An Investigation of Nondeterminism in Functional Programming Languages

THESIS

Submitted in fulfilment of the requirements

for the Degree of

MASTER OF SCIENCE

of Rhodes University

by

Gwyneth Clare Graham

February 1997

Abstract

This thesis investigates nondeterminism in functional programming languages. To establish a precise understanding of nondeterministic language properties, Sondergaard and Sestoft's analysis and definitions of functional language properties are adopted as are the characterizations of weak and strong nondeterminism.

This groundwork is followed by a denotational semantic description of a nondeterministic language (suggested by Sondergaard and Sestoft). In this manner, a precise characterization of the effects of strong nondeterminism is developed.

Methods used to hide nondeterminism to in order to overcome or sidestep the problem of strong nondeterminism in pure functional languages are defined. These different techniques ensure that functional languages remain pure but also include some of the advantages of nondeterminism.

Lastly, this discussion of nondeterminism is applied to the area of functional parallel language implementation to indicate that the related problem and the possible solutions are not purely academic. This application gives rise to an interesting discussion on optimization of list parallelism. This technique relies on the ability to decide when a bag may be used instead of a list.

Acknowledgments

Throughout this project my supervisor, Professor E.P. Wentworth, has provided me with valuable advice and support. His insight into a wide variety of topics resulted in stimulating conversations and debates and encouraged me to develop a keen interest in functional programming. He deserves special thanks for reading the draft copies of this thesis.

Graham Smith, who often acted as a sounding board, provided valuable peer interaction and discussion. Without the technical support provided by Billy Morgan, Shaun Bangay, Graham Smith and Greg Watkins, the type setting of this thesis would have been an insurmountable obstacle.

I would like to thank my friends, family and the members of the Rhodes University Computer Science Department for their continuous encouragement and support.

Contents

1 Introduction **1**

1.1 Functional Programming Languages 2

1.1.1 Structure 2

1.1.2 Properties 3

1.1.3 Church-Rosser Theorem and Confluence 4

1.1.4 Pure and Impure Functional Languages 5

1.2 Introduction to Nondeterminism 7

1.2.1 Strong and Weak Nondeterminism 8

1.2.2 Effects of Nondeterminism 8

1.3 Project Goals 9

2 Nondeterministic Constructs and Strong Nondeterminism **11**

2.1 Types of Strong Nondeterminism 11

2.1.1 Angelic Nondeterminism 12

2.1.2 Demonic Nondeterminism 12

2.1.3 Erratic Nondeterminism 13

2.1.4 Global versus Local Nondeterminism 13

2.2 Effect on Language Properties 14

2.2.1 Referential Transparency 15

2.2.2 Definiteness 16

2.2.3 Unfoldability 17

2.2.4 Conflict between Unfoldability and Definiteness 18

2.3 Parameter Passing Techniques 19

2.3.1 Effect in Deterministic Languages 19

2.3.2 Effect in Nondeterministic Languages 20
2.3.3 Relation to Conflict 21

3 Program Semantics **23**
3.1 Denotational Semantics 24
3.1.1 Simple Semantic Domains 24
3.1.2 Valuation Functions 26
3.1.3 Recursive Functions 27
3.2 Semantics of Nondeterministic Programs 30
3.2.1 Power Domains 32
3.2.2 Types of Nondeterminism 33
3.2.3 Singular versus Plural Semantics 34
3.2.4 Strict versus Nonstrict Semantics 35
3.2.5 Parameter Passing techniques 36
3.3 Implementation Issues 37
3.4 Alternative Semantics 38
3.4.1 Partially Deterministic Languages 39
3.4.2 Hughes Natural Semantics 39

4 Methods of Hiding Nondeterminism **41**
4.1 Streams 41
4.1.1 Definition of *Amb* using Streams 43
4.1.2 Semantics 44
4.1.3 Classification 45
4.1.4 Disadvantages 45
4.2 Burton's Method 46
4.2.1 Definition of *choice* 46
4.2.2 Semantics 47
4.2.3 Classification 48
4.2.4 Disadvantages 49
4.3 Monads 50
4.4 Peyton Jones' Method 51
4.5 Comparison of Methods 52

5 Parallel Processing and Functional Languages **54**
5.1 Functional Parallel Processing . . . 54
5.1.1 Explicit Parallelism . . . 55
5.1.2 Implicit Parallelism . . . 56
5.1.3 Implicit Parallelism and Annotations . . . 57
5.2 List Parallelism . . . 58
5.2.1 List Notation . . . 59
5.2.2 List Structure . . . 60
5.3 Efficiency . . . 63
5.3.1 Granularity . . . 63
5.3.2 Degree of Parallelism . . . 64
5.4 Expressibility . . . 65

6 Parallel Functional Programming with Linda **67**
6.1 Linda . . . 67
6.1.1 Tuple Space . . . 68
6.1.2 Linda Constructs . . . 69
6.1.3 Advantages and Criticisms . . . 70
6.2 Explicit Parallelism using Linda . . . 71
6.2.1 Linda and Streams . . . 73
6.3 Implicit Parallelism using Linda . . . 74

7 Optimization of Linda Implementation **76**
7.1 Properties of Lists and Bags . . . 78
7.1.1 Boom Hierarchy . . . 78
7.1.2 The Bird-Meertens Formalism . . . 80
7.2 List to Bag Convertibility . . . 83
7.2.1 Bag Propagation . . . 85
7.3 Bag Analysis . . . 86
7.3.1 Backward Analysis . . . 87
7.3.2 Abstract Domain . . . 87
7.3.3 Abstract Context Functions . . . 88
7.3.4 Propagating Contexts . . . 90

7.3.5 Extensions to Bag Analysis 93
7.4 Implications of Optimization 95

8 Conclusion **98**

A Definitions from the Bird-Meertens Formalism **101**
A.1 Notation 101
A.2 Function Types 102
A.3 Definition of Map and Reduce 102
A.4 Promotion Rules 103

B Conversion Rule Proofs **104**
B.1 Reduction Conversion Rule 104
B.2 Bag Propagation Rule through Map 105

C Undecidability of the Commutivity Problem **106**
C.1 Proof 106

References **108**

List of Figures

1.1 Two Different Reduction Sequences. 5

3.1 Properties of the Nondeterministic Language Variants. 37

4.1 Classification of Approaches to Nondeterminism. 53

6.1 System Structure for a Typical "Farm of Workers". 74

7.1 Summary of Boom Hierarchy. 80
7.2 Abstract Context Functions. 89
7.3 Abstract Context Functions for Primitive List Functions. . . 94

A.1 Definition of Map and Reduce. 102
A.2 Bird-Meertens Promotion Rules. 103

Chapter 1

Introduction

Functional languages have continued to grow in popularity over the past two decades, and advocates continue to find new applications. Nondeterminism is an essential requirement in applications such as operating systems and is desirable in some forms of parallel processing. To successfully implement these types of applications, nondeterminism needs to be integrated into functional languages. There have been a number of approaches but problems exist.

Nondeterministic choice can wreck some of the elegant mathematical properties of the pure languages. The interaction of nondeterminism with specific implementation mechanisms, or nondeterminism in the presence of non-terminating computations has some non-obvious and subtle consequences.

The effects of the integration of different types of nondeterminism need to be classified. Then the trade-offs which result in exploiting certain types of nondeterminism in specific situations can be identified. In this manner, informed decisions on how to extend functional languages can be made in order to profitably implement specific applications which require nondeterminism.

To date, the issues and approaches have not been collated and organized. In addition, the terminology used by different authors is often inconsistent or contradictory. This thesis draws together the disparate work, providing a

consistent framework which will assist functional language implementors.

1.1 Functional Programming Languages

By 1989, consensus on a universal definition of functional languages had not been reached [Hudak, 1989]. It is important for us to begin our study with a precise description of what we understand to be functional languages.

1.1.1 Structure

The functional programming style is characterized by the lack of implicit state. The result is a declarative language whose underlying model of computation is the function [Hudak, 1989] [Hughes, 1989] [Bird, 1988]. The state is carried explicitly in state-oriented computations by passing parameters to the functions. As a result, an important characteristic of functional languages is the lack of assignment statements.

Functional languages do not include control structures such as while loops. These control the sequence in which state is modified in imperative languages, so that a precise and deterministic result is achieved. The lack of implicit state renders these constructs redundant in functional languages. However, looping is achieved through the use of recursion [Hudak, 1989].

Functions can be expressed in a manner which emulates the structure of mathematical equations. A single function is defined by several separate equations. Each equation is valid under certain circumstances according to the rules of pattern matching.

Functions, known as higher order functions, can be passed as arguments, stored in data structures and returned as results. This feature is profitably used in modularization of programs to "glue" separate modules together [Hughes, 1989].

Lazy evaluation is another distinguishing concept of functional languages. This ensures that an argument is only evaluated when and if it is needed. Lazy evaluation can be exploited in order to evaluate (finite portions of) infinite data structures [Peyton Jones, 1987] [Michaelson, 1989].

1.1.2 Properties

Bird and Wadler state "The primary role of the programmer is to construct a function to solve a problem. This function $\cdots$ is expressed in notation that obeys normal mathematical principles" [Bird, 1988]. To understand this statement, consider the following mathematical equation

$$f(x) = x + x$$

The following equalities hold

$$\begin{aligned} f(5+7) &= (5+7)+(5+7) \\ &= 12+12 \\ &= f(12) \\ &= 24 \end{aligned}$$

It is important to note that once a value has been bound to a variable, it does not vary. For example x is always equal to the value of $(5+7)$. Even though the value of $(5+7)$ cannot change, the expression can be replaced by equivalent expressions during simplification.

Any subexpression may be replaced by an equivalent subexpression without affecting the generality or properties of the function. This is loosely termed referential transparency, and is assumed to be a property of typical mathematical systems.

The significance of Bird and Wadler's statement is that pure functional languages exhibit these mathematical properties. The mathematical nature of

functional languages is exploited in formal reasoning about functional programs during program optimization and verification.

It may seem intuitive to assume that the same statement holds for programs written using any programming style. However, imperative programs are good counter examples. Consider the following function written in C.

```
int Double (int x) {
    if (global_flag == 0) {
      global_flag = 1;
      return x;
    }
    else
      return x+x;
}
```

The value of the global variable may vary during evaluation and *Double* not only depends on the global variable, but explicitly alters it. The replacement of subexpressions with simpler equivalent subexpressions is no longer governed by 'standard' mathematical rules. For example $double(5) + double(5)$ may evaluate to 20, but only if the flag is non-zero before the expression is evaluated. Therefore, the language is not referentially transparent.

1.1.3 Church-Rosser Theorem and Confluence

One model of evaluation of a function proceeds by selecting and reducing a reducible expression (redex). More than one redex may be present in a functional expression. How does the evaluation order of the redexes effect the result of evaluating the expression?

The expression $f(5 + 7) = (5 + 7) + (5 + 7)$ contains two redexes. The final

result obtained is independent of the evaluation sequence used, as indicated below.

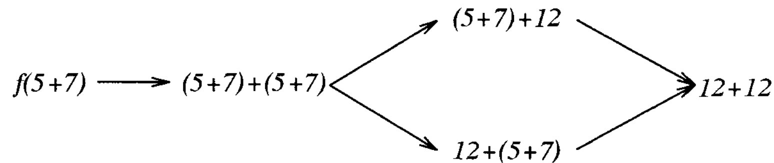

Figure 1.1: Two Different Reduction Sequences.

But is this one example indicative of functions in general?

According to the Church-Rosser theorem, in referentially transparent languages, the same result is obtained regardless of the chosen reduction order (if the evaluation process terminates). An important consequence is that functional programmers are freed from concerns involving evaluation order [Hudak, 1989].

The Church-Rosser theorem also indicates that a reduction sequence known as normal order reduction will always return a result if one exists. Lazy evaluation is an implementation of normal order reduction and, therefore, ensures termination whenever possible [Peyton Jones, 1987].

1.1.4 Pure and Impure Functional Languages

It is generally agreed that the functional style is characterized by the use of function application and composition rather than sequencing. In addition, the presence of other constructs such as higher order functions, discussed in section 1.1.1, are characteristic of the style. Therefore, why is it impossible to cite a unique globally accepted definition of functional languages?

Some authors demand the absence of any primitives which result in side ef-

fects. This agrees with the above description of functional languages. The resultant languages are referred to as pure languages. Some authors are less demanding. They believe that functional languages should adhere to the general functional style. However, they only discourage the use but do not demand the absence of primitives which result in side effects [Hudak, 1989].

Pure languages are referentially transparent and exhibit confluence whilst impure languages lack some, if not all, of these properties. Therefore, the algebraic transformations of an impure program are limited. This directly impacts on the ability to optimize and verify the code. Conversely, it is argued that efficiency and even expressibility of an impure language is greater than that of a pure language.

However, purists advocate that "purely functional languages are not only sufficient for general computing needs but are also better because of their purity" [Hudak, 1989]. In support of impure languages, John Hughes argues "it is a logical impossibility to make a language more powerful by omitting features, no matter how bad they may be" [Hughes, 1989].

We find the ability to verify and optimize code attractive, especially since these techniques rely on already known and accepted mathematical principles. In addition, confluence allows for elegant parallelisation of functional languages [Peyton Jones, 1989].

For the remainder of this thesis, we adopt the purists approach. Deterministic languages discussed in this thesis can be considered to be referentially transparent. Unless otherwise stated, we assume a deterministic functional language does not contain assignment statements or other constructs which introduce side effects. In addition, the programs must exhibit the property of confluence. Therefore, using this definition of a pure language, the output of a pure function is solely dependent on its explicit input.

For the rest of this thesis, we assume all functional languages are implemented using lazy evaluation techniques. In some cases, it is necessary to assume the language is implemented eagerly. This will be indicated explicitly.

1.2 Introduction to Nondeterminism

The following two definitions of determinism will be used in this thesis.

- DEF1:- "A program is deterministic if its evaluation on the same input always produces the same output" [Schmidt, 1988].
- DEF2:- "Assume we are given a discrete system subject to changes, or state transitions. If at every point of time, the system's present state contributes an amount of information sufficient for us to deduce its state after the next transition, then it is said to be deterministic" [Sondergaard & Sestoft, 1992].

If the evaluation of a program (for a given set of input) results in a set of possible values (DEF1 is violated), the resultant language is said to be nondeterministic.
If DEF2 is violated, the state after the next state transition cannot be predicted at every point in time. Again the resultant language is nondeterministic.

At first, the above two definitions appear to be identical. However, subtle differences exist. DEF2 does not make any reference to the final result. If a language is nondeterministic due to a violation of DEF2, the result may be a single result or a set of possible results. DEF1 refers to the overall result obtained after the evaluation of a program. If the program is nondeterministic (due to a violation of DEF1), more than one possible answer must exist.

1.2.1 Strong and Weak Nondeterminism

To fully understand the above subtleties two types of nondeterminism namely strong and weak nondeterminism need to be defined.

Strong Nondeterminism:- The case where one of a possible number of results is acceptable, and no mechanism is used to favour any particular result, is referred to as strong nondeterminism.

Weak Nondeterminism:- The case where use of nondeterministic components of a program results in a deterministic program, is referred to as weak nondeterminism. "That is, the final results should never depend on the particular choices taken by an evaluator" [Sondergaard & Sestoft, 1992].

If DEF1 is violated, then the language is definitely strongly nondeterministic. If DEF2 is violated, it may be that DEF1 still holds or it may be violated. In the former case the language is weakly nondeterministic.

1.2.2 Effects of Nondeterminism

There are definite consequences to introducing strong nondeterminism using nondeterministic constructs. These consequences are discussed in detail in section 2.2.4 once the necessary terms and constructs have been defined. However, the problem is outlined below.

Consider the function

$$f(x) = x + x$$

If the language is referentially transparent, the following is always true.

$$\begin{aligned} f(\mathit{some\ expr}) &= \mathit{some\ expr} + \mathit{some\ expr} \\ &= 2 * \mathit{some\ expr} \end{aligned}$$

Assume *some expr* can be substituted by more than one possible value (the language is strongly nondeterministic). The same result cannot be guaranteed each time *some expr* is evaluated. Therefore, it cannot be assumed that

$2 * some\ expr$ is always equal to $some\ expr + some\ expr$. In other words, the language is not referentially transparent (loosely defined in section 1.1.2).

1.3 Project Goals

Nondeterminism in functional languages may limit the mathematical tractability of the language. Consequently, some algebraic transformations, required during optimization and verification of functional programs, cannot be performed with confidence. If nondeterminism is not integrated into functional languages, these languages cannot be used to implement a number of applications which rely on nondeterminism. Hence, it appears as if the use of pure functional languages is limited.

Our goal is to investigate the impact of nondeterminism in functional languages. In particular, we wish to establish if there is a way of implementing pure functional languages with the added advantages afforded by nondeterminism.

In order to discuss the implications of nondeterminism in functional language clearly and concisely, we introduce and use the definitions and terminology suggested by Sondergaard and Sestoft [Sondergaard & Sestof, 1990]. In addition, we investigate the use of denotational semantics to specify nondeterministic languages. This allows for precise characterization of the nature of nondeterministic functional languages and the resultant trade-offs of adding strongly nondeterministic constructs.

