Data flow [27], 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 directed flow graphs, 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. When the data token all the input arcs are arrived, a node gets fired (Executed) and a result data token is placed on its output arc. As the sequencing of firing is based on data dependencies only, a greater degree of parallelism can be supported on this model when compared to control flow. Several data flow architectures are operational/under development [29,56,27].

The Reduction model makes no distinction between program and data, both instructions and operands are expressions, and hence it has no concept of a flow of control either instruction based (control flow) or data based (data flow). Reduction
model is based on mathematical reasoning where program execution goes like simplifications of mathematical expressions. As more than one sub-expression can be simplified in functional framework, this model supports parallelism naturally.

Considering various implications of the three models. Treleaven et al [69] have concluded: control flow lacks useful mathematical properties for reasoning about programs, and parallelism is alien to its concept; data flow permits highly parallel implementation, but its utility as a general purpose program organisation is questionable but is more suited to specialist applications; reduction appears to be the most natural candidate for providing efficient support to functional programming. Graph reduction, a new technique first invented by wades worth [77] is inherently a parallel activity. It supports simultaneous and asynchronous evaluations at several sites within the graph. 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 confined to the developments in reduction based machine models only. However, a brief introduction to the functional programming and lambda calculus is first presented in the following section.
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

2.2 FUNCTIONAL PROGRAMMING AND LAMBDA CALCULUS

Imperative languages are characterised as having an implicit 'state' that is modified (i.e. side effected) by constructs (i.e. commands) in the source language. As a result, such languages generally have a notion of 'sequencing' (of the commands) to permit precise and deterministic control over the state. The roles played by sequencing and assignment in these languages make them essentially command oriented. That is, at some level, programs are sequences of commands to be obeyed. In this respect they resemble the underlying machine on which they run.

In contrast, functional languages are characterized as having no implicit state, and thus the emphasis is placed entirely on programming with expressions (or terms). These languages have more expressive power [73], and their use can increase programmer productivity by allowing him to concentrate more on algorithmic details rather than worrying about low level details such as keeping track of variables through the various constructs that cause side effects. Programs in this class of languages are side effect free and thus preserve the property of referential transparency, which means that 'equals can be replaced by equals'. The notion of referential transparency allows clean equational reasoning which is very powerful, not only for reasoning about programs but also informally in writing and debugging programs.
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

The first functional language was LISP developed by McCarthy as a formalism for reasoning about recursion equations as a model of computation [51]. However, later dialects of LISP are imperative as they include both sequencing and assignment. Some of the functional languages that have been developed are FP [7], SASL [91], HOPE [24], KRC [73], LML [3], Haskell [38] etc.

2.2.1 Functional Programming [13,49]

Functional languages are based on the single concept of function application and thus they are also known as applicative languages. A program is made up of a set of definitions and an expression. An expression states the intention of program while definitions act as rewrite rules for evaluating the expression. Application is expressed by juxtaposition e.g. the expression (f 3) denotes the application of a function f to the argument 3. It has left associative 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. A function is a kind of program which accepts input (in the form of arguments) and produces output (the value of the function call). The concept of 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 (arguments) within brackets, e.g.

\[ F(x) = (x \times x) + 3 \]
\[ \text{Max}(a,b) = \text{if } a > b \text{ then } a \text{ else } b \]

In functional notation, \( f(x) \) is interpreted as \( f \) applied to \( x \), and therefore the brackets enclosing the variable are dropped. However, with functions of more than one variable such as \( \times \) or \( \text{max} \) in Eq. 2.1 an interpretation problem arises. Currying, a method introduced by Schonfinkel [67] and extensively used by Curry [21], is used to resolve it. The method represents all multi-argument functions as sequences of unary ones. To express 'the sum of 3 and 4' we write \(((+ 3) 4)\). The expression \((+ 3)\) denotes a function that adds 3 to its argument. Thus the whole expression means 'the function + applied to the argument 3. The result of which is a function applied to 4'. Dropping the parentheses, it can also be written as \((+3 4)\) to mean the same thing. Based on the above, Eq. 2.1 is rewritten as

\[ F(x) = (+(* x x)) 3 \]
\[ \text{Max}(a, b) = \text{IF}( > a b) a b \]