However, understanding these consequences is not sufficient. Our next goal is to investigate alternative methods of hiding nondeterminism which overcome or sidestep the problems outlined in section 1.2.2.

Hughes [Hughes & Moran, 1995] and Jackson and Burton [Jackson & Burton, 1994] attempt to solve the problem by changing the semantics of the

language. This approach leads to a restriction of the language semantics or a restriction on the type of nondeterminism which may be present. We investigate and compare alternative methods, which do not impose these restrictions.

Next, we investigate whether pure functional languages are indeed inappropriate for the implementation of parallel systems. This serves as a good illustration of how to employ methods of introducing and hiding strong nondeterminism. In particular, we focus on Linda, a parallel processing paradigm, in this discussion.

Linda consists of a set of constructs which introduce the notion of processes and are used to extend sequential languages. Asynchronous communication is achieved through a global state space called tuple space. We show that, if Linda constructs are introduced explicitly into a functional language, strong nondeterminism and the related complications result.

Instead of exploiting explicit parallelism, we investigate the use of Linda in order to exploit weak nondeterminism. The Linda paradigm is used at a level transparent to the user without sacrificing important properties of functional languages. Lists are used to express parallelism - any redexes within a list will be placed into tuple space to be evaluated in parallel.

In functional languages, the order of lists elements is important. However, any parallel language which accesses a distributed data structure (pool of data), may benefit from reduced synchronization constraints. In these cases, the data may be removed from the pool in any order and the structure used to represent these values would ideally be a bag instead of a list. For optimization purposes, we use the Bird-Meertens formalism to characterize when a list may be treated as a bag. We also introduce a form of backward analysis which can be used to detect when this form of optimization is possible.

Chapter 2

Nondeterministic Constructs and Strong Nondeterminism

McCarthy's ambiguous operator, *amb*, returns one of its two arguments. E_1 *amb* E_2 returns either E_1 or E_2. The choice of argument is random with one exception, if E_1 is undefined then the value of (E_1 *amb* E_2) is E_2 and vice versa [Hughes & Moran, 1995].

According to DEF1 in section 1.2, the *amb* construct introduces nondeterminism since, given the same input of E_1 and E_2, there exists a set of possible results $\{E_1, E_2\}$. This type of nondeterminism is classified as strong nondeterminism. The consequences of introducing strong nondeterminism into functional languages will be discussed in detail in the rest of this chapter.

2.1 Types of Strong Nondeterminism

The effects of nontermination on expressions which contain a nondeterministic construct such as McCarthy's *amb* need to be considered. "The reason is that now we have to discuss possibly nonterminating computations and the way in which possible termination depends on the non-deterministic choices made during the evaluation of the program" [Sondergaard & Sestoft, 1992].

The definition of *amb* above states that if either argument is undefined the alternative argument is returned as the result. If both choices are undefined, then the result is undefined. However, there are other ways of defining *amb*.

Three different types of strong nondeterminism can be defined. The type which is present within a language depends on the way in which *amb* is defined to handle nontermination.

The following expression will be used to illustrate the difference between the three types of strong nondeterminism and the conditions under which each exists. For the purpose of this discussion, bottom ($\perp$) is used to represent a nonterminating computation or undefined argument.

$$f(\perp \; amb \; 3) + f(2 \; amb \; 4) \; where \; f(x) = x + x \tag{2.1}$$

2.1.1 Angelic Nondeterminism

McCarthy's *amb* can be defined such that all choices are made in favour of termination if possible [Sondergaard & Sestoft, 1992]. This is consistent with the definition given above and is referred to as angelic nondeterminism.

Using this method, equation 2.1 can be rewritten as $f(3) + f(2 \; amb \; 4)$ since termination is always favoured and $(\perp \; amb \; 3)$ evaluates to 3.

2.1.2 Demonic Nondeterminism

amb can be defined in a slightly different way. Demonic nondeterminism can be implemented by specifying that as soon as one of the arguments is nonterminating, the evaluation of *amb* should result in nontermination. "With demonic nondeterminism choices are made in favour of nontermination if at all possible" [Sondergaard & Sestoft, 1992].

Consider expression 2.1, $(\perp \; amb \; 3)$ will always evaluate to $\perp$. Therefore, the example can be rewritten as $f(\perp) + f(2 \; amb \; 4)$.

2.1.3 Erratic Nondeterminism

It may seem more intuitive to ensure that the implementation of *amb* does not favour either termination or nontermination. This is referred to as erratic nondeterminism. "In this kind of nondeterminism choices are made, operationally speaking, by flipping a coin, which means that nothing is done to obtain or prevent termination" [Sondergaard & Sestoft, 1992].

If this approach is chosen ($\perp$ *amb* 3) may either return the value of $\perp$ or 3. The evaluation process does not favour either result.

2.1.4 Global versus Local Nondeterminism

Consider the expression below, and assume choices are made in favour of termination (angelic nondeterminism).

$$f(0\ amb\ 1)\ where\ f(x) = if\ (x = 0)\ then\ f(0)\ else\ x + x \tag{2.2}$$

Evaluation of the expression may be restricted to ensure each choice only evaluates to bottom if both arguments are undefined. This method does not consider the effects the choice has on later evaluation of the expression or whether evaluation of the expression terminates as a whole. The resultant nondeterminism is known as local angelic nondeterminism.

In other words, when (0 *amb* 1) is evaluated, either 0 or 1 is a possible result since neither is equal to bottom (the only choice which has to be avoided if local choices are made). Therefore, evaluation of expression 2.2, assuming local angelic nondeterminism, might not terminate.

The choice of 1 ensures termination of the evaluation process of expression 2.2, whereas 0 leads to later nontermination. The case where the overall termination of the evaluation of an expression is considered is referred to as global angelic nondeterminism. Similar cases should be taken into consider-

ation when implementing demonic nondeterminism.

The three main types of nondeterminism plus the global variations give five possible semantics for *amb*.

2.2 Effect on Language Properties

Once the method of handling nontermination has been considered, other complications which result due to nondeterminism still need to be understood and handled. The following nondeterministic expression will be used to illustrate this point.

$$f(2\ amb\ 4)\ where\ f(x) = x + x \tag{2.3}$$

Simple unfolding leads to $(2\ amb\ 4) + (2\ amb\ 4)$. When evaluated, any of the results listed below are possible.

$$\begin{array}{ccccc} 2 & + & 2 & or \\ 2 & + & 4 & or \\ 4 & + & 2 & or \\ 4 & + & 4 & \end{array}$$

By contrast, if $(2\ amb\ 4)$ is evaluated before its result is bound to x, the possible results are

$$\begin{array}{cccc} 2 & + & 2 & or \\ 4 & + & 4 & \end{array}$$

This contradiction indicates that, once *amb* is a construct of the language, the language is no longer referentially transparent according to the definition in section 1.1.2.

Sestoft and Sondergaard illustrate that the definition of referentially transparency is not consistent between texts [Sondergaard & Sestof, 1990]. Essentially, a number of different concepts are all referred to as referential transparency. Consequently, this leads to inconsistency in defining the effects of nondeterminism in functional languages.

Sondergaard and Sestoft try to clarify the notion of referential transparency. They decouple the concepts into three independent properties namely, referential transparency, unfoldability, and definiteness. In the process referential transparency was re-defined and simplified. In addition the emphasis is shifted from referential transparency to the core issues of definiteness and unfoldability.

Using these definitions as a basis, the effects of nondeterminism in functional languages can be expressed more precisely. Therefore, we discuss Sondergaard and Sestoft's definitions below and adopt this terminology for the rest of this thesis.

2.2.1 Referential Transparency

Sondergaard and Sestoft suggest the following definition for referential transparency.

> "An operator is referentially transparent if it preserves applicability of Leibniz's law, or substitutivity of identity: the principle that any subexpression can be replaced by any other equal in value" [Sondergaard & Sestoft, 1992].

Using the above definition it can be concluded that expression 2.3, although perhaps not initially apparent, is referentially transparent. Although $(2\ amb\ 4)$ will eventually evaluate to 4 or 2, to be strictly correct, $(2\ amb\ 4)$ should be replaced by the set $\{2, 4\}$ to preserve referential transparency.

This definition is different to that defined in section 1.1.2. Essentially a different meaning of equality is used in each case. In section 1.1.2, an expression always denotes a single value. The notation that one expression can replace another equal in value essentially means that both expressions must denote the same single value.

By contrast, Sondergaard and Sestoft's expressions denote sets of values. Their replacement with an "equal" expression implies any other expression which denotes the same set. Since an *amb* expression is said to equal a set of values, the actual choice has been delayed and the effects are not characterized by referential transparency.

The use of this form of equality means referential transparency co-exists with nondeterminism. However, the effects of nondeterminism have not disappeared. As indicated above, the emphasis has now been placed on the more specific notions of definiteness and unfoldability which can more precisely express the effects of choosing a single value from the set of possible values during evaluation of a nondeterministic expression.

2.2.2 Definiteness

A language is definite if a variable such as x has the same value at each position (within a single scope) in a function [Sondergaard & Sestoft, 1992].

This property is relied on quite heavily in typical mathematical systems. When a mathematical expression is evaluated, the value of a variable remains constant even if it occurs more than once.

In expression 2.3, x occurs more than once, within a single scope. If x can only equal 2 each time it occurs, or alternatively 4 at each position, the language is definite and the possible results are

$$2 \; + \; 2 \quad or$$
$$4 \; + \; 4$$

Note that if the language is indefinite, the following results are also possible

$$2 \; + \; 4 \quad or$$
$$4 \; + \; 2$$

2.2.3 Unfoldability

Using lambda calculus [Michaelson, 1989], a language is unfoldable if $(\lambda x \cdot e)e' = [e'/x]e$ [Sondergaard & Sestoft, 1992]

Consider

$$f(2\ amb\ 4)\ where\ f(x) = x + x$$

For the above expression to be unfoldable the following expression must always be true

$$f(2\ amb\ 4) = ((2\ amb\ 4) + (2\ amb\ 4))$$

The unfolded expression $((2\ amb\ 4) + (2\ amb\ 4))$ evaluates to

$$4 \; + \; 4 \quad or$$
$$2 \; + \; 2 \quad or$$
$$2 \; + \; 4 \quad or$$
$$4 \; + \; 2$$

The same set of results is obtainable when $f(2\ amb\ 4)$ is evaluated, but only if the language is not definite, see section 2.2.2. Therefore, unfoldability implies the language is not definite.

In addition, regardless of the method used to evaluate $f(2\ amb\ 4)$, if the language is definite the possible results are limited to

$$\begin{array}{l} 4 + 4 \quad or \\ 2 + 2 \end{array}$$

Therefore, definiteness implies the language is not unfoldable.

Taking both cases, we conclude that this fragment cannot be definite and unfoldable simultaneously.

2.2.4 Conflict between Unfoldability and Definiteness

The above discussion indicates that expression 2.3 can either be definite, or unfoldable but cannot be definite and unfoldable simultaneously. Sondergaard and Sestoft demonstrate that this is the case for all functions which contain nondeterministic constructs. "In the presence of non-determinism, we cannot obtain both definiteness and unfoldability" [Sondergaard & Sestof, 1990].

This statement has important consequences. When strong nondeterminism is introduced into functional languages through the use of nondeterministic constructs, a choice needs to be made between unfoldability and definiteness. Is unfoldability more important than definiteness?

Algebraic transformations based on folding and unfolding definitions are vital in proving properties about functional programs and are used extensively in compilation techniques. As a result, unfoldability is a property which functional programmers consider important. Definiteness is considered to be an intuitive and expected property of most systems. Therefore, there is no easy solution to this conflict once it exists.

2.3 Parameter Passing Techniques

Functional languages often make use of lazy evaluation. Lazy evaluation works on the principle of only evaluating an expression when the result is needed or evaluation cannot be postponed any longer [Peyton Jones, 1987] [MacLennan, 1990].

Technically, in deterministic systems, there are at least two different lazy parameter passing techniques namely, call-by-name and call-by-need. Since both are lazy techniques, evaluation of the arguments is postponed until the value is required (if at all) [Peyton Jones, 1987]. The arguments are passed to the function before they are evaluated and remain unevaluated until they are needed inside the function body [Field & Harrison, 1988].

The essential difference between these techniques is only apparent when a variable occurs more than once in an expression. The difference is centered around the decision of whether to re-evaluate the argument each time the variable occurs. Call-by-name dictates that the argument is re-evaluated for each occurrence of the variable. When call-by-need is used, once an argument is evaluated, the value is stored for possible future use and retrieved each time it is needed [Field & Harrison, 1988] [Reade, 1991].

However, it is important to note that call-by-need is not equivalent to the call-by-value technique, even though in both cases the value is stored after evaluation. The call-by-value technique, also classified as eager evaluation, 'eagerly' evaluates all arguments before the function is evaluated, regardless of whether the argument values will be used or not within the function body [Field & Harrison, 1988] [Peyton Jones, 1987].

2.3.1 Effect in Deterministic Languages

If the language is deterministic the results of using call-by-need and call-by-name are identical. "In traditional languages, call-by-name and call-by-need

behave differently if the argument expression produces side effects. In a functional language there are no side effects and so these two calling mechanisms always generate the same result" [Field & Harrison, 1988].

It is obvious, however, that call-by-name requires a lot of unnecessary re-evaluation of arguments. As a result, call-by-need is favoured by implementors.

An independent, but often confused notion, is that of strictness. "A function is said to be strict if the result is undefined when it is applied to an undefined argument, and is said to non-strict otherwise" [Field & Harrison, 1988] [Reade, 1991]. This can be expressed as

$$f(\perp) = \perp$$

[Bird, 1988] [Peyton Jones, 1987]

In deterministic languages, lazy evaluation can be used to implement strict or non-strict semantics. Eager evaluation can only be used to implement strict semantics. This is the favoured technique if strict semantics is required since it is more efficient than lazy evaluation.

However, the terminology tends to be used loosely in this area. "If a function is non-strict, we say that it is lazy. Technically, this is an abuse of terminology, since lazy evaluation is an implementation technique which implements non-strict semantics" [Peyton Jones, 1987]. It is important to note the difference between these concepts, especially when dealing with nondeterminism.

2.3.2 Effect in Nondeterministic Languages

More thought needs to be given to decisions regarding parameter passing techniques when nondeterminism is present. The difference between the call-by-name and call-by-need is no-longer restricted to an issue of efficiency. The

set of results obtained depends on the parameter passing technique used.

As an illustration, consider the following expression

$$f(0\ amb\ 1)\ where\ f(x) = x + x$$

When the above function is evaluated using the call-by-name technique, $(0\ amb\ 1)$ is re-evaluated each time an x is encountered. Therefore, $f(0\ amb\ 1)$ is equivalent to $(0\ amb\ 1) + (0\ amb\ 1)$. In other words, the language is unfoldable.

If the call-by-need technique is used $(0\ amb\ 1)$ is evaluated once only, and the result is stored for future use. Therefore, the value of x is the same at each position. This value is determined during the initial evaluation of the argument. Therefore, the language is definite.

2.3.3 Relation to Conflict

If definiteness is a required property of a language, then a parameter passing technique which ensures that each occurrence of a variable is identical must be used. Both call-by-value and call-by-need satisfy this requirement.

When the call-by-name technique is used, there are no restrictions to ensure that all occurrences of a single variable are identical and the resultant function is unfoldable.

As indicated in section 2.2.4, unfoldability and definiteness cannot exist simultaneously in the presence of nondeterminism. Therefore, the choice of parameter passing technique should be made with the required properties in mind.

The above discussion is from the perspective of the implementor. In addition, only the parameter passing techniques traditionally used in deterministic implementations have been discussed. In papers by Hennessy and Ashcroft

[Hennesy & Ashcroft, 1977], Clinger [Clinger, 1982] and Sestoft and Sondergaard [Sondergaard & Sestoft, 1992] it is indicated that the situation is even more complex. These complexities are discussed in the following chapter.

Chapter 3

Program Semantics

The discussion in chapter 2 indicates that different parameter passing techniques, in the presence of nondeterminism, can greatly effect the properties of a strongly nondeterministic language.

That is, a number of different meanings (the semantics of the language) may exist for a language with a specific syntax (grammatical structure). Different methods can be used to formalize the semantics of a language. Operational Semantics, Denotational Semantics, and Axiomatic Semantics [Nielson & Nielson, 1992] are three common approaches.

A full understanding of the impact of nondeterminism relies on further issues namely, singular/plural semantics, nonstrict/strict functions and the three different types of nondeterminism [Sondergaard & Sestoft, 1992].

Denotational Semantics is an effective tool for formalizing the impact different values and combinations of the above three properties have on a nondeterministic language. Therefore, this technique and its use for formalizing strongly nondeterministic functional languages is discussed in this chapter.

3.1 Denotational Semantics

"Denotational Semantics: Meanings are modeled by mathematical objects that represent the effect of executing the constructs. Thus only the effect is of interest, not how it is obtained" [Nielson & Nielson, 1992].

These mathematical objects are called denotations and are elements of a semantic domain. The denotational description of a program is determined by mapping the syntactic descriptions onto these domain elements.

3.1.1 Simple Semantic Domains

Consider the following simple language expressed using the Backus-Naur Form (BNF). In preparation for the later introduction of Sondergaard and Sestoft's [Sondergaard & Sestoft, 1992] language which includes a single function of exactly two arguments, x and y, we begin with a simplified language that includes x and y, but omits the function at this stage.

$exp \rightarrow int$	constant
$\quad \mid \; x$	first variable
$\quad \mid \; y$	second variable
$\quad \mid \; exp + exp$	addition

The definition of the semantic domain is the first step to finding the meaning (denotation) of programs expressed using this syntax. Assume a domain is a set of elements with common properties.

A primitive semantic domain is "the set fundamental to the application being studied. Its elements are atomic and they are used as answers or 'semantic outputs'" [Schmidt, 1988]. In the above example, *int* (the set of integer values) is the primitive domain since the meaning of a function can be expressed

in terms of its integer value.

The terminals of a language can be mapped directly onto the primitive domain. However, not all syntactic structures can be described using this direct mapping. The language also consists of complex syntactic structures, for example the addition function which takes two arguments and returns a value.

To express the meaning of these complex structures the use of a compound domain, a domain constructed from one or more primitive domains, is required. For this example, the compound domain is a function space represented as $A \to B$. Each function in this domain space takes a value from domain A and returns a value in domain B. The $\to$ constructor creates the compound domain from its operand domains.

The overall denotation of an expression must be in the domain *int* as discussed above. Therefore, B is the set *int*. However, each function takes two arguments. This cannot be represented using the primitive semantic domain *int*.

Domain A is actually a compound domain ($int \times int$, or int^2) called a product domain. In general, a product domain consists of ordered pairs constructed from two underlying domains X and Y using the cross product constructor $\times$ which is defined as

$$X \times Y = \{\, (x, y) \mid x \in X \ and \ y \in Y \,\} \tag{3.1}$$

Operations can be applied to compound domains. Two disassembly operations (which return the first and second element of an ordered pair respectively), are applicable to the product domain. Function application on the function space $A \to B$ returns the appropriate element in domain B, given an element in domain A.

3.1.2 Valuation Functions

Valuation functions map syntactic constructs onto their denotations. "There is one valuation function for every nonterminal in the BNF description of a computer language" [Melton *et al.*, 1994].

The only nonterminal in the example language is *exp*. The type of the evaluation function for *exp* is,

$$E : exp \longrightarrow int^2 \longrightarrow int$$

This indicates that the valuation function takes a syntactic expression and a pair argument and maps them onto the set *int*. The argument pair effectively represents the current values to which x and y are bound (each from the set *int*).

The valuation function for *exp* is described below where
$n, v, u \in int$; $w \in int^2$

$$\begin{aligned}
E[\![n]\!]w &= \mathrm{n} \\
E[\![x]\!]w &= u \text{ where } (u,v) = w \\
E[\![y]\!]w &= v \text{ where } (u,v) = w \\
E[\![e_1 + e_2]\!]w &= E[\![e_1]\!]w + E[\![e_2]\!]w
\end{aligned}$$

In this example w represents the "state" or "environment" of the expression which is an ordered pair of values. The syntactic expressions are enclosed by the brackets $[\![\ ]\!]$. This ensures these expressions are not confused with the valuation functions or semantic expressions.

Most importantly, the meaning of complex syntactic expressions is built up from the meanings of the subexpression which constitute the complex expressions. This is known as compositionality and is characteristic of Denotational Semantics.

Therefore, the denotation of an expression consists of the combination of the denotations of the subexpressions. "The domain operations which are applied are in fact the denotations of the syntactic constructors which build up the complex syntactic construct from its simple substructures" [Melton *et al.*, 1994].

3.1.3 Recursive Functions

Recursive functions are fundamental to the functional style of programming. A single two-argument function is now added to the language as well as a conditional construct.

$pgm \longrightarrow$	$f(x,y) == exp$	
$exp \longrightarrow$	int	constant
	$\mid x$	first variable
	$\mid y$	second variable
	$\mid exp + exp$	addition
	$\mid exp \longrightarrow exp, exp$	conditional
	$\mid f(exp, exp)$	function application

The recursive function may be undefined for some values of x and y. Therefore, the set int needs to be extended. A new set $int_\perp = int \cup \{\perp\}$ allows for the representation of this function (other restrictions on $int_\perp$ will be discussed later in this chapter).

Before the valuation functions of this extended language are defined, it is necessary to understand the notion of least fixed points. Consider the following recursive program described using the above BNF description.

$$f(x,y) == y \rightarrow 1, f(x,y)$$

This may be re-written as

$$F = \tau F$$

where $F = f(x,y)$
$\tau(g) = y \rightarrow 1, g$

A function defined by the recursive program (meaning of the recursive program) is the solution for F. That is, it is a least fixed point[1] of τ. Notice that the recursive function defines more than one function, since F can be substituted by any of the following fixed points of τ [Manna, 1974].

$$g_{\perp}(x,y) == y \rightarrow 1, \perp$$
$$g_1(x,y) == y \rightarrow 1, 1$$
$$\ldots$$
$$g_n(x,y) == y \rightarrow 1, n$$

Since more than one fixed point may exist, the least fixed point is taken to be the function defined by the recursive program. The least fixed point "shares its results" with all other fixed points [Nielson & Nielson, 1992]. To formalize this notion of sharing results, an ordering $\sqsubseteq$, called a partial ordering which is reflexive, transitive, and anti-symmetric is imposed on sets of values that can be denoted.

Fixed point theory [Manna, 1974] [Nielson & Nielson, 1992] [Melton *et al.*, 1994] [Schmidt, 1988] shows that a functional such as τ always has a unique least fixed point since denoted values are restricted to complete partial orderings (Cpo's) (which are referred to as domains in this context) and all functions on these Cpo's are continuous.

The fixed points of the above recursive program are ordered as in

$$g_{\perp} \sqsubseteq g_1$$

[1] x is a fixed point of τ if $\tau(x) = x$

$$g_\perp \sqsubseteq g_2$$

$$\ldots$$

$$g_\perp \sqsubseteq g_n$$

In fact, $g_\perp$ is the least fixed point and therefore the function defined by the recursive program $f(x,y) == y \rightarrow 1, f(x,y)$.

Now consider the valuation functions of the extended example language that assign denotations to programs and expressions. They have the types

$F{:}pgm \rightarrow int_\perp$
$E{:}exp \rightarrow den \rightarrow den$ where $den = int_\perp{}^2 \rightarrow int_\perp$

The semantics of the extended example language is described below where $d \in den$; $n \in int$; $u, v \in int_\perp$; $w \in int_\perp{}^2$;
z_i abbreviates $E[\![e_i]\!]dw \in int_\perp$.

$$
\begin{aligned}
F[\![f(x,y) == e]\!] &= d(0,0)w \ \textit{where rec}\ d = E[\![e]\!]dw \\
E[\![n]\!]dw &= n \\
E[\![x]\!]dw &= u \text{ where } (u,v) = w \\
E[\![y]\!]dw &= v \text{ where } (u,v) = w \\
E[\![e_1 + e_2]\!]dw &= \text{if } z_1 = \perp \text{ or } z_2 = \perp \text{ then } \perp \\
&\quad\ \text{else } z_1 + z_2 \\
E[\![e_1 \rightarrow e_2, e_3]\!]dw &= \text{if } z_1 = \perp \text{ then } \perp \text{ else if } z_1 \neq 0 \\
&\quad\ \text{then } z_2 \text{ else } z_3 \\
E[\![f(e_1, e_2)]\!]dw &= d(z_1, z_2)
\end{aligned}
$$

This denotational semantic description describes recursive programs. Every recursive program has a "state" or "environment" represented by $w = (u,v)$. The state w is always initialized to $(0,0)$. In other words, variables x and y are bound to 0 in the initial environment. Different bindings for x and y are created during function application. The semantics of the conditional follows

that of the C language. If e_1 evaluates to non-zero the value of e_2 is chosen, otherwise e_3 is chosen.

Notice that the recursive nature of $f(x, y)$ is captured in the semantic description by the definition of $dw = E[\![e]\!]\, dw$. This d is passed explicitly as an extra argument to E and is an element of the domain *den*. The meaning of the recursive program e is the least fixed point of $E[\![e]\!]$, represented by d. For this least fixed point to exist, $E[\![e]\!]$ must be continuous on some domain.

Up until now, $int_\perp$ used in the above semantic description of the example language has been considered to be a set. However, to support the theory required to establish the existence of d, an ordering is imposed on the set and is referred to as a domain. This ordering is
$(\forall x_1, x_2 \in int_\perp)$
$[(x_1 \sqsubseteq x_2 \iff (x_1 = \perp) \vee (x_1 = x_2)]$

Likewise, it is necessary to order *den*. The domain *den* is ordered pointwise so for $d_1, d_2 \in den$

$$d_1 \sqsubseteq d_2 \iff \forall w \in {int_\perp}^2 \,.\, d_1 w \sqsubseteq d_2 w$$

[Sondergaard & Sestoft, 1992]

It can be concluded that the semantic description for the extended language is well defined since "for every expression e, the function $E[\![e]\!]$ is continuous and has a least fixed-point" [Sondergaard & Sestoft, 1992]. This least fixed point is represented by d, the denotation of the recursive program.

3.2 Semantics of Nondeterministic Programs

The example language is extended further through the introduction of the nondeterministic construct *amb*. Again, the syntactic description of the language first needs to be extended and is described as

$$
\begin{array}{lll}
pgm & \rightarrow f(x,y) == exp & \\
exp & \rightarrow int & \text{constant} \\
 & |x & \text{first variable} \\
 & |y & \text{second variable} \\
 & |exp + exp & \text{addition} \\
 & |exp\ amb\ exp & \text{nondeterministic choice} \\
 & |exp \rightarrow exp, exp & \text{conditional} \\
 & |f(exp, exp) & \text{function application}
\end{array}
$$

A possible denotational semantic description (given below) of the above language is described in detail in the next sections. The consequences of changing the semantics is also discussed.

Below, $d \in den$; $n \in int$; $u, v \in \mathcal{P}[int_\perp]$; $w \in \mathcal{P}[int_\perp]^2$; z_i abbreviates $E[\![e_i]\!]dw \in \mathcal{P}[int_\perp]$.

$$
\begin{array}{ll}
F[\![f(x,y) == e]\!] & = d(0,0)\ where\ rec\ d = E[\![e]\!]d \\
E[\![n]\!]dw & = n \\
E[\![x]\!]dw & = u \text{ where } (u,v) = w \\
E[\![y]\!]dw & = v \text{ where } (u,v) = w \\
E[\![e_1 + e_2]\!]dw & = \{\text{if } r_1 = \perp \text{ or } r_2 = \perp \text{ then } \perp \\
 & \quad \text{else } r_1 + r_2 \mid r_1 \in z_1 \wedge r_2 \in z_2\} \\
E[\![e_1\ amb\ e_2]\!]dw & = z_1 \text{ combine } z_2 \\
E[\![e_1 \rightarrow e_2, e_3]\!]dw & = \{\perp \mid \perp \in z_1\} \\
 & \quad \text{combine } \{r_2 \mid r_2 \in z_2 \wedge z_1 \backslash \{\perp, 0\} \neq \emptyset\} \\
 & \quad \text{combine } \{r_3 \mid r_3 \in z_3 \wedge 0 \in z_1\} \\
E[\![f(e_1, e_2)]\!]dw & = d(z_1, z_2)
\end{array}
$$

3.2.1 Power Domains

The denotational description for *amb* indicates that its value is either equal to the value of the first or second expression. But it would not be correct to make a choice at this point. Therefore, these values are combined to form a set of possible results through the use of *combine* which will be discussed in more detail below.

The following recursive program[2] includes the *amb* construct.

$$f(x, y) == x \rightarrow x + x, f(1\ amb\ 2, 0)$$

Due to the definition of *amb* above, (1 *amb* 2) evaluates to the set of values $\{1, 2\}$. When any one of these values is chosen and is bound to both occurrences of x (so as to implement definiteness), the program is described by the set of possible values $\{2, 4\}$. When each occurrence of x is independently bound to the value of either 1 or 2 the program is described by the set of possible values $\{2, 3, 4\}$. The results in each case are different, however, in both cases a set is needed to represent the results of evaluating the above program.

Since the value of a recursive program is now represented as a set of values, it is no longer correct to represent the value of *pgm* as an element of $int_\perp$. The values are sets of elements in $int_\perp$ and the power set $\mathcal{P}int_\perp$ contains all possible sets of answers.

However, $\mathcal{P}int_\perp$ needs to be ordered to ensure the theory of least fixed points still applies since the programs are still recursively defined. Three possible orderings on $\mathcal{P}int_\perp$ are considered. These orderings result in power domains (represented as $\mathcal{P}[int_\perp]$) [Sondergaard & Sestoft, 1992].

[2]Notice the slightly convoluted syntax, due to Sestoft's restricted notation in which the valuation function of a program always implicitly calls f with argument (0,0). In this example, $f(0, 0)$ must rewrite to $f((1\ amb\ 2), 0)$ which in turn evaluates to $x + x$ where x is bound to 1 *amb* 2

The crux of the difference between these orderings arises because all three power domains do not contain all the elements of $\mathcal{P}int_\perp$ [Main, 1987] [Clinger, 1982] [Sondergaard & Sestoft, 1992]. That is, a power domain is a subset of the powerset ($\mathcal{P}[int_\perp] \subseteq \mathcal{P}int_\perp$). These power domains (each which is a different subset of the power set) are defined below.

Let D be a domain, partially ordered by $\leq$. The relations H, S and $P \subseteq (\mathcal{P}D)^2$ all denoted by $\sqsubseteq$, are defined by

$$
\begin{array}{lll}
H : A \sqsubseteq B & \text{iff} & \forall a \in A \cdot \exists b \in B \cdot a \leq b \\
S : A \sqsubseteq B & \text{iff} & \forall b \in B \cdot \exists a \in A \cdot a \leq b \\
P : A \sqsubseteq B & \text{iff} & \forall a \in A \cdot \exists b \in B \cdot a \leq b \wedge \\
 & & \forall b \in B \cdot \exists a \in A \cdot a \leq b
\end{array}
$$

H, S and P represent the Hoare, Smyth and Plotkin orderings respectively [Sondergaard & Sestoft, 1992].

3.2.2 Types of Nondeterminism

As discussed in section 2.1 , different types of nondeterminism exist depending on how nontermination is handled. Consider the recursive program

$$f(x, y) == x \rightarrow x + x, f(0 \; amb \; 1, 0)$$

This may be defined to always terminate (set of results is $\{2\}$) or never terminate (set of results is $\{\perp\}$). Alternatively, neither termination nor termination may be favoured (set of results is $\{2, \perp\}$). These scenarios give rise to angelic nondeterminism, demonic nondeterminism and erratic nondeterminism respectively. The way in which the choice of power domain coincide with these concepts is discussed below.

In the Hoare power domain, $\{\perp\}$ represents nontermination. Any other

nonempty set $A \neq \{\perp\}$ represents a set of possible results. If A is a nonempty set then $A \cup \{\perp\} = A$. This means that the union of two sets results in a set of possible results where nontermination is impossible, as long as one of the sets is not equal to $\{\perp\}$.
A denotational description which uses the Hoare power domain describes a language which is globally angelic since nontermination is avoided if at all possible.

In the Smyth power domain, $\{\perp\}$ also represents nontermination and any non-empty set other than $\{\perp\}$ is a set of possible results. "The discrete Smyth power domain can be thought of as containing the finite subsets of $int \cup \{\perp\}$" [Sondergaard & Sestoft, 1992]. Any set containing $\perp$ is mapped to $\{\perp\}$.
This corresponds to globally demonic nondeterminism since nontermination is always chosen if possible.

In the Plotkin power domain, $\perp$ is treated like any other element. It no longer represents a set of values. Therefore, if $\perp$ is present in a set, it is a possible answer as are the other elements in the set. This means nontermination may result (if $\perp$ is chosen).This corresponds to erratic nondeterminism since nontermination is not favoured or avoided.

Notice the union of the sets in each case is different. This is handled in the denotational semantics by defining *combine* appropriately.

3.2.3 Singular versus Plural Semantics

Recall that without nondeterminism, $den = int_{\perp}{}^2 \rightarrow int_{\perp}$. However, when nondeterminism is present, the compound domain *den* is defined in terms of the chosen power domain (which establishes the type of nondeterminism present). In the above semantics of the extended nondeterministic example

language, *den* is defined as

$$\mathcal{P}([int_\perp])^2 \rightarrow \mathcal{P}[int_\perp]$$

Each function variable is bound to a set of possible values. This is referred to as plural semantics. As a result the language is not definite since each occurrence of a variable (such as x) may be bound to a different value.