The if-then-else construct in the definition max has been replaced by a curried IF operator which is a primitive function whose semantics is given by

\[ \text{IF True } a b = a \]
\[ \text{IF False } a b = b \]
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

Expressions are built through a hierarchy of functions where more complicated ones are built from the earlier defined ones. For example, a function for finding the maximum of three numbers can be built on the definition of function max as

\[ \text{Max} \ a \ b \ c = \text{max} \ ( \text{max} \ a \ b ) \ c \]

An expression \( (\text{max}3 \ 5 \ 3 \ 9) \) is evaluated using definitions of max and max3 through the following rewrites:

\[
\begin{align*}
\text{Max} \ 3 \ 5 \ 3 \ 9 & \rightarrow \text{max} \ (\text{max} \ 5 \ 3) \ 9 \\
& \rightarrow \text{max} \ (\text{IF} \ (>5 \ 3) \ 5 \ 3) \ 9 \\
& \rightarrow \text{max} \ (\text{IF} \ \text{True} \ 5 \ 3) \ 9 \\
& \rightarrow \text{max} \ 5 \ 9 \\
& \rightarrow \text{IF} \ (>5 \ 9) \ 5 \ 9 \\
& \rightarrow \text{IF} \ \text{False} \ 5 \ 9 \\
& \rightarrow 9
\end{align*}
\]

**Recursive Functions**

In imperative languages repeated computations are expressed in a program by making use of some sort of loop constructs (such as for or while). For example, to compute the sum of first n positive integers a Pascal function may be written as
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

Function sum (n:integer): integer;

var

loopcounter, acc: integer;

begin

acc := 0;

for loopcounter = 1 to n do

acc = acc + loopcounter;

sum = acc

end:

In a functional language there are no looping constructs and there is certainly no destructive assignment facility which could update the value of a counter. Here, the problem is solved using recursive function definition. In a recursive definition, a function invokes itself on the right-hand side. The functional version of the above problem can be written as

\[
\text{Sum } n = \text{IF} (= n 0) \circ (+ n \text{sum} (- n 1))
\]

Recursive functions are extensively used in functional languages. KRC [70] is purely a functional language based on recursion equations. Although recursion leads to very abstract and concise solutions to numerical problems, its real power lies in the treatment of functions which operate on abstract data types.
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

Constructor Functions

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, LEAF and BRANCH are constructors for tree type structures. Using these constructors an expression which denotes the single-element list containing 3 can be written as

\[
\text{CONS}(3, \text{NIL})
\]

Where NIL is also a data constructor which stands for an empty list. Unlike other functions, however, constructor functions have no rules. The expression CONS(3.NIL) is therefore a value in that it cannot be further simplified by using any function rule. Here, constructor functions serve only to construct, i.e. bind together, data.

A list containing the numbers 1, 2 and 3 is expressed as

\[
\text{CONS}(1, \text{CONS}(2, \text{CONS}(3, \text{NIL})))
\]

Data structures can appear as parameters to a function e.g. a function sum returning the sum of a list is defined as

\[
\text{Sum NIL} = 0
\]

\[
\text{Sum}(\text{x : xs}) = +\text{x} (\text{sum}\text{xs})
\]

It can be seen that the sum is recursively invoked until the empty list is encountered. 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 the next chapter.

**High–order Functions**

Functional languages draw no distinction between functions or data. Functions can be made elements of structures, passed as parameters and returned as results. Functions which take other functions as parameters or return another function as a result are called higher-order functions. Turner [73] 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

\[
\text{map } f \text{ NIL} = \text{NIL} \\
\text{map } f \text{ (x: xs)} = (f \text{ x}) : (\text{map } f \text{ xs})
\]

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 the original list. Higher-order functions can be used to bring about an extremely compressed programming style. Consider, for example, the definition of `sum` given earlier. This represents an extremely common pattern of recursion for folding a list using a given binary operator (in this case `+`) and a given starting value (in this case `0`). The general pattern can be captured in the following definition of a 3-argument function, `fold`

\[
\text{Fold } op \text{ s } \text{NIL} = \text{s} \\
\text{Fold } op \text{ s } (x : xs) = op \text{ x} (\text{fold } op \text{ s } xs)
\]

Now the sum can be defined in a single line, just by partially parameterising `fold` as

\[
\text{Sum} = \text{fold } plus \text{ 0}
\]
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

The advantage is that a greater number of analogous functions become available without any further effort, for example

Product = fold times 1

J. Hughes in his paper [40] has demonstrated that the functional languages provide two new kinds of glue: higher-order functions and lazy evaluation. Using these glues one can modularise programs in new and exciting ways.