However, an alternative way of defining *den* is

$$int_\perp{}^2 \rightarrow \mathcal{P}[int_\perp]$$

In addition, the denotational description of the function expression is defined as

$$E[\![f(e_1, e_2)]\!]\, dw = distributed\ combination\ \{d(u, v) \mid u \in z_1\ and\ v \in z_2\}$$

Each variable is bound to a single value. This is referred to as singular semantics. Note that the result is still a set of values and the language is definite.

3.2.4 Strict versus Nonstrict Semantics

As discussed in section 2.3.1, a function can be strict or nonstrict. A function f in the example language is strict if the function returns $\perp$ when either of the arguments is $\perp$. The addition and conditional functions are strict functions. If either of their arguments are undefined, the result is undefined.

The recursive function is defined as a nonstrict function in the above semantic description. This is because the function is evaluated regardless of whether u or v are equal to $\perp$. The denotational semantics in this case would normally be implemented using lazy evaluation techniques [Sondergaard & Sestoft, 1992].

However, the recursive function can be defined as a strict function instead. This can be represented by the denotation below which would normally be implemented using eager evaluation techniques.

$$d(u, v) = if\ u = \{\perp\}\ or\ v = \{\perp\}$$
$$\text{then } \{\perp\} \text{ else } (E[\![e]\!]d(u, v)) \cup (\{\perp\} \cap (u \cup v))$$

[Sondergaard & Sestof, 1990].

3.2.5 Parameter Passing techniques

There are a number of choices which need to be made when defining parameter passing techniques for nondeterministic languages. The type of nondeterminism present may be angelic, demonic or erratic. In addition, choices between singular and plural semantics, and strict and nonstrict functions need to be made.

The denotational descriptions describe global nondeterminism and a total of twelve different combinations of choices ($3 \cdot 2 \cdot 2$) are possible. Most importantly, the semantics of each combination is different [Sondergaard & Sestoft, 1992]. Clinger in his paper "Nondeterministic Call by Need is Neither Lazy Nor by Name" also highlighted the distinctions between these different semantics [Clinger, 1982].

Clinger concentrated on nonstrict (lazy) functions. He realized that because of these different combinations, call-by-need and call-by-name are no-longer illustrative of all possible parameter passing techniques. He suggested four of the twelve parameter passing techniques given by Sondergaard and Sestoft.

Even though at least twelve different parameter passing techniques are possible, the conflict between definiteness and unfoldability still exists in each. As indicated in section 2.3.3, the parameter passing techniques determine which properties are present (or vice versa). The effect on the language properties

of each of the twelve parameter passing techniques is tabulated in the figure below originally described by Sondergaard and Sestoft [Sondergaard & Sestoft, 1992].

	Definite	Indefinite
Unfoldable	None	Those having nonstrict plural semantics
Not Unfoldable	Those having singular semantics	Those having strict plural semantics

Figure 3.1: Properties of the Nondeterministic Language Variants.

A language implementor accustomed to deterministic languages, may not be aware of the large variety of parameter passing techniques possible in nondeterministic languages. Decisions regarding the call mechanisms and interpretation of *amb* have great impact on the properties of the resulting language. If these decisions are made without full knowledge of their impact, the resultant language may easily exhibit different properties to those expected.

3.3 Implementation Issues

The need for a semantic description to specify or verify a system seems to be evident and the insight and precision which the denotational semantic description brings to the analysis reveals the full extent of the subtleties.

However, note that the above method cannot be used to describe locally angelic or locally demonic systems. In order to successfully implement a globally angelic language, all evaluation paths which leads to nontermination must be avoided (if possible). However, the question of whether an evaluation path leads to nontermination is undecidable.

A globally angelic implementation would need to explore all possible paths simultaneously (or interleaved). Therefore, resources such as memory would

be in demand to search the state space to ensure termination (if possible). Locally angelic systems are easier to implement and, therefore, may be preferred.

For most language implementations, there is no direct mapping from the semantics of a program to the implementation. It must be recognized that semantics only capture some aspects of the implementations behaviour. That is, the semantics is an abstraction of the implementation. Therefore, during the mapping from a semantic description to an implementation, decisions not described by the semantics need to be made.

For example, the issue of fairness is not described by denotational semantics. At one extreme, choice can be implemented so that its first argument is always chosen. It can be argued that this is equivalent to nondeterministic choice where the first element is always chosen "randomly".

Most people accept that for a fair implementation of nondeterministic choice both values should be possible answers. However, there is no method of ensuring the choice is completely fair. Often both arguments are evaluated in parallel and the first to be computed is chosen, thus biasing short computations.

Even though both implementations differ, they are described by the same semantic description.

3.4 Alternative Semantics

A variation of the denotational semantics, as discussed above, results in a number of languages with different properties. However, figure 3.1 indicates that no mere change in the denotational semantic description, similar to those defined in the above section, will result in a language which is unfoldable and definite.

However, the idea of using and changing different semantics to overcome the conflict of unfoldability and definiteness has been pursued. Jackson and Burton suggest a semantics for a partially deterministic languages which is definite and unfoldable. Hughes suggests a natural semantics (a form of operational semantics [Nielson & Nielson, 1992]) for a unfoldable and definite nondeterministic language.

3.4.1 Partially Deterministic Languages

Jackson and Burton argue that Sondergaard and Sestoft are using "a fairly general form of nondeterminism" [Jackson & Burton, 1994]. They restrict the type of nondeterminism in their language, consequently, they are unfoldable and definite. "The only type of nondeterminism that occurs in our language is that the evaluation of some expressions may or may not terminate" [Jackson & Burton, 1994].

The domain of the language contains singleton sets or sets consisting of two elements. In the latter case, one of the elements is always equal to $\perp$ which represents nontermination.

This approach removes the ability to choose between two non-bottom values and avoids the dilemma caused by the conflict of unfoldability and definiteness.

However, nondeterministic choice is central to most discussion on nondeterminism and is also central to the discussion in this thesis. Therefore, we find this approach too limiting.

3.4.2 Hughes Natural Semantics

Hughes suggests a natural semantics which is built on top of Launchbury's natural semantics for laziness [Hughes & Moran, 1995]. This natural seman-

tics is used to model languages which contain a singular *amb*, to provide strong nondeterminism.

To achieve singularity, the language needs to be transformed into a normalized form in which "all applications are of an expression to a variable" [Hughes & Moran, 1995] and all variable names are distinct.

For example, the following simple expression

$$f(0\ amb\ 1)\ where\ f(x) = x + x$$

is normalized as

$$f(x) = (let\ y = x\ in\ y + y)$$

Although the first form is not unfoldable (if singularity is assumed $f(0\ amb\ 1) \neq (0\ amb\ 1) + (0\ amb\ 1)$) the second form "protects" itself by explicitly creating a single use of x and hence only one unique (singular) value for y.

Only a limited notion of unfoldability is possible. In some circumstances the need to remove the inconsistencies due to the conflict outweigh the added restrictions of normalization. In addition, algebraic transformations are applicable within this framework.

Future work on Hughes natural semantics will involve establishing an equational theory with the objective to allow reasoning about nondeterministic languages [Hughes & Moran, 1995]. Norvell and Hehner suggest a logical specification and the use of bunches which allows for the transforming of functional programs with similar semantics to those expressed using Hughes Natural Semantics [Norvell & Hehner, 1993].

However, it is important to note that even though inconsistencies do not exist, a normalized language which contains nondeterminism is not pure.

Chapter 4

Methods of Hiding Nondeterminism

Both Hughes [Hughes & Moran, 1995] and Jackson and Burton [Jackson & Burton, 1994] suggest languages in which disadvantages associated with strong nondeterminism have been addressed. The semantics of these languages are limited in respect to the type of nondeterminism or the extent of unfoldability allowable. In this chapter methods of hiding nondeterminism which do not alter the semantics of the basic deterministic functional language are discussed.

Two methods attempt to model the nondeterministic choice as having being made "outside" the program, the result of which is passed in as an explicit input. Another method which exploits weak nondeterminism is also discussed. All resultant languages, unlike strongly nondeterministic languages, are pure.

4.1 Streams

Input and output (I/O) is an accepted part of programming. However, the pure nature of a language may be forfeited if I/O constructs are added to a pure functional language.

For example, consider the pure function

$$f(x) = x + x$$

The output generated by this function is solely dependent on its input. Now allow the user to supply a number. This is obtained from the input stream by the function *get_char* [Hudak & Sundaresh, 1988]. If this value is added to x (rather than doubling x) the function is expressed as

$$f(x) = x + get_char$$

The function is no longer pure as the output is not solely dependent on explicit arguments to f. The result is also dependent on the value returned by get_char. In addition, the value of get_char may be different each time it is called. This complicates matters in particular when get_char is used more than once within a function.

Hudak [Hudak & Sundaresh, 1988] is one of many who have researched the problem of adding I/O to pure functional languages. The ideal is to implement efficient I/O whilst preserving the pure nature of functional languages which are referentially transparent and side-effect free.

Streams is one technique[1] used to implement I/O for pure languages. A stream is a lazy list of request or response objects. A program passes a stream of I/O requests to the operating system which are processed. The corresponding I/O responses are passed back to the main function which consumes these responses as explicit input. In this manner, the function acts like a "black box" where there is a direct correspondence between the input stream of responses and output stream of requests.

The responses are consumed lazily and requests can be output before the

[1] The discussion also applies to continuations and Hudak's *systems* model. It has been shown that simple translations exist between these techniques and streams [Hudak & Sundaresh, 1988].

first response has been consumed. This allows for the requests and responses to be interleaved. In this manner, an I/O request may be passed to the operating system (as output) before the I/O response (input) is consumed. Implementation of this method relies on lazy evaluation of streams.

The consequences of adding *amb* directly to a pure functional language are similar to those related to adding I/O constructs. Therefore, streams can be used to implement "amb-like" nondeterminism in pure languages such that functions behave as "black boxes".

In this manner, the benefits of nondeterministic choice are obtained whilst retaining referential transparency, definiteness and unfoldability in the language.

4.1.1 Definition of *Amb* using Streams

The main program accepts a stream of responses as input. A list of requests is generated as output. This list is solely dependent on the inputs. In Haskell [Hudak *et al.*, 1992] a program, which accepts a stream of responses and outputs a stream of requests, is said to engage in a dialogue. This is represented as follows.

$type\ Dialogue = [Response] \rightarrow [Request]$
$main :: Dialogue$

A request called *Amb* needs to be defined. This takes two arguments, the two values to choose between. The associated response is the actual value which has been chosen.

When a program containing an *Amb* request is evaluated, this request is passed to the operating system (output of the program). The choice is made by the operating system "outside" the program. The result is passed back

to the program as a response object of the response stream. The response stream is input to the program on which the program is directly dependent. Thus the resultant language is pure.

4.1.2 Semantics

Consider the following program written in Haskell where the !! operator selects the ith element of a list.

$$main\ resps = [\ Amb\ 1\ 2,\ Amb\ 1\ 2,\ AppendChannel\ "stdout"\ (resp!!1\ + resp!!1\ + resp!!2)]$$

A single response will never have more than one value and hence the program is definite. If a subsequent *Amb* request is identical to a previous *Amb* request (as above), each request is essentially different. This difference is conveyed by the position of the request within the request stream.

Notice that the first response is used twice and this results in singular semantics. To achieve plural semantics, a new response associated with an identical request needs to be used. This is the purpose of the second response and request in this example.

Consider the following program

$$main\ resps = [\ Amb\ 1\ 2,\ AppendChannel\ "stdout"\ (let\ e = resp!!1\ in\ e + e)]$$

The language is unfoldable since this is equivalent to

$$main\ resps = [\ Amb\ 1\ 2,\ AppendChannel\ "stdout"\ (resp!!1\ + resp!!1)]$$

In addition, the above program is referentially transparent in terms of Sondergaard and Sestoft's definition.

4.1.3 Classification

Semantically, the program's output depends solely on the stream of inputs and is referentially transparent, definite, and unfoldable. The resultant semantics is that of a deterministic program.

Operationally, the *Amb* request is output before the response is consumed. This allows for the choice of one of two values, at run time, without affecting the overall semantics of the program.

The use of responses (and the fact that each response can only have one value) essentially normalizes the language as specified by Hughes' natural semantics. Except a mechanism of ensuring unfoldability is not directly provided by the syntax of the primitive language but is achieved through the use of streams.

4.1.4 Disadvantages

The success of streams depends on the ability to interleave the responses and requests. However, this can also be a disadvantage since it can easily result in subtle bugs and deadlock [Hudak & Sundaresh, 1988].

In addition, this method requires choice specific input to be passed to the main function. That is, the position and value of each response within the stream is dependent on the requests. A subfunction can accept the stream of responses, access the required responses and return the unused tail of the stream which can be used by the next subfunction. This has an effect of single threading the evaluation of subfunctions. Consequently, it is difficult, if not impossible, to write concurrent programs using this method.

4.2 Burton's Method

Burton defines a data type called a decision and a function called *choice*. Together, these allow the functionality of *amb* and similar choice operators to be emulated [Burton, 1988]. The techniques employed by the streams method and Burton's method are similar. The pure nature of functional languages is preserved by passing an additional argument to the functions which will encapsulate information regarding choices.

In contrast to the streams method, Burton's method relies on an infinite binary tree of decisions to be passed as explicit input, initially unfixed. The tree can be split as often as needed and the new subtrees are passed to the subfunctions which require decisions. Hence concurrent or parallel evaluation of subfunctions is possible. This alternative approach is, therefore, very attractive for implementing parallel or concurrent systems as opposed to using streams.

4.2.1 Definition of *choice*

A decision can have the value of *one* or *two*. Otherwise, a decision is undefined ($\perp$). A tree of decisions is passed as an argument to the program since the program itself cannot construct a decision.

The following syntax and axioms used to define *choice*, given any type alpha.

$$choice : decision \times alpha \times alpha \longrightarrow alpha$$

$$choice(one, a, b) = a$$
$$choice(two, a, b) = b$$
$$choice(\perp, a, b) = \perp$$

Consider the following example

$$E(v) = choice(v, 2, 3) + choice(v, 1, 4) \tag{4.1}$$

The decision v, passed to expression E, can be equal to *one* or *two*. If v has never been used in a choice, it is called a free decision and is initially unfixed. When encountering a free decision, while making a choice, the implementation can nondeterministically select either the 2 or the 3, and simultaneously fix the value of the decision.

Once the value of the decision v is fixed, it cannot be changed. For example, if 2 is returned as the value of $choice(v, 2, 3)$, v is fixed to the value of *one*. Therefore, a subsequent evaluation of $choice(v, 1, 4)$ must produce 1 and the result of expression 4.1 is 3. Conversely, the implementation might have chosen 3 in the left expression thereby fixing v to *two*, and forcing the right expression to choose 4.

4.2.2 Semantics

Notice that a pure language, extended using *choice*, is definite. In other words, a variable will always have one and the same value (within scope). To illustrate this consider the following expression

$$f(choice\ (v, 3, 1))\ where\ f(x) = x + x \tag{4.2}$$

The possible values after evaluation are

- $3 + 3$ ($v = one$)
- $1 + 1$ ($v = two$)

Definiteness is not forfeited even if call-by-name evaluation techniques are used to evaluate the above expression. All evaluation techniques give rise to the above results. This is due to the fact that once a value has been assigned to a decision, it cannot be changed.

Since the language is definite and nondeterministic languages can either be definite or unfoldable, it may be tempting to conclude that the language cannot be unfoldable.

However, consider expression 4.2 above. It is unfoldable since

$$f(choice(v,3,1)) = choice(v,3,1) + choice(v,3,1)$$

This is always true, regardless of the evaluation technique used. Again this is because once v has been assigned a value, it cannot be altered.

By making the decision definite (can only have one value), the flexibility of choice still exists but the conflict between definiteness and unfoldability has been overcome.

Finally, using the definition of referential transparency in section 2.2.1, it can be deduced that as long as the *choice* expression is replaced by an equivalent value, the language is referentially transparent.

4.2.3 Classification

Although not vigorously proven, it can be concluded that a language extended using the *choice* function is referentially transparent, unfoldable and definite.

Semantically, the result of the program depends on the value of the decisions which are passed into the program as *a priori* data. The program is therefore deterministic since, for each tree of decisions passed as a parameter, only one result is possible.

Operationally, the decisions are not fixed before the program is executed. As a result, there is freedom to make on-the-fly choices during which time

the value of the decisions become fixed. These values represent the nondeterministic choices made by the *choice* function.

The fact that the language is semantically deterministic explains why Sondergaard and Sestoft's conflict does not arise.

4.2.4 Disadvantages

Since the program itself must not create its own decisions (otherwise strong nondeterminism results), these variables must be provided as *a priori* data, and passed as parameters into the computation. This requires an additional argument to each function.

Hughes and O'Donnell have expressed that this "plumbing" is difficult to keep consistent and is an added inconvenience [Hughes & O'Donnell, 1989]. This becomes an issue particularly during algebraic transformations. Essentially, by making the decisions definite and ensuring they are consistent, a form of normalization is present. This is not "hardwired" into the semantics of the language which results in both flexibility and a need for extra effort to keep the system consistent.