All functional languages derive their semantic base in the lambda calculus developed by Alonzo Church [20]. Lambda calculus is a simple and expressive language, which is sufficiently powerful to express all functional programs. Ability to translate a higher-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.

2.2.2. Lambda Calculus

The lambda calculus [25] is a calculus of functions, and was developed by Church in 1930’s for dealing with some basic issues in computability theory [20], in particular to define precisely the intuitive notation of which functions of the positive integers could be computed in a mechanical or algorithmic way. Church proposed that these effectively calculable functions should be identified with those functions that are expressible in the lambda calculus.

It is clear that the value of an expression such as \( E = x \times x + 3 \) depends on the value of \( x \), and that this expression defines a function \( f \), commonly written as
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

\[ F(x) = x \times x + 3 \]

This notation is ambiguous: does \( f(x_1) \) denote a function of \( x_1 \) or the value of \( f(x) \) at \( x = x_1 \)? normally, the context will provide the correct interpretation. In the lambda calculus these ambiguities do not arise. The operation of forming a function definition from a simple expression is called lambda abstraction, and the definition of \( f(x) \) given above corresponds to the abstraction expression:

\[ F = \lambda x. E = \lambda x. + (* x x) 3 \]

Here, \( \lambda \), \( x \) and \( E \) are known as the abstraction operator, bound variable and body of the expression respectively. Function \( f \) itself is independent of \( x \): its value is a function, which when applied to an argument yields a result whose value may be non-functional (e.g. a constant), or another function. It is denoted by an applicative expression as

\[ (\lambda x. E) M \]

Where \( M \) is the argument to which the abstraction expression \( (\lambda x. E) \) is applied.

A variable can occur in an expression in two ways: bound or free (57). The bound variable is the one involved in abstraction, as described above. All other variables are free. For example, in the abstraction expression

\[ \lambda x. + (*x x) (*x y) \]

\( x \) and \( y \) are free in the expression \( ( + (*x x) (*x y)) \): \( x \) is bound and \( y \) is free in the expression \( (\lambda x. + (*x x) (*x y)) \).
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

The essential syntax of the lambda calculus is very concise and is summarised in the following productions (31)

\[ E : = \text{constant} \mid \text{built-in constant} \]

\[ \text{Variable} \mid \text{variable names} \]

\[ E_1 E_2 \mid \text{application} \]

\[ (E) \mid \text{bracketing} \]

\[ \lambda \text{variable}.E \mid \text{abstraction} \]

... (2.4)

Lambda expressions having syntax governed by Eq. 2.4 are well formed expressions. 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.

The rule of maximum importance is \(\beta\)-rule which deals with the application of a lambda abstraction to an argument. It is formally defined as

\[(\lambda x.E) M \rightarrow E(M/x)\]

Where \(E(M/x)\) denotes that every free occurrence of \(x\) in expression \(E\) is replaced by the expression \(M\). Thus the result of applying the abstract in Eq. 2.3 to an argument 5 is obtained as follows:

\[(\lambda x.+(x\ x\ )\ 3)\ 5 \rightarrow (+\ 5\ 5\ )\ 3 \rightarrow +\ 28\]

by \(\beta\)-rule

(by operator rules)

Application of \(\beta\)-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. An application in which β-rule is applicable is called reducible expression, or redex.

There are two other conversion rules in the lambda calculus: αis concerned with changing the name of a bound variable, and ηis for removing redundant lambda abstractions. These are formally defined as

α-rule: \( \lambda x.E \quad \lambda y.E \ [y/x] \)  
y is not free in E

η-rule: \( (\lambda x.E \ x) \quad E \)  
x is not free in E

and E is a function

α-rule is essentially a variable-name change rule. It says that expression remains unchanged if a variable in it is changed consistently. This rule is basically used to avoid name clashes [57]. η-rule is a kind of optimisation because it removes redundant abstractions. These rules when used from left to right, are called reductions (instead of conversions) and are written with a ‘\( \rightarrow \)’ symbol and are called abstractions when used from right to left and are written with a ‘\( \leftrightarrow \)’ symbol in place of ‘\( \leftrightarrow \)’.