A similar problem exists if the method is used in a parallel system. A single decision may be referenced by more than one subprocess. Once the value of the decision is fixed by one subprocess, the other subprocess must be aware of the fact to ensure consistency within the system. Therefore, theory governing the use of shared variables in parallel or concurrent systems must be employed.

In addition Hughes and O'Donnell argue that functions such as *choice* are not bottom-avoiding unless the decision which is passed to the operation is unfixed. However, we believe freedom to make choices whilst retaining the mathematical nature of pure languages far outweighs the inconvenience of

maintaining the tree of decisions within an expression.

4.3 Monads

Monads allow for the expression of techniques using a higher level of abstraction. Continuations (a method similar to streams) can be implemented using monads. The use of monads in this manner does not decrease the disadvantages of single threading but Wadler claims that the advantage to using this technique is an increase in the readability of pure languages [Wadler, 1990].

Apart from using monads to implement methods similar to streams, Wadler suggests an interesting (but we believe limited) approach to incorporating nondeterminism into pure languages using monads [Wadler, 1990]. This approach is similar to the approach suggested by Hughes and O'Donnell [Hughes & O'Donnell, 1989].

Hughes and O'Donnell's idea is to add a set data type to a pure language. The *amb* operation is implemented through set union. Other language constructs are also described in terms of primitive set operations. The set is represented by one of its elements only.

Therefore, when the program terminates a single element form the set (which represents the possible values of the program) needs to be chosen. Strong nondeterminism is introduced at this point. Schreiner argues that this method "hides the problem more than actually solving it" [Schreiner, 1993]. It also differs from other methods of hiding nondeterminism discussed above which never ultimately result in strong nondeterminism.

Even though Wadler's monads supply a method of implementing Hughes and O'Donnell's technique, there is no way of hiding the strong nondeterminism which results. Whether singular or plural semantics are present (language is definite or unfoldable respectively) depends on the way the monads are

defined. If monads were used to implement Hughes and O'Donnell's method completely, angelic nondeterminism must be ensured.

This method is not as desirable as the other methods used to hide nondeterminism (described above) if a pure language is desired. However, this could be considered to be an alternative way of representing and reasoning about strongly nondeterministic languages.

This method is not restricted to singular semantics as is the approach using bunches discussed in section 3.4.2. However, we believe that this technique provides no more advantages than reasoning about strongly nondeterministic languages using a denotational semantic description. Therefore, we will not pursue this method further.

4.4 Peyton Jones' Method

In the above sections, methods of obtaining "amb-like" nondeterminism have been discussed. Exploitation of weak nondeterminism needs to be considered next. In particular the effects of weak nondeterminism on the properties of pure functional languages need to be ascertained.

Pure functional languages exhibit confluence. This means the redexes of a functional language can be evaluated in any order. This is precisely our definition of weak nondeterminism. Most importantly, it is possible to evaluate redexes in parallel. Peyton Jones and Eisenbach are just two of many who have used a method known as graph reduction to implement parallel evaluation of redexes [Peyton Jones, 1989] [Eisenbach & Sadler, 1988].

Weak nondeterminism can be exploited without adding any new constructs. The only variable parameter is the order of evaluation. Therefore, semantically, the language is deterministic. However, a weak form of nondeterminism is obtained operationally.

4.5 Comparison of Methods

When *amb* is added to a language, the language is semantically nondeterministic. This results in the conflict between unfoldability and definiteness (if the semantics or type of nondeterminism is not restricted). Semantically deterministic languages are referentially transparent, unfoldable and definite.

Burton's method and streams offer a combination of the two scenarios. Operationally the languages are nondeterministic but semantically they are deterministic. As a result, they exhibit the mathematical properties of pure functional languages and "amb-like" nondeterminism. Therefore, these methods can be used (in certain circumstances) to introduce and handle a form of nondeterminism without the problems associated with *amb*.

Peyton Jones' method does not offer "amb-like" or strong nondeterminism. However, weak nondeterminism is present operationally which can be exploited effectively. Semantically, the language is deterministic and exhibits the properties of pure functional languages.

Each different method should be used in specific circumstances, depending on the properties required for the final system. In the previous chapters, the methods have been described (using consistent terminology) and the merits of each have been indicated. Even though this assists the implementor in understanding the impact of nondeterminism, it still may be difficult to decide on the most appropriate method needed to implement a system.

We have developed the following table which summarizes the possible methods which can be used to implement nondeterministic systems.

		Operationally		
Semantically		Deterministic	Weakly Nondeterministic	Strongly Nondeterministic
	Deterministic	Pure Functional Languages	Peyton Jones' Method	Burton's Method / Streams
	Nondeterministic	X	X	Nondeterministic Constructs

Figure 4.1: Classification of Approaches to Nondeterminism.

This table has been organized in such a manner that the implementor need decide on two fundamental properties of the system only. Once an implementor knows whether the system should be semantically nondeterministic or deterministic and operationally deterministic or nondeterministic, the table can be used to identify suitable methods for implementing the nondeterministic system.

The implementor still needs to be fully aware of the ramifications of using the method (especially if strongly nondeterministic systems are required). Not all methods indicated will be totally appropriate in all cases. It may be necessary to adapt the indicated methods to suit the specific problem (examples of this are given in chapters 6 and 7).

Chapter 5

Parallel Processing and Functional Languages

Parallel processing encourages the use of nondeterminism or, in some cases, necessitates the use of nondeterminism. In order to demonstrate how the theory of nondeterminism can be applied, some issues relating to parallel functional processing are discussed.

This chapter gives an overview of parallel processing using functional languages. The following two chapters highlight concepts related to the implementation of functional parallel systems using the Linda paradigm.

5.1 Functional Parallel Processing

"Creating a parallel program depends on finding independent computations that can be executed simultaneously" [Scientific Computing Associates Inc., 1993]. The approach to this may vary, especially if functional languages are used to specify the parallel program.

The techniques of specifying parallel programs differ in terms of when independent tasks are identified and expressed. A programmer may explicitly indicate the independent tasks and the required synchronization and com-

munication between these tasks. This is referred to as explicit parallelism.

On the other hand, parallelism may be identified at compile time rather than design time. In this case, the programmer is exempt from identifying parallel tasks and considering the related concerns of synchronization and communication between tasks. This approach is referred to as implicit parallelism.

A third technique requires explicit annotation of programs evaluated using implicit parallelism. This can be considered to be a compromise between the techniques of implicit and explicit parallelism. These three techniques are outlined below.

5.1.1 Explicit Parallelism

This approach relies on identifying and specifying independent tasks at design time. This requires specifying the problem in terms of a parallel algorithm which meets the specification [Scientific Computing Associates Inc., 1993].

"The algorithm then has to be mapped onto the abstractions provided by the programming language" [Peyton Jones, 1989]. That is, a number of tasks (consisting of sequential activities) are identified which can be executed in parallel. These tasks cannot run in isolation. Therefore, the programmer needs to explicitly specify the "interfaces between tasks, which allow them to synchronized and communicate without hazards" [Peyton Jones, 1989].

Constructs which can be used to describe and control communication and synchronization between processes, are essential. These must be constructs of the parallel language used to specify the parallel system. The type of constructs present in the language depend on the communication technique employed.

The essence of explicit parallelism is the explicit specification of the paral-

lel tasks and synchronization and communication between processes, by the programmer. In addition it is the responsibility of the programmer to ensure the system does not deadlock or provide unacceptable results. This concern needs to be considered due to the nondeterminate nature of explicit parallel systems [Peyton Jones, 1989].

This approach is generally favoured by imperative programmers but has been considered by a number of functional system implementors. Examples of two such languages which facilitate this method of parallelism (both which use the Linda paradigm) are Lock and Jahnichen's Linda system [Lock & Jahnichen, 1990] and Jagannathan's TS language which is an extension of scheme [Jagannathan, 1991b].

5.1.2 Implicit Parallelism

A different approach, particularly advocated by functional programmers, is exploitation of implicit parallelism. A programmer does not identify or specify which tasks may be evaluated in parallel. As a result, the programmer does not need to consider concepts such as communication and synchronization of tasks. Therefore, no new parallel constructs are required or added to the functional language.

Instead, a complete "sequential" functional program is passed to the compiler. During the compilation stage, possible parallelism is identified. As a result subexpression constituting the program are evaluated in parallel. This is possible since the property of confluence allows for simultaneous evaluation of subexpressions without ill effect.

In addition, confluence ensures that implicit parallel systems are determinate [Peyton Jones, 1989]. Another feature of implicit parallelism is that no special measures need to be taken by the programmer to protect shared data access by concurrent or parallel processes since state is not explicitly altered.

At the implementation level , of course, parts of the graph are overwritten and locking mechanisms must prevent simultaneous updates.

Contrary to possible initial perception, even though a "sequential" program is evaluated (in parallel), some algorithms lend themselves to parallel evaluation more than others. Therefore, even though parallelism is not explicitly described, a good parallel algorithm is still essential for an efficient parallel implementation [Peyton Jones, 1989].

Peyton Jones argues that implicit parallel programming allows a programmer to "express the essential features of a parallel algorithm, without simultaneously having to detail the solution to a number of lower-level problems" [Peyton Jones, 1989].

Two examples of architectures used to evaluate functional languages in parallel are GRIP [Peyton Jones, 1989] and ALICE [Eisenbach & Sadler, 1988]. Both architectures use parallel graph reduction [Peyton Jones, 1987] to detect and control parallelism of parallel functional programs.

5.1.3 Implicit Parallelism and Annotations

Carriero and Gelernter expressed reservations about whether functional programming languages can be used successfully as parallel languages [Carriero & Gelernter, 1989]. This concern rests on the fact that these languages rely on the compiler and the run-time system to decide what to execute in parallel. They argue that implicitly parallel programs, currently do not and probably will never perform as well as explicitly parallel programs [Carriero & Gelernter, 1989] [Kale, 1989].

In response to Carriero and Gelernter, Kale states that this problem can be overcome by annotating functional programs [Kale, 1989]. These annotations are used to express additional information to assist in the efficient

implementation of the program in parallel. This places extra responsibility on the programmer, which is avoided if implicit parallelism is exploited.

This approach results in a compromise between explicit and implicit parallelism. Explicit constructs are used to describe, define and control parallel evaluation of the algorithm. However, the style of implicit parallelism is still adhered to.

One possible mechanism for expressing parallelism using annotations is to use lists to express tasks which can be evaluated in parallel. That is, all list elements are evaluated in parallel. The list is a primary data structure of functional languages and allows for implicit program annotation while the functional style of programming is preserved.

5.2 List Parallelism

Bird suggests a style of functional programming which expresses programs using the composition of a collection of higher-order functions [Bird, 1989]. Map and fold are two such higher-order functions from which all other (homomorphic) functions can be composed. This is the basis of the Bird-Meertens formalism which will be discussed in section 7.1.2 [Bird, 1988].

The map function, which we will express using the " * " symbol, applies a function to each element of a list. For example

$$f * [a_1, a_2, \cdots, a_n] = [fa_1, fa_2, \cdots, fa_n]$$

The fold function, which we will express using the "/" symbol is used to reduce a list using the reduction operator specified. For example

$$+/[a_1, a_2, \cdots, a_n] = a_1 + a_2 + \cdots + a_n$$

Programs written using this notation manipulate lists. From the Church-Rosser theorem, it is known that redexes, including list elements, can be

safely evaluated in parallel [Peyton Jones, 1987] provided the list is finite and all elements are well defined (we return to this point in section 5.2.2). Therefore, the above list notation can be used to indicate which subexpressions to evaluate in parallel.

5.2.1 List Notation

The list parallelism technique described in section 5.2 always evaluates a list in parallel if possible. Therefore, a list is no longer used purely as a data structure. A programmer cannot specify that the list should not be evaluated in parallel. This is a potential problem.

A solution to this problem is to use a separate list notation to specify lists which may be evaluated in parallel, if possible. This affords more flexibility but increases the use of annotations.

Normal list notation is used to express parallelizable lists. Every list expressed using this notation is evaluated in parallel if possible.

But what is "normal list notation"? A list will be represented using the notation $[e_1, e_2, \ldots, e_n]$. This is actually short hand for $e_1 : e_2 : \ldots : e_n : nil$ where : is the cons operator of a list. A single element can be added to an existing list using cons. That is

$$e_1 : L_1 = e_1 : e_2 : e_3 : e_4 : nil$$

where $L_1 = e_2 : e_3 : e_4 : nil$

An alternative notation, the append function $+\!\!+$, exists. Using this notation, list $[e_1, e_2, e_3]$ can be represented as $[e_1] +\!\!+ [e_2] +\!\!+ [e_3]$. This append function can be used to join two lists such as

$$[e_1, e_2, e_3] +\!\!+ [e_4, e_5, e_6] = [e_1, e_2, e_3, e_4, e_5, e_6]$$

Both notations are used in the literature and each can be written in terms of the other. The append notation allows for the subdivision of a list into sublists whereas cons allows for manipulation of a list through individual elements (in a sequential manner from head to tail).

As in the literature, both append and cons are used in this thesis. The notation is chosen to suit the nature of the discussion. It is suitable to use cons when implementation issues are discussed. On the other hand it is convenient to use append when "mathematically reasoning" about lists.

The functions *head* and *tail*, are two functions used to manipulate lists. The *head* function returns the first element of the list while the *tail* function returns the list minus the first element.

The list notation does not need to be changed drastically to indicate lists which must be evaluated sequentially. An annotated cons notation (:') is used explicitly to construct these lists. Therefore, the list $e_1 :' e_2 :' e_3$ is evaluated sequentially. For the purposes of this thesis, it is sufficient to use normal list notation and every list is evaluated in parallel if possible.

5.2.2 List Structure

When the elements of a list are evaluated in parallel, it is assumed that the list is finite and each element is defined. However, not all lists may be parallelized since it cannot always be assumed that these two conditions hold.

To illustrate this point, consider the following function applied to a list

$$head\ [e_1, e_2, e_3] = e_1 \tag{5.1}$$

The *head* function only requires the value of the first element within the list in order to return a value. If either e_2 or e_3 evaluate to $\perp$ this is inconsequential since these expressions are not evaluated by *head*.

If list parallelism is exploited, all list elements are evaluated, even if they are not required by the function applied to the list (unless speculative parallelism is employed). If either e_2 or e_3 evaluate to $\perp$, sequential evaluation of expression 5.1 terminates but parallel evaluation of this expression would not terminate. This is unacceptable. Therefore, the circumstances under which it can be safely assumed that the list is finite and each list element is defined need to be identified.

Consider the following expression.

$$length\ [e_1, e_2, e_3] = 3$$

The *length* function only evaluates the structure of the list. The value of each list element is not needed. The *sum* function on the other hand

$$sum\ [e_1, e_2, e_3] = e_1 + e_2 + e_3$$

needs to evaluate the structure of the list and the value of each element.

Burns classifies evaluators used to evaluate expressions according to the degree to which the evaluator evaluates the list [Burn, 1989]. Those evaluators which do not evaluate the list at all are classified as ξ_0 evaluators. ξ_1 evaluators reduce the list to weak head normal form. In WHNF, it is possible to determine whether the list is empty or non-empty without further evaluation. If the structure of the list needs to be evaluated (as in the length example), the evaluator is classified as ξ_2. Finally, evaluators which evaluate the structure of the list and the value of each element of the list are classified as ξ_3 evaluators.

For example cons (:) defined as

$$e_1\ :\ l_1 \tag{5.2}$$

where l_1 is a list and e_1 is a single value, requires ξ_0 evaluation of l_1. Evaluation of the *sum* function requires an ξ_3 evaluator.

If an ξ_3 evaluator is necessary to evaluate an expression containing a list, a finite list in which each element is defined is required for termination of the evaluation process. These constraints are not extra constraints placed on the list solely due to parallel evaluation of the list. Therefore, this list may be safely evaluated in parallel. Parallelization of a list evaluated by any other class of evaluator may lead to inconsistencies or inefficient evaluation.

A composition of functions can be applied to a list. The context of an expression can influence the optimal choice of evaluator required. The *sum* and cons may be combined as follows

$$sum\ (e_1\ :\ l_1)$$

An ξ_0 evaluator is required to evaluate $e_1 : l_1$. However, the *sum* function requires an ξ_3 evaluator. In complex expressions, the idea is to apply the strongest evaluator required to evaluate the whole expression. This evaluator should be applied as early as possible (i.e. it should be propagated as deeply into the subexpressions as possible) without doing evaluation that would never have been required.

For example, an ξ_0 evaluator could be used to evaluate l_1, but that would miss the opportunity for parallel evaluation that exists. An analysis could determine from the context that ξ_3 evaluation of the result will eventually be required and cannot be avoided. Therefore, by ensuring that cons requires ξ_3 evaluation, no extra evaluation is done. However, early realization of the need of an ξ_3 allows for parallel evaluation of the lists and hence optimization of the evaluation of the expression. Thus ξ_3 is the optimal evaluator for l_1 in this context.