Reduction order

Evaluation of an expression consists of successively reducing redexes until the expression contains no redexes. An expression with no redex is said to be in normal form.

However, an expression may contain more than one redex, so reduction can proceed by alternative routes. But, all the reduction sequences do not necessarily reach
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

the normal form if one exists. For example, the expression \((\lambda x.x) (\lambda x.x)\) has no normal form, and the expression

\[(\lambda x.5) ((\lambda x.x) (\lambda x.x))\]

has a normal form '5' which can be reached if the left-most redex is evaluated first. And the other hand if we first reduce the sub-expression \((\lambda x.x) (\lambda x.x)\), the evaluation fails to terminate. This shows that the order of reduction affects the outcome of an evaluation. However, two theorems, called – Church-Reosser Theorems (CRT) I and II [20], are highly reassuring in this state of affairs.

**THEOREM I:** If \(E_1 \leftrightarrow E_2\), then there exists an expression \(E\), such that \(E_1 \rightarrow E\) and \(E_2 \rightarrow E\).

An important corollary of this theorem says that no expression can be converted to two, distinct normal forms. Thus all reduction sequences will reach the same result provided they terminate. The theorem assures that any reduction sequence can be sued without the risk of reaching the wrong result, but a particular sequence may not terminate.

**THEOREM II:** If \(E_1 \rightarrow E_2\) 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 atmost one possible result and normal order must reach it. The proofs of the theorems can be seen in [32.20].

In normal order reduction, the outer-most, left-most beta redex is always selected next. The scheme is strictly sequential, and has call-by-need (also known as lazy)
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

semantics, since the rule ensures that the argument is never evaluated before being passed to the body of the function. Another reduction sequence is applicative order, where the argument is never evaluated before being passed to the body of the function.

Another reduction sequence is applicative order, where the argument is reduced to normal form before being passed to the function. An expression in normal form is typically an abstraction expression, in which no further beta reductions are possible, or a non-functional value in its most primitive form. Thus, the argument eventually passed to the function is in irreducible form. Applicative reduction has call-by-value (also known as eager) semantics. There is no constraint of sequentiality imposed on an applicative evaluator. i.e., all eligible redexes may be reduced in parallel.

Although normal order reduction is 'safer' than applicative order reduction it is often the case that a normal order reduction is much slower than the corresponding applicative reduction. The reason is not hard to find: in applicative reduction the argument of an application is reduced to normal form once and once only before substitution for the bound variable. However, the argument is substituted (may be many times) into the operator of an application in its unreduced form in a normal order reduction. Therefore it may be necessary to reduce it many times during the subsequent stages of the reduction.

Recursion

In a high level language recursion is expressed by a circular definition with the help of function names. But, in lambda calculus there is no notation of naming functions,
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

a lambda abstraction itself is regarded as a default name. It would seem impossible to achieve the circularity by means of ordinary name references in the lambda calculus. It is surprising that recursion can be expressed indirectly without explicit reference to the idea of naming. Recursion, in lambda calculus, is handled through ‘Y’ combinator, known as 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

\[ YH \rightarrow H (Y H) \]

This property of Y expresses recursion in its purest form, since it can be used to express all other recursive functions. Implementations based on lambda calculus use Y as a built-in operator with the 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 lax9iness of normal order is of great help.

2.3 Haskell

Haskell is a generarl-purpose, purely functional programming language exhibiting many of the recent innovations in functional (as well as other) programming language research, including higher order functions, lazy evaluation, static polymorphic typing, user-defined datatypes, pattern matching, and ilist comprehensions. It is also a very complete language in that it has a module facility, a well-defined functional I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

integers, and floating-point numbers. In this sense Haskell represents both the culmination and solidification of many years of research on functional languages— the design was influenced by languages as old as Iswim and as new as Miranda.

Haskell also has several interesting new features; most notably, a systematic treatment of overloading, an orthogonal abstract datatype facility, a universal and purely functional I/O system, and by analogy to list comprehensions, a notation of array comprehensions.

Haskell is not a small language. The decision to emphasize certain features such as pattern matching and user-defined datatypes and the desire for a complete and practical language that includes such things as I/O and modules necessitates a somewhat large design.

2.3.1 Formal Semantics

Denotational semantics and functional programming have close connections, and the functional programming community emphasizes the importance of formal semantics in general. For completeness and to show how simple the denotational semantics of a functional language can be given as]

\[
\text{Bas} = \text{Int} + \text{Bool} + \ldots \quad \text{Basic values}
\]

\[
\text{D} = \text{Bas} + (\text{D} \rightarrow \text{D}) \quad \text{Denotable values}
\]

\[
\text{Env} = \text{id} \rightarrow \text{D} \quad \text{Environments}
\]

\[
\text{E: Exp} \rightarrow \text{Env} \rightarrow \text{D}
\]
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

K: Con → D

E[[x]]env = env[[x]]

E[[c]]env = K[[c]]

E[[e_1, e_2]]env = (E[[e_1]]env)(E[[e_2]]env)

E[[\lambda x.e]]env = (\lambda v.E[[e]]env[v/x])

E[[e where x_1 = e_1; \ldots; x_n = e_n]]env

= E[[e]]env

where

env' = fix \lambda env'.env((E[[e_1]]env')/x_1,

\ldots,

(E[[e_n]]env'/x_n)

This semantics is relatively simple, but in moving to a more complex languages such as Haskell the semantics can become significantly more complex due to the many syntactic features that make the languages convenient to use.

2.4 Reduction Machine

Machines that simplify an expression, continually until it is in the simplest possible form (normal form) according to certain reduction rules are termed as reduction machines. A reduction machine takes a program in high-level language as input and gives the value of the program as output. 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 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 suitable to the machine, and which has some simple rules for reduction. A reduction computer, in its simplest form, can be represented as shown in Fig. 2.1.

![Figure 2.1 Representation of Reduction Computer.](image)

reduction machines. Lambda calculus [63] is one such intermediate language in which \( \beta \)-conversion is the principal reduction rule and forms the basis for many others. The lambda calculus can represent, through its simple syntax and semantics, all the features of a high-level functional language.
2.4.1 Lambda Calculus as Intermediate Language.

A lambda expression can be represented in the form of a tree which reflects its syntactical structure. The leaves of the trees are built-in operators, constants or variables names. The application of a function to an argument is represented as

```
@  
\ /  
f  a
```

where ‘@’ indicates an application node type. Functions of several arguments, dealt through currying, are shown through left associative application resulting in a left linear binary tree such as

```
@  
 /  
a2  
@  
/  
f  a1
```

A function in the lambda notation is of the form \( \lambda \) (formals), body, and its application to actual parameters is reduced through \( \beta \)-reductions expressed as

\[
(\lambda \text{ (formals), body}) \text{ actuals} \quad \overset{\beta}{\longrightarrow} \quad \text{body (with actuals substituted for formals)}
\]

A \( \beta \)-reduction constructs a new copy of the lambda body where free occurrences of a bound variable are replaced by the actual argument. This process is called instantiation.

There are two major techniques for substituting the argument into the body of the function:
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

- String reduction, in which complete copy of the argument is substituted into the function body for every occurrence of the bound variable.

- Graph reduction, in which a pointer to the argument is substituted into the function body for every occurrence of the bound variable.

In string reduction each copy of the argument graph is reduced separately. Since the argument may contain reducible expressions, which maybe large expressions, this approach may introduce unnecessary copying and also duplication of work in recomputing the same result. The performance of this 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 (10) 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 [50] cellular machine, implementing Backus’ FP [6] class of languages, is based on distributed string reduction and relies on massive parallelism to overcome its inefficiency. Berkling has done excessive work on string reduction systems based on applicative λ-expressions [9].

In graph reduction, pointers to the argument are substituted and hence any reductions within the argument are performed once only, and benefit all recipients of the result of the argument. Using normal order with graph reduction, Wadsworth [77] combined the benefits of the normal order and applicative order yielding a reduction system called normal order graph reduction or lazy evaluation [33].
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

The term substitution is not always meant literally (in the sense of textual substitution): In some reduction schemes for instance, the address of the argument is associated with the bound variable during β-reduction, and stored in the environment. Subsequently, the function may look up this address in the environment. However, some parallel schemes e.g. (CTDNet [28], favour a literal substitution mechanism.