A form of analysis called abstract analysis can be applied to complex functions to ascertain the optimal evaluator for the expression in context. A list can be safely and optimally evaluated in parallel if and only if the analysis

can infer that overall ξ_3 evaluation is necessary.

It must be noted that the problem of determining whether an arbitrary expression requires a particular evaluator is undecidable. The integrity of the system can be ensured as long as the analysis is conservative. That is, if ξ_3 evaluation is safe, but the analysis cannot prove this, the analysis reports that ξ_0 or ξ_1 evaluation is needed. This results in a decrease in potential parallelism.

5.3 Efficiency

When list parallelism is exploited, a list is only evaluated in parallel if analysis indicates that an ξ_3 evaluator is required. This results in a conservative approach to parallelism.

However, parallelism tends to be fine grained when list parallelism is exploited. A couple of issues need to be addressed in terms of improving or ensuring efficient implementations. These issues include the problem of controlling granularity and the degree of list parallelism exploited.

5.3.1 Granularity

If parallelism is too fine grained communication overheads overpower the benefits of parallelism. However, if the program is not sufficiently parallelized, speedup is reduced. Therefore, granularity must be balanced to ensure maximum speedup.

Exploitation of list parallelism allows the programmer to directly and implicitly manipulate granularity. Consider the following example

$$f_1 * \cdot f_2 * \cdot f_3 * [e_1, e_2, e_3]$$

Bird-Meertens formalism indicates that

$$f * [e_1, e_2, e_3] = [f\, e_1, f\, e_2, f\, e_3]$$

Therefore, it is possible to expand the above expression into either of the three expressions below.

1. $f_1 * \cdot f_2 * [f_3\, e_1, f_3\, e_2, f_3\, e_3]$
2. $f_1 * [(f_2 \cdot f_3)\, e_1, (f_2 \cdot f_3)\, e_2, (f_2 \cdot f_3)\, e_3]$
3. $[(f_1 \cdot f_2 \cdot f_3)\, e_1, (f_1 \cdot f_2 \cdot f_3)\, e_2, (f_1 \cdot f_2 \cdot f_3)\, e_3]$

The above expressions are listed in order of coarser granularity. In addition, the larger the list, the greater the degree of parallelism. These list properties allow the programmer to control the degree of parallelism and the granularity.

This technique relies on human intervention, which is preferred by some implementors. However, it could be automated through the use of analysis tools. The process of ensuring optimum granularity is a topic of on going research, particularly run-time analysis techniques [Aharoni *et al.*, 1992] and is not within the scope of this project.

5.3.2 Degree of Parallelism

Granularity is not the only factor which needs to be considered in relation to efficiency. Consider the following list operation

$$+/[e_1, \cdots, e_n]$$

Each element is evaluated in parallel. Initially it appears as if maximum parallelization exists. However, sequentialization is forced at the point of initiating the parallel tasks and collecting the results of each task evaluation. A similar situation was encountered by Kuchen and Gladitz while exploiting implicit parallelism using bags on MIMD machines [Kuchen & Gladitz, 1992].

A possible solution is to subdivide the list. The process of initiating the parallel tasks and collecting the results of the sublists occurs in parallel and less of a bottle neck exists at each "collection point".

However, this approach incurs additional synchronization and communication overheads. In addition, decisions regarding when and to which degree a list should be divided are critical. Therefore, the complexity of this subdivision could outweigh the benefits.

5.4 Expressibility

List parallelism is a form of implicit parallelism. Implicit parallelism focuses on decomposing a single task into a number of parallel tasks whereas explicit parallelism focuses on combining a number of independent tasks into a new task by executing them is parallel.

Carriero and Gelernter express the difference by considering how to parallelize a program which searches for an item in a database. They claim that implicit parallelism would be used to parallelize the individual search whereas explicit parallelism would be used to run many conventional searches in parallel [Carriero & Gelernter, 1989].

Carriero and Gelernter also claim that explicit parallelism is better since a larger number of cases can be expressed using this method alone. Unfortunately, it is difficult if not impossible to quantify the expressibility of explicit parallelism versus implicit parallelism. Different examples could be devised to express different merits for each method and both methods are advocated [Carriero & Gelernter, 1989] [Kale, 1989]. We believe this topic could be debated ad infinitum.

Most expressions can be expressed in terms of lists and list functions. For ex-

ample $e_1 + e_2$ can be expressed as $+/[e_1, e_2]$. The syntax chosen to represent a list and the degree to which lists are indicated using the list parallelism notation directly impacts on the granularity and hence efficiency of the program evaluation. This places restrictions on the expressibility of this approach.

However, despite this notation's limitations, it is flexible enough for our purposes. Sufficient parallelism can be exploited and controlled (even though the programmer needs to be aware of efficiency issues when constructing expressions) through the use of list parallelism.

Chapter 6

Parallel Functional Programming with Linda

As members of the Linda interest group at Rhodes University, we are interested in research relating to the extension and exploitation of Linda. In particular, we focus on determining whether Linda can be used to implement pure parallel functional languages.

We consider possible approaches to implementing functional parallel systems using Linda. In particular, the consequences of using each approach, in relation to the effect on properties of pure functional languages, are highlighted.

6.1 Linda

The Linda paradigm is used to extend existing sequential languages to produce parallel programming dialects. This is achieved by adding Linda operators to these sequential languages known as base languages. Typical examples of base languages are C, C++, and Modula-2 [Gelernter, 1988] [Carriero & Gelernter, 1989]. The nature and functioning of these base languages are unaffected by the introduction of Linda constructs which are orthogonal to these languages [Carriero & Gelernter, 1989].

A base language is used to express the sequential tasks which are composed into a coherent parallel program through the use of Linda constructs. Therefore, Linda introduces the notion of processes and controls the synchronization, creation and communication of processes written in a base language [Carriero & Gelernter, 1989].

Gelernter argues that the Linda paradigm addresses some of the commonly expressed problems of explicit parallel programming [Gelernter, 1988] [Carriero & Gelernter, 1989]. To fully understand the potential of Linda, and the drawbacks of the paradigm, we will briefly describe the basic concepts of the Linda paradigm.

6.1.1 Tuple Space

Processes which execute in parallel need to be synchronized and exchange data during execution. Synchronization and communication of Linda processes is achieved through the use of tuple space rather than message passing or shared variables [Carriero & Gelernter, 1989].

Tuple space is an unordered bag of data items known as tuples. Client processes may insert, remove and read tuples from tuples space, through the use of the primitives provided by Linda [Rehmet, 1994] [Carriero & Gelernter, 1989]. Communication is achieved between two processes if one process places a tuple into tuple space and another process removes this tuple from tuple space (not necessarily immediately) [Carriero & Gelernter, 1989].

Once a tuple has been placed into tuple space, there is no means of identifying its origin nor does the sender have any knowledge of the receiver. In addition, this tuple may be accessed by one or more processes, even after the execution of the process which placed the tuple in tuple space is complete. Therefore, communication is anonymous, asynchronous and spatially and temporally decoupled [Carriero & Gelernter, 1989].

6.1.2 Linda Constructs

As indicated above, the only Linda constructs which are needed are those which create processes, and insert or remove tuples from tuple space. Therefore, there are only six Linda constructs in total, all of which act on tuples. A summary of the information on tuples and Linda constructs available in [Rehmet, 1994], [de Heer-Menlah, 1991], [Carriero & Gelernter, 1989], [Gelernter, 1988], and [Scientific Computing Associates Inc., 1993] is given below.

"A Linda tuple consists of a set of typed fields, the first of which is generally regarded as a label or logical name" [Rehmet, 1994]. An example of a tuple is $("Value1", 30, x)$.

Tuples are placed into tuple space by using the *out* operation. If any fields are expression, they are evaluated by the process before the tuple is placed into tuple space. This includes variables which are replaced by their values. Once a tuple has been placed into tuple space, the process may continue with further computation. In other words, the process does not block after an *out* operation.

The *in* operation is used to retrieve tuples from tuple space. All *in* statements consist of the *in* operation followed by a tuple template. The tuple which is retrieved by the *in* statement is the first tuple in tuple space which matches the tuple template. If no matching tuple exists, the process which is retrieving the tuple blocks until a matching tuple is placed into tuple space.

When a tuple template is matched against a tuple in tuple space, each field must match exactly according to value and type [Rehmet, 1994]. Some fields of the tuple template may be formals. These are prefixed by question marks. In this case each field in the matching tuple, which corresponds to a formal, may have any value but must have the same type as the formal.

The structure of *rd* statements is similar to that of *in* statements. However, the *rd* operation is used to retrieve a copy of a tuple from tuple space. The original tuple remains in tuple space and can be retrieved by further processes.

The operations *rdp* and *inp* are similar to the *rd* and *in* operations. However, if a matching tuple is not present in tuple space, a boolean value (false) is returned instead of the tuple, and the process does not block.

Lastly, the *eval* operation is used to create a new process [Gelernter, 1988]. This operation places an active tuple, without evaluating the tuple fields, into tuple space [Rehmet, 1994] [Gelernter, 1988]. The evaluation of the active tuple is independent and parallel to that of the process which placed it into tuple space. After evaluation, the tuple is replaced by an ordinary data tuple (a passive tuple) which can be retrieved from tuple space.

6.1.3 Advantages and Criticisms

Some commonly accepted advantages of the Linda paradigm include those listed below

Ease of use is a direct result of exploiting tuple space, a distributed data structure. " A distributed data structure is valuable because it abstracts low-level details about process synchronization and communication to high-level algorithmic design issues involving data structure access and generation" [Jagannathan, 1991a].

Orthogonality allows for the clean integration of Linda commands into a base language. Orthogonality adds to the ease of use since the basic functioning of a familiar base language is unaffected.

Portability of Linda across a number of different architectures alleviates the necessity of re-writing programs for each new architecture. Linda is portable since the programs are logically independent of the system

architecture. Linda software manages the low-level details such as the physical location of processes and how data is moved between processes [Scientific Computing Associates Inc., 1993].

Dynamic Load Balancing is afforded at run-time due to the asynchronous and anonymous nature of communication between processes [de Heer-Menlah, 1991].

However, efficient implementation of Linda is potentially limited. Davidson argues that tuple space is a potential bottleneck [Davidson, 1989] and the associative matching of tuples can be expensive [de Heer-Menlah, 1991]. De-Heer-Menlah suggests a technique, implemented during compilation of Linda programs, which helps overcome these optimization problems [de Heer-Menlah, 1991].

Kahn and Miller are concerned by the increased complexity involved in compiling and optimizing large Linda systems. They suggest the use of a number of tuple spaces as a solution [Kahn & M.S.Miller, 1989].

Other weaknesses relate to the lack of protection in accessing tuple space and the inability to implement garbage collection techniques [Shapiro, 1989].

6.2 Explicit Parallelism using Linda

Initially one may consider exploiting explicit parallelism which results in a parallel language used to program parallel systems. The above discussion suggests that Linda is an ideal paradigm for this purpose. An existing functional language (the base language) could be extended by explicitly adding Linda constructs [Gelernter, 1988].

However, Lock and Jahnichen indicate that it is not as simple as it first appears. "If we allow these primitives to appear in functional expressions, we observe that the referential transparency is lost since the result of an *in* oper-

ation depends on the state of the tuple state, and the *out* operation changes this state" [Lock & Jahnichen, 1990].

Once again, the principle that the outputs of the function should depend solely on its explicit inputs is violated. Therefore, they propose a two-tier system. Only processes which map tuple space to tuple space may contain Linda constructs. These upper-level functions are not referentially transparent, but they encapsulate pure functional expression which in turn may not contain any Linda constructs.

Lock and Jahnichen's approach has the obvious disadvantage that the resultant language is not pure. It is only possible to rely on the mathematical properties of the separate pure functions. Therefore, algebraic transformations cannot be applied to the whole program with confidence.

It is important to note that Linda commands introduce nondeterminism into the base language. During an *in* or *rd* operation more than one matching tuple may exist, but only the first matching tuple may ever need to be retrieved. Therefore, these are nondeterministic constructs very similar to the *amb* construct.

In addition, not only may the choice differ during different executions, but the tuples which exist in tuple space (when a given process reaches a specific point) may also differ. This is because communication between processes is asynchronous, anonymous and temporarily decoupled.

Based on Sestoft and Sondergaard's findings either unfoldability or definiteness will be compromised when nondeterministic constructs are present in a language. Therefore, alternative ways of exploiting Linda without introducing nondeterminism or ways of hiding the resultant nondeterminism need to be considered.

6.2.1 Linda and Streams

Since Linda is essentially a technique based on Input/Output to tuple space, it should expected that previous work on I/O in functional languages could be adapted to once again provide the "output depends solely on inputs" model. According to the table in figure 4.1, Burton's method or streams could be used to obtained the desired results.

Streams are suited to this purpose since the number of items from which to choose need not be specified. This allows for the arbitrary choice of a matching tuple from tuple space. Tuple space is used to allow separate process to communicate [Hudak & Sundaresh, 1988].

Instead of adding Linda commands directly, requests corresponding to the Linda commands are used. For example an *In* request takes a tuple template as an argument. The resultant response is a matching tuple from tuple space. Even though more than one tuple may match, the response is fixed to this value and semantically, the process which issued the request is deterministic.

For this method to work successfully, each process should function as a separate program and consume its own list of responses (hence communicate with other processes through tuple space only). Therefore, the input to the main function is a list of streams.

At the top level of the functional program, the independent processes will be spawned, and each will be passed its own stream for communication with the external tuple space. A generalization of the system structure for a typical "farm of workers" is given below

$$\begin{aligned}
main &:: ([[Lresp]], [Resp]) \rightarrow ([[Lreq]], [Req]) \\
boss &:: ([Lresp], [Resp]) \rightarrow ([Lreq], [Req]) \\
worker &:: [Lresp] \rightarrow [Lreq]
\end{aligned}$$

$$\begin{aligned} main((l : ls, resp)) = \ & let\ (us, vs) = boss\ (l, resps) \\ & \quad others \ = map_in_parallel\ worker\ ls \\ & in\ (us : others, vs) \end{aligned}$$

The system structure is diagrammatically represented in figure 6.1 below.

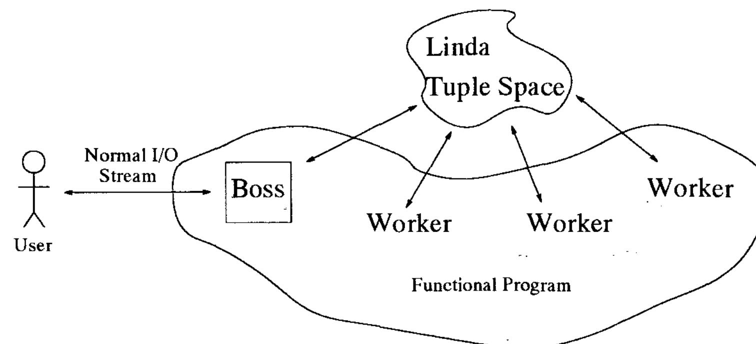

Figure 6.1: System Structure for a Typical "Farm of Workers".

The use of streams in accessing tuple space ensures that each process is definite. In many cases the system is referentially transparent as a whole. However, there are no restrictions which ensure that only programs which are "well behaved" can be written. In these cases the integrity of the entire system cannot be guaranteed (as in Lock and Jahnichen's approach). This is particularly true since communication is asynchronous and temporarily decoupled. This complicates the process of ensuring that the program as a whole is referentially transparent, unfoldable and definite.

6.3 Implicit Parallelism using Linda

Alternatively, instead of exploiting explicit parallelism, a form of implicit parallelism can be exploited in functional languages. This has been implemented successfully using parallel graph reduction techniques [Peyton Jones, 1989] [Peyton Jones, 1987] [Eisenbach & Sadler, 1988].

This technique is attractive. As indicated by the table in figure 4.1 in chapter 4, the exploitation of weak nondeterminism results in an operationally deterministic language and semantically weakly nondeterministic language. Ultimately, the language is still sequential (even though it is evaluated in parallel) and, therefore, is referentially transparent, unfoldable and definite.

It is therefore desirable to achieve similar results using the Linda paradigm. This would require evaluation of subexpressions in parallel through the use of Linda constructs at a level transparent to the user. Since the constructs are not added directly into the language, parallelism is implicit.

In particular Linda is suitable for the implementation of list parallelism. List elements (which are subexpressions of the functional program) are encapsulated into tuples. Subprocesses capable of evaluating functional expressions retrieve tuples from tuple space and output tuples containing the result of evaluation of the subexpression. In this manner, tuple space is used to control the parallel evaluation of list elements (at a level transparent to the user).

This can be used to extend existing parallel implementation which rely on a pool of tasks. All techniques and problems related to ensuring correct sharing of data (and perhaps parallel access to a graph of the program) still need to be considered and implemented.

Chapter 7

Optimization of Linda Implementation

An interesting consequence of using Linda, with its very general matching mechanism for withdrawing tuples from tuple space, is the ability to relax the order in which tuples are retrieved from tuple space. This can be advantageous in some circumstances and we consider this feature in conjunction with list parallelism.

For example, consider the function *search* used to search a list for a specific unique item. A possible implementation of the search involves dividing the list into sublists. Each sublist is searched in parallel (that is the search function is mapped onto each of the sublists).

$$\begin{aligned} search\ L &= or/ \cdot search * [L_1, L_2, \ldots, L_n] \\ &= or/\ [search\ L_1, search\ L_2, \ldots, search\ L_n] \end{aligned}$$
$where\ L_1 \ldots L_2\ is\ some\ partitioning\ of\ L$

In general, the order of the list must be preserved. These demands regarding specific sequencing of results introduce precedence constraints. We must wait for the result of searching sublist L_1 to appear in tuple space before we can retrieve the result of the search on sublist L_2 etc. even if other subsearch results are already available in tuple space.

But notice that in this example it is desirable to accept the results of each subsearch in any order. As soon as the first true answer is obtained, the rest of the answers may be ignored. There is no profit in waiting for the first result if the second result is available especially if this result is the answer to the search problem.

In general, list parallelism may be implemented more efficiently if results could be retrieved in any order, corresponding to when the results are available in tuple space. Technically, by relaxing the retrieval order, a bag of elements is retrieved from tuple space rather than a list.

For example, assume the elements of list $[e_1, e_2, e_3]$ are evaluated in parallel where e_1 evaluates to 1, and e_2 and e_3 evaluate to 2. The equivalent bag is represented as $\Lbag 1, 2, 2 \Rbag$ and its elements may be retrieved in any of the following possible orders.

- $1, 2, 2$
- $2, 1, 2$
- $2, 2, 1$

If the addition operation is used to reduce the result, there are no adverse consequences if this result is represented as a bag instead of a list. This is because the results of reducing of a list and bag under these circumstances are equivalent.

$$\begin{aligned} +/\ \Lbag 1,2,2 \Rbag &= 1+2+2 \\ &= 2+1+2 \\ &= 2+2+1 \\ &= +/\ [1,2,2] \end{aligned}$$

If the difference of the list elements is required, the expected result is

$$-/\ [1,2,2] = 1-2-2$$

However, a number of different results are possible if the difference of the equivalent bag is calculated instead.

$$\begin{aligned} -/ \wr 1,2,2 \int &= 2-1-2 \textit{ or} \\ &= 2-2-1 \textit{ or} \\ &= 1-2-2 \end{aligned}$$

Therefore, under these circumstances it is impossible to use a bag instead of a list without introducing nondeterminism.

We conclude that list parallelisation can be optimized by treating lists as bags but only under certain conditions. Our focus is to characterize those properties of lists and bags needed to facilitate the automatic detection of opportunities for this optimization.

7.1 Properties of Lists and Bags

The first step to characterizing lists and bags requires that a mathematical representation of these data structures is established. The Boom hierarchy, outlined below, is used to obtain this characterization. In addition, general list and bag operations and laws governing their application are defined using the Bird-Meertens formalism.

7.1.1 Boom Hierarchy

The Boom Hierarchy is a hierarchy of types fundamental to the Bird-Meertens formalism, a calculus of total functions [Hoogendijk & Backhouse, 1994] [Meertens, 1986] [Bird, 1988]. This hierarchy is used to mathematically describe data structures.

The most general data structure in the Boom hierarchy is the full binary tree. Given a set D, which does not contain the *nil* element, a domain of

binary trees containing leaves from D is formed using two constructors.

The first constructor is the embedding function. This maps a value from D into an element of the domain of full binary trees. Effectively, this creates leaf nodes from primitive elements in D. The second constructor is the join function which maps two full binary trees into a single full binary tree. That is, the join constructor effectively creates the internal nodes of the tree.

The set of binary trees has all the properties of a domain (refer to section 3.1.3). In addition, the domain of binary trees generated by the embedding and constructor functions is an algebra.

Each further level of the Boom hierarchy is a domain based on the binary tree. Each new level is obtained by successively placing further restrictions on the algebra of the previous level.

Non-full Binary trees (not every internal node has two children) form the next level of the hierarchy. These are obtained by adding *nil* to the set D. The full binary tree constructor (referred to as TREE-JOIN) is still used to construct these non-full binary trees.

The next level of the hierarchy consists of the domain of lists. The set D still contains *nil*, however an extra requirement of associativity is placed on the join function. This distinguishes lists from trees in that the "hierarchical depth" of trees is lost since the trees are flattened to form lists.

Bags and sets, the next levels in the hierarchy, are obtained by successively introducing the requirements that join is commutative, and idempotent [Hoogendijk & Backhouse, 1994] [Meertens, 1986]. The Boom hierarchy is summarized in the following table.

Structure	JOIN Constructor	JOIN Properties	Nil Present
Binary Tree	TREE-JOIN	-	No
Tree	TREE-JOIN	-	Yes
List	LIST-JOIN	Associative	Yes
Bag	BAG-JOIN	Associative, Commutative	Yes
Set	SET-JOIN	Associative, Commutative, Idempotent	Yes

Figure 7.1: Summary of Boom Hierarchy.

It is evident from the Boom hierarchy that the bag join constructor (BAG-JOIN) and list join constructor (LIST-JOIN) differ. BAG-JOIN is more restrictive than LIST-JOIN since it is associative and commutative rather than just associative. This difference alone is the distinguishing factor between these two data structures.

7.1.2 The Bird-Meertens Formalism

The Bird-Meertens formalism is a calculus of total functions. These functions can be applied to all structures in the Boom hierarchy. Bags and lists are particularly relevant to the rest of this chapter. Therefore, only these structures will be discussed further.

Firstly, we define the list and bag domains using the Bird-Meertens notation. The basis of domain construction is still the embedding and join constructors. We assume that the base set to which the embedding function is applied has the type α.

The domain of lists is created using the embedding function $[\cdot]$ and the join function $+\!\!+$ (LIST-JOIN in the previous section). The empty list $[\,]$ is the identity element of $+\!\!+$.

The domain of lists of type $[\alpha]$ is the algebra $([\alpha], +\!\!+, [\,])$ is an algebra gen-

erated under the embedding function $[\cdot] : \alpha \rightarrow [\alpha]$. More specifically, the algebra is a free monoid.

Similarly, the domain of bags is an algebra $(\wr\alpha\int, \uplus, \wr\int)$ generated under the embedding function $\wr \cdot \int : \alpha \rightarrow \wr\alpha\int$ where $\uplus$ is the bag join constructor and $\wr\int$ is the identity element of $\uplus$. More specifically, this algebra is a free *commutative* monoid [Bird, 1988].

Homomorphisms

Not only can we identify the domain of lists and bags as algebras, but other algebras are also of interest. For example, the set of integers together with the addition operator and the identity element 0 comprises an algebra (a free monoid).

Consider the length function $\# : [\alpha] \rightarrow N$ which determines the length of a list. The following three expressions are satisfied by the length function. Hence, # is referred to as a uniquely defined homomorphism from the monoid $([\alpha], +\!\!+, [\,])$ to $(N, +, 0)$ [Bird, 1988].

$$\begin{aligned} \#[\,] &= 0 \\ \#[a] &= 1 \\ \#(x +\!\!+ y) &= \#x + \#y \end{aligned}$$

Homomorphisms capture the notion of invariance under a divide-and-conquer or decompositional approach. The definition of the length function indicates that # is applied to each individual list element (in isolation). The corresponding values in the set of natural numbers are added to produce a final result.

However, not all functions applied to lists are homomorphisms. The function which removes adjacent duplicates from a list is not a homomorphism. Informally, this function needs to be applied to the list as a whole since

information about previous elements determine if the current element is a duplicate.

Bird-Meertens Definition of Map and Reduce

Map and reduce, informally defined in section 5.2, can be applied to the data structures characterized by the Boom hierarchy. The Bird-Meertens formalism allows for the formal definition of these functions.

Definitions of map and reduce over lists and bags are defined in appendix A.3. The essential informal meaning of these definitions is characterized in words below.

Consider expression $f * x$ where x is a list. If f is mapped onto:

- an empty list the result is an empty list.
- a singleton list, a singleton list which contains the result of applying the function to the original list element is returned.
- a list consisting of a number of items, the data structure is divided into two. The results of recursively applying the map function to each portion of the data structure are concatenated to form a new list.

This explanation is similar for $g * x$ where x is a bag.

Consider expression $\oplus/x$ where x is a list or bag. If the $\oplus$ operator is used to reduce:

- an empty bag or list, the result is the identity element.
- a singleton bag or list, the value contained within the data structure is returned.
- a bag or list consisting of a number of items, the data structure is divided into two. The results of recursively applying the reduce function

to each portion of the data structure are combined using the operator used to reduce the bag or list.

Assume that $f, g : \alpha \rightarrow \beta$. Note that $f*$ (defined in appendix A.3) is a homomorphism from $([\alpha], +\!\!+, [\,])$ to $([\beta], +\!\!+, [\,])$. In addition, $g*$ is a homomorphism from $(\wr\alpha\int, \uplus, \wr\,\int)$ to $(\wr\beta\int, \uplus, \wr\,\int)$. That is, $f*$ is a homomorphism on lists and $g*$ is a homomorphism on bags.

The operator $\oplus/$ in appendix A.3 (assumed *oplus* is associative and commutative, therefore $\oplus : \alpha \times \alpha \rightarrow \alpha$) is a homomorphism from $([\alpha], +\!\!+, [\,])$ to $(\alpha, \oplus, id_{\oplus})$ and a homomorphism from $(\wr\alpha\int, \uplus, \wr\,\int)$ to $(\alpha, \oplus, id_{\oplus})$. That is, $\oplus/$ is a homomorphism on a list and bag as defined in appendix A.3.

It is important to note that $\oplus/$ is a homomorphism on a list or bag only if the operator $\oplus$ is at least as algebraically rich as the join constructor of the data structure to which it is being applied. That is, $\oplus/$ is a homomorphism on lists only if $\oplus$ is at least associative. Similarly, $\oplus/$ is a homomorphism on a bag only if $\oplus$ is at least associative and commutative (as is the case in the definitions in appendix A.3).

Map and reduce have been targeted in this discussion since all homomorphisms can be represented as combinations of the map and reduce functions [Bird, 1988]. This removes the necessity to discuss other functions which are homomorphisms.

7.2 List to Bag Convertibility

The Bird-Meertens promotion rules (refer to appendix A.4) express the equations defining $f*$ and $\oplus/$ as identities between functions. These rules allow for the algebraic manipulation of expressions which contain bags or lists. However, for our purposes a further step needs to be taken. Rules need to

be defined which establish when a list can be converted to a bag.

If a list is replaced by a bag during optimization the function must still be well defined. It is safe to move from the list level of the Boom hierarchy to the bag level if the function being applied to the list is a bag homomorphism. In these cases the function is a homomorphism on lists as well as bags.

If the operator used for reducing the original list is associative and commutative, the list can be treated as a bag since the operator is also a homomorphism on the resultant bag.

In general, to substitute one data structure (list) by another (bag) requires moving between the levels in the Boom hierarchy. The reduction operator applied to the data structure in the current level needs to be at least as rich as that of the join operator of the data structure in the new level. This ensures that the substitution of one structure by the other will be safe.

Under the above conditions, the functions are well defined during optimization. The following conversion rule indicates when (and if) it is safe to use a bag instead of a list. This rule has been proved using the Bird-Meertens promotion rules (Refer to appendix A.4 and B.1).

Conversion Rule

Assume $\oplus$ is associative and commutative and *bag* is the bagify function (refer to appendix B for formal definition) which converts a list into a bag. Then

$$\oplus / = \oplus / \cdot bag \tag{7.1}$$

The rule states that any list can be represented as a bag as long as it is reduced by an associative and commutative operator. Otherwise, the effect of ignoring the order of the list is not canceled.

7.2.1 Bag Propagation

Most expressions consist of complex compositions of the map and reduce operators. Once again, as in section 5.2.2 the maximum benefit will accrue if the freedom to use a bag can be exploited as early as possible within the expression. The propagation rules indicate how the bagify function can be propagated through complex expressions.

The proof of propagation rule 1 is given in appendix B.2. Propagation rules 2 and 3 have not been formally proven here, but are straight forward.

Propagation Rules

Let

l	be any list
$\oplus$	be any associative and commutative operator
bag	be the bagify function used to convert a list to a bag
e	be a composite expression
f	be some arbitrary function
p	be some arbitrary predicate

1. $bag \cdot f* l = f* \cdot bag\ l$
2. $bag\ [f\ i \mid i \leftarrow e, p\ i] = [f\ i \mid i \ \leftarrow bag\ e, p\ i]$
3. $bag\ (e_1 \mathbin{+\!\!+} e_2) = bag\ (bag\ e_1 \mathbin{+\!\!+} bag\ e_2)$

The meaning of each of the above rules is expressed as follows

1. The list may be bagified before or after f is applied. Repeated application of this rule is expressed as $bag \cdot f_1 * \cdot f_2 * \cdots f_n * \cdot\ l = f_1 * \cdot f_2 * \cdots f_n * \cdot bag\ l$ where n is any natural number.
2. The bagify function may be propagated into a comprehension expression.

3. Two sublists may be individually bagified as long as the result of concatenating these resultant sub bags is bagified.

It is important to note that if a list is bagified, lists nested within the bagified list are unaffected. That is, the individual bag elements are not necessarily bags themselves. This is expressed as

$$bag\ [e_1\ , e_2] \neq [bag\ e_1\ , bag\ e_2]$$

Rule 3 initially looks incorrect. This is because list concatenation and bagify are being applied to a bag. This rule is expressed using this notation since a new bag structure is not implemented. A list is considered to be a bag if the order of the elements is not preserved. Two such lists are concatenated using $\#$. If the element order of a list has already been ignored, there is no further consequence of bagifying this list and is a legal operation.

7.3 Bag Analysis

Assume we have a simple language in which all list operations are restricted to map and reduce and the only two non-list operators are subtraction and addition. In addition, functions which take list arguments are included in the language. These functions consist of "allowable" combinations of the list and non-list operators, constants and list variables. We would expect that the above propagation and conversion rules would be sufficient to analyze expression of this simple language.

Consider the function below which takes a list as an argument. $\oplus$ represents a commutative and associative operator and $\ominus$ represents a non-commutative operator (may be associative).

$$\lambda x. \oplus / x - \ominus / x$$

The variable x appears more than once within the function. The convertibility rules above indicate that the argument bound to the first instance of x

can be treated as a bag. However, the second occurrence of x is reduced by a non-commutative operator and must be treated as a list. A single argument bound to x cannot be treated both as a list and bag. The safest choice is to treat the argument as a list.

This situation increases the complexity of deciding on list to bag convertibility. To simplify the implementation of the system, the decisions regarding bag convertibility in this situation may be left to the programmer. These decisions can be indicated through the use of annotations.

However, automatic analysis of the expressions would increase the usability of the system. This requires the use of what we have termed bag analysis, a backward analysis technique [Hughes, 1987].

7.3.1 Backward Analysis

Backward analysis is an analysis technique which calculates information from the context of an expression. This information is used for optimization purposes. "The flow of information is inwards, from enclosing expressions to enclosed sub-expressions, rather than outward as is the case in abstract interpretation" [Hughes, 1987].

In our case the information should indicate whether it is safe to treat a list argument as a bag instead of a list within expressions. This information is called the context.

7.3.2 Abstract Domain

Backward analysis relies on the use of domains. This ensures, via least fixed points, that the recursive programs are well defined (assuming all functions on the domain are continuous). Some context domains can be very complex, particularly if they are infinite and their generality and infinite nature make an accurate analysis undecidable.

To simplify the analysis, the use of a subdomain which describes the context sufficiently but "does not make more distinctions than necessary for the desired analysis" is advocated [Hughes, 1987]. This subdomain is called an abstract domain. With a finite abstract domain, the overall analysis becomes decidable although the results become "safe approximations". That is, the analysis may fail to find every opportunity for transforming a list to a bag, but those that it does detect will be accurate (or safe).

We suggest the use of an abstract domain which consists of two elements. The one element is B (bag) and the other is L (list). A list in context L must be evaluated as a list, otherwise it may be evaluated as a bag (the order of the list can be ignored).

The ordering on the domain is $B \leq L$. If a list can be bagified (context B) it must also be safe to treat the list as a list since context L is less specific than context B. However, the converse is not true. If a list is in the context L, it is undecidable whether it is safe to treat the list as a bag. So, in this context the safest choice is to evaluate the list as a list.

The element $ABSENT$ is usually present in a context domain. This is used to express the situation in which the list argument for which the context is being determined, is not present within an expression. $ABSENT$ is not an element of the abstract domain. In creating the abstract domain, $ABSENT$ is mapped onto the least point in the abstract domain which approximates it. In our case, $ABSENT$ is mapped to B.

7.3.3 Abstract Context Functions

During the analysis of an expressions, the context is transformed. Propagation rules (discussed in the section below) are used to propagate the context through the expression. However, each time a language primitive is located,

the manner in which this affects the transformation of the context needs to be ascertained.

Predefined abstract context functions, which correspond to the language primitives, provide the rules for such transformations. These rules together with the propagation rules can be used to define context functions for user defined functions.

A function within an expression has at least one associated context function. The number of abstract context functions associated with each function is dependent on the number of arguments to the function. Therefore, an abstract context function is denoted by $f\#_n$, where

f indicates the name of the associated function
$\#$ indicates the fact that the function is a context function
n indicates the position of the argument for which this context function is defined

Examples of a number of abstract context functions used for bag analysis of our simple language are described in figure 7.2. It is assumed that $\oplus$ is any associative and commutative operator and $\ominus$ is any non-commutative operator.

Context Function	Definition
$\oplus/\#$	$\lambda x \cdot B$
$\ominus/\#$	$\lambda x \cdot L$
$f * \#$	$\lambda x \cdot x$
$+\#_n$	$\lambda x \cdot x$ For all n
$-\#_n$	$\lambda x \cdot x$ For all n

Figure 7.2: Abstract Context Functions.

These abstract functions are monotonic and continuous on the abstract domains are sufficient for our purposes.

7.3.4 Propagating Contexts

For the following discussion, first order functions are assumed. Given an expression E, it is initially assumed that the context is L. This is propagated inwards to obtain the overall context of any list arguments E might contain.

"We define a function C such that, if x is a free variable of E, then $Cx[E]\alpha$ is the context propagated from E to x" [Hughes, 1987]. The context which is to be propagated inwards is referred to as α and must be an element of the abstract domain. E refers to any expression through which the context is propagated. The following rules are used to govern the process of propagating the context inwards through E for our simple language.

1. $Cx[x]\alpha = \alpha$
   Propagating a context through a simple variable (equal to the free variable of the analysis) preserves that context.

2. $Cx[E]\alpha = B$ if x does not occur free in E
   If x is absent from E, it is safe to assume that x can be bagified in this subexpression.

3. $Cx[fE_1 \cdots E_n]\alpha = Cx[E_1](f\#_1\alpha)\ \&\ \cdots\ \&\ Cx[E_n(f\#_n\alpha)]$

Rule 3 describes the propagation of context through a function application. An abstract context function, corresponding to the function and argument position is applied to the current context for each argument. Each resultant context is propagated through the corresponding function argument. The final resultant contexts are combined together (to calculate the overall context) using the & function defined as

&	L	B
L	L	L
B	L	B

This table implements a conservative approach. An overall bag context is permitted only if none of the "sub" contexts are equal to L.

Example

Consider the following simple, but illustrative, function where x and y are free list variables.

$$Combine(x, y) = (+/x) + (-/y) + (+/y)$$

The context of free variable x is calculated as follows (refer to figure 7.2 for definitions of all context functions).

$$\begin{aligned}
Cx[\,Combine\,]L &= \text{Definition of } Combine \\
&\quad Cx[(+/x) + (-/y) + (+/y)]L \\
&= \text{Rule 3 where } + \text{ is the function} \\
&\quad Cx[+/x](+\#_1 L) \ \&\ Cx[(-/y) + (+/y)](+\#_2 L) \\
&= \text{Definition of } +\#_n \\
&\quad Cx[+/x]L \ \&\ Cx[(-/y) + (+/y)]L \\
&= \text{Rule 2} \\
&\quad Cx[+/x]L \ \&\ B \\
&= \text{Rule 3 where } +/ \text{ is the function} \\
&\quad Cx[x](+/\#L) \ \&\ B \\
&= \text{Definition of } +/\# \\
&\quad Cx[x]B \ \&\ B \\
&= \text{Rule 1} \\
&\quad B \ \&\ B \\
&= \text{Definition of } \& \\
&\quad B
\end{aligned}$$

This analysis tells us that before passing an actual argument to bind to x, it is safe to bagify the list (when retrieved from tuple space).

Now the analysis is repeated to see whether the argument to y can be bagified. The context of free variable y is calculated as follows

$$
\begin{aligned}
Cy[Combine]L &= \text{Definition of } Combine \\
& \quad Cy[(+/x) + (-/y) + (+/y)]L \\
&= \text{Rule 3 where } + \text{ is the function} \\
& \quad Cy[+/x](+\#_1 L) \ \& \ Cy[(-/y) + (+/y)](+\#_2 L) \\
&= \text{Definition of } +\#_n \\
& \quad Cy[+/x]L \ \& \ Cy[(-/y) + (+/y)]L \\
&= \text{Rule 2} \\
& \quad B \ \& \ Cy[(-/y) + (+/y)]L \\
&= \text{Rule 3 where } + \text{ is the function} \\
& \quad B \ \& \ Cy[-/y](+\#_1 L) \ \& \ Cy[+/y](+\#_2 L) \\
&= \text{Definition of } +\#_n \\
& \quad B \ \& \ Cy[-/y]L \ \& \ Cy[+/y]L \\
&= \text{Rule 3 where } -/ \text{ is the function} \\
& \quad B \ \& \ Cy[y](-/\#L) \ \& \ Cy[+/y]L \\
&= \text{Definition of } -/\# \\
& \quad B \ \& \ Cy[y]L \ \& \ Cy[+/y]L \\
&= \text{Rule 3 where } +/ \text{ is the function} \\
& \quad B \ \& \ Cy[y]L \ \& \ Cy[y](+/\#L) \\
&= \text{Definition of } +/\# \\
& \quad B \ \& \ Cy[y]L \ \& \ Cy[y]B \\
&= \text{Rule 1} \\
& \quad B \ \& \ L \ \& \ B \\
&= \text{Definition of } \& \\
& \quad L
\end{aligned}
$$

The second argument of *Combine* must be treated as a list since it is reduced using an operator which is not commutative and associative in one position (this information is captured within the abstract context functions). However, the first argument can safely be treated as a bag. It can be deduced that the context function for *Combine* is

$$\begin{aligned} Combine\#_1 &= \lambda x \cdot B \\ Combine\#_2 &= \lambda x \cdot L \end{aligned}$$

Consider the following function which uses the function *Combine*, where x is a list variable.

$$Same(x) = Combine(x, x)$$

The context of the first (and only argument) for *Same* is L (as expected) since

$$\begin{aligned} Cx[Same]L = & \text{ Definition of } Same \\ & Cx[Combine(x,x)]L \\ = & \text{ Rule 3 where } Combine \text{ is the function} \\ & Cx[x](Combine\#_1 L)\ \&\ Cx[x](Combine\#_2 L) \\ = & \text{ Definition of } Combine\#_n \\ & Cx[x]B\ \&\ Cx[x]L \\ = & \text{ Rule 1} \\ & B\ \&\ L \\ = & \text{ Definition of } \& \\ & L \end{aligned}$$

7.3.5 Extensions to Bag Analysis

The above bag analysis can only be used to analyze functions which directly manipulate lists using map and reduce. However, a function may be defined

recursively in such a way that the individual list elements are directly manipulated. The analysis needs to be extended to handle these types of functions.

The length function is defined recursively below.

$$length(x) = \text{if } null\ x \text{ then } 0 \text{ else } 1 + length\ (tail\ x)$$

This function manipulates individual list elements. The function *tail* is used to return the rest of the list minus the first element. The function *null* determines if the list is empty.

We need to extend the language by adding functions which are used to isolate and manipulate list elements. To extend the bag analysis, their corresponding context functions are defined as

Context Function	Definition
$head\#$	$\lambda x \cdot L$
$tail\#$	$\lambda x \cdot x$
$:\#_1$	$\lambda x \cdot L$
$:\#_2$	$\lambda x \cdot x$
$null\#$	$\lambda x \cdot B$

Figure 7.3: Abstract Context Functions for Primitive List Functions.

In addition, many recursive functions use if statements. The following rule governs the process of propagating the context inwards through an if statement.

$$Cx[\text{ if } e_1 \text{ then } e_2 \text{ else } e_3]\alpha = Cx[e_1]\alpha \,\&Cx[e_2]\alpha \,\&Cx[e_3]\alpha$$

This is a conservative rule since if any subexpression of the if then statement return an L context, then the whole expression returns an L context.

An analysis of the length function (using fixed point iteration to determine the context of the recursive functions) returns a context of B as expected. However, analysis of the *sum* function defined recursively below returns a context of L.

$$sum(x) = \text{if } null\ x \text{ then } 0 \text{ else } head\ x + sum\ (tail\ x)$$

Even though a safe result is returned, the analysis fails to recognize possible parallelisation where the original analysis of sum expressed using $+/$ succeeds. The ability to decide on the context for the whole list is lost when list elements are manipulated individually by *head*, and non-list operators. The *head* function also forces a context of L. Therefore, this method of bag analysis is not particularly successful for recursively defined functions.

Now consider an expression which contains nested lists expressed using the high level list notation.

$$f(x) = \oplus/[\oplus/x, \ominus/[1,2,3], \ominus/x]$$

Each time a new list level is entered, previous information (context) must be ignored since each level is totally isolated form previous and future levels (as was discussed in section 7.3.4 on propagation rules). The sublist is expressed as individual elements joined with cons (:). The context function of this list primitive correctly indicates that all previous context must be ignored at this new list level. Therefore this primitive is still important even when the high level list notation is adopted.

7.4 Implications of Optimization

The basis of the optimization technique is the decision of whether a list will ultimately be reduced using an associative and commutative operator. The programmer may find this relatively easy to decide. However, if the optimization process is automated, the compiler needs to decide if an operator may be associative and commutative. This is undecidable in general(refer to

appendix C for the outline to a proof).

When the nature of a function or operator is unknown and cannot be calculated, the analysis must still be safe. Although undecidable, it is still possible to construct a conservative algorithm which may find some cases and which assumes non-commutativity in all others.

If a function on a list is not a homomorphism on a bag, the list elements may still be evaluated in parallel. However, the resultant list must be reconstructed preserving the order before the function is applied to the list. A particular example may be the function *head* which returns the first element of a list. Therefore, a programmer needs to optimize the use of associative and commutative operators to maximize the efficiency of the system. This can be considered to be a restriction on the expressibility of the system.

Bag analysis for high level list constructs is successful whereas analysis of recursive function which manipulate the individual elements of lists is less successful. This is because vital information regarding list structure and context is lost if the list is "flattened" when individual list elements are manipulated using non-list operators.

The implications of this observation are that the use of high level constructs to exploit lists, using Bird-Meertens list notation, is preferable to recursive functions which directly manipulate individual elements. This observation further supports the choice the Bird-Meertens list notation made in section 5.2.

In the circumstances where a bag may be used instead of a list, there always exists a node in the graph which contains an associative and commutative reduction operator. This cancels out the effects of ignoring the order of the list. In addition, it has been ascertained that the bag can be successfully propagated through the subgraph between this node and the list which is

to be bagified. However, it is important to note that this subgraph is still subject to the rules described by Sondergaard and Sestoft (refer to chapter 2).

If any new functions are inserted into this region of the graph during compilation and optimization of the program, it must be realized that unfoldability of these functions cannot be assured. Consider the following expression

$$+ / \cdot f_3 * \cdot f_2 * \cdot f_1 * l \tag{7.2}$$

A bag may be introduced at the point where $+/$ is used to reduce the list. This bag may be propagated down to l and this indicates that a bag may be used instead of the list. Therefore the expression is now expressed as

$$+ / \cdot f_3 * \cdot f_2 * \cdot f_1 * \cdot bag\ l \tag{7.3}$$

At this point, expression 7.2 and 7.3 are equivalent. However, consider the following identity function

$$id\ x = if\ (x = x)\ then\ x\ else\ \bot$$

Now insert this function between f_3 and f_2 such that

$$+ / \cdot f_3 * \cdot id \cdot f_2 * \cdot f_1 * \cdot bag\ l \tag{7.4}$$

Initially, it appears as if this is legal since expressions 7.2 and 7.3 are equivalent. However, notice that expression 7.4 is no longer unfoldable. Therefore it is not equal to the expression $+/ \cdot f_3 * \cdot id \cdot f_2 * \cdot f_1 * \cdot l$ which has not been optimized.

This does not mean that the technique is incorrect. However, a compiler writer needs to be aware of these nondeterministic regions and must treat them carefully.

Chapter 8

Conclusion

This thesis has investigated aspects of nondeterminism in relation to pure functional languages. In particular, we ascertained the problem related to introducing nondeterminism into functional languages. In chapter 2 the essential notions of weak and strong nondeterminism and Sondergaard and Sestoft's analysis allowed us to precisely characterize the problem as one of a conflict between definiteness and unfoldability.

In chapter 3 three further properties of strongly nondeterministic languages namely, strict/nonstrict functions, singular/plural semantics, and type of nondeterminism were identified. A denotational semantics for these strongly nondeterministic languages was considered since this allowed for the precise specification of how the resultant semantics relied on the combination of the above three properties. This characterization increased the awareness of the wide (and possibly unexpected) spectrum of available parameter passing techniques and the consequences of employing each technique.

Methods used to hide nondeterminism were defined in chapter 4. These methods result in pure functional languages. However, the investigation of these techniques indicated that purity is always obtained at a price. Implicit restrictions, similar to those imposed explicitly on languages expressed using Hughes natural semantics, always exist.

We devised a classification table (figure 4.1 in chapter 4) in which all nondeterministic methods discussed are classified. This provides a mechanism which allows implementors to identify a suitable technique of implementing functional parallel systems. They need only consider whether the implementation should result in an (operationally and semantically) deterministic or nondeterministic system.

Implementation of a parallel system using functional languages may initially seem impossible if a pure system is required. The main restriction is a need for the system to be semantically deterministic. The rest of this thesis focused on ascertaining if functional languages are appropriate for functional language implementation. This discussion focused on the Linda parallel processing paradigm.

The discussion in chapter 6 highlighted the strongly nondeterministic nature of functional languages extended using Linda directly. Even a system implemented using an explicit parallelism technique, but which ensured semantic nondeterminism, was shown to be unsuitable due to the inability to ensure that a program is pure as a whole.

As an alternative to explicit parallelization, we exploited weak nondeterminism in chapter 6 which was also indicated as a suitable method for the implementation of pure functional parallel systems. We defined list parallelism, a form of implicit parallelism, which offers a method of implementing weak nondeterminism easily tailored to exploit the properties of Linda. This method of exploiting list parallelism can be generalized to any system implemented using a distributed data structure and is not restricted to Linda implementations.

In chapter 7 optimization techniques which involved using bags instead of lists were used to increase opportunities for parallelism. The context in

which such optimizations were applied was restricted through the use of bag analysis and conversion and propagation rules. In this manner overall determinism for the system was ensured even though Linda produces results in a nondeterministic order.

Bag analysis allowed for analysis of expressions which manipulated lists using high level list operators such as map and reduce. However, we concluded that bag analysis is not effective for analysis of recursive programs which manipulate individual list elements explicitly. This further supports the adoption of list parallelism expressed using the Bird-Meertens notation.

There are many pitfalls an implementor has to be aware of during the implementation of a nondeterministic (or potentially nondeterministic) system. This thesis highlights the wealth of techniques which can be used both directly and in adapted form to successfully implement applications such as parallel processing systems. We have been able to show that there is a definite place for functional implementations of these types of systems.

Appendix A

Definitions from the Bird-Meertens Formalism

A.1 Notation

Bird uses the symbols $+\!\!+$ (list concatenation) and $\uplus$ (bag concatenation) which correspond to LIST-JOIN and BAG-JOIN respectively.

The functions $[\cdot]$ and $\wr \cdot \int$ are the embedding constructors for lists and bags respectively. Therefore,

$$
\begin{aligned}
[\cdot]\, a &= [a] \\
\wr \cdot \int a &= \wr a \int
\end{aligned}
$$

The empty bag and list are represented by $\wr\,\int$ and $[\,]$ respectively.

A.2 Function Types

The functions f and g defined within these appendices have the following types

$f : \alpha \rightarrow \beta$

$g : \alpha \rightarrow \beta$

$f* : [\alpha] \rightarrow [\beta]$

$g* : \wr\alpha\int \rightarrow \wr\beta\int$

$\oplus : \alpha \times \alpha \rightarrow \alpha$ ($\oplus$ is assumed to be associative and commutative).

A.3 Definition of Map and Reduce

Figure A.1 defines map and reduce (for both bags and lists) using Bird's Notation [Bird, 1988].

	Lists	Bags
Map	$f * [] = []$ $f * [a] = [fa]$ $f * (x + y) = (f * x) + (f * y)$	$g * \wr\int = \wr\int$ $g * \wr a\int = \wr ga\int$ $g * (x \uplus y) = (g * x) \uplus (g * y)$
Reduce	$\oplus/[] = id_{\oplus}$ $\oplus/[a] = a$ $\oplus/(x + y) = (\oplus/x) \oplus (\oplus/y)$	$\oplus/\wr\ \int = id_{\oplus}$ $\oplus/\wr a\int = a$ $\oplus/(x \uplus y) = (\oplus/x) \oplus (\oplus/y)$

Figure