2.4.2 Other Forms of Intermediate Language

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 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 needlessly.

A solution to the above problem is to compile each lambda body into a fixed sequence of instructions which, when executed, builds an instance.

2.5 Parallel Functional Programming

An often-heralded advantage of functional languages is that parallelism in a functional program is implicit; it is manifested solely through data dependencies and the semantics to more conventional languages, where explicit constructs are typically used to invoke, synchronize, and in general coordinate the concurrent activities. In fact, as
discussed earlier, many functional languages were developed simultaneously with work on high parallel data flow and reduction machines, and such research continues today.

In most of this work, parallelism in a functional program is detected by the system and allocated to processors automatically. Although in certain constrained classes of functional languages the mapping of process to processor can be determined optionally in the general case the optimal strategy is undecidable, so heuristics such as load balancing are often used instead. But what if a programmer knows a good (perhaps optimal) mapping strategy for a program executing on a particular machine, but the compiler is not smart enough to determine it? And even if the compiler is smart enough, how does one reason about such behaviour? We could argue that the programmer should not be concerned about such details, but that is a difficult argument to make to someone whose job is precisely to invent such algorithms.

The fundamental idea behind process-to-processor mapping is quite simple. Consider the expression e1+e2. The strict semantics of + allows the subexpressions e1 and e2 to be executed in parallel- this is an example of what is meant by saying that the parallelism in a functional program is implicit. But suppose now that we wish to express precisely where (i.e., on which processor) the subexpressions are to be evaluated; we may do so quite simply by annotating the subexpressions which appropriate mapping information. An expression annotated in this way is called a mapped expression, which has the following form:

X.Exp on proc
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

Which intuitively declares that exp is to be computed on the processor identified by proc.
The expression exp is the body of the mapped expression and represents the value to which the overall expression will evaluate (and thus can be any of expression, including another mapped expression). The expression proc must evaluate to a processor id. Without loss of generality the processor ids, or pids, are assumed to be integers, and there is some predefined mapping from those integers to the physical processors they denote. Returning now to the example, we can annotate the expression (e1+e2) as follows:

\[(e1 \text{ on } 0) + (e2 \text{ on } 1)\]

Where 0 and 1 are processor ids. Of course, this static mapping is not very interesting. It would be nice, for example, if we were able to refer to a processor relative to the currently executing one. We can do this through the use of the reserved identifier self, which when evaluated returns the pid of the currently executing processor. Using self we can now be more creative. For example, suppose where a ring of n processors that are numbered consecutively; we may then rewrite the above expression as

\[(e1 \text{ on left self}) + (e2 \text{ on right self})\]

Where left pid = \(\text{mod} \ (\text{pid} - 1)n\)

Right pid = \(\text{mod} \ (\text{pid} + 1)n\)

[\(\text{mod} \ x \ y \text{ computes } x \text{ modulo } y\), which denote the computation of the two subexpressions in parallel on the two neighboring processors, with the sum being computed on self.}

34
Chapter 2. REVIEW OF FUNCTIONAL PROGRAMMING

To see that it is desirable to bind self dynamically, consider that one may wish successive invocations of a recursive call to be executed on different processors—this is not easily expressed with lexically bound annotations. For example, consider the following list-of-factorials program, again using a ring of processors:

\[
(map \text{fac}[2,3,4]) \text{ on } 0
\]

where \(map f[] = []\)

\[
f(x:xs) = f x: ((map f xs) \text{ on } (\text{right self}))
\]

Note that the recursive call to map is mapped onto the processor to the right of the current one, and thus the elements 2, 6 and 24 in the result list are computed on processors 0, 1, and 2, respectively.

Para-functional programming languages have been shown to be adequate in expressing a wide range of deterministic parallel algorithms clearly and concisely. It remains to be seen, however, whether the pragmatic concerns that motivate these kinds of language extensions persist, and if they do whether or not compilers can become smart enough to perform the optimizations automatically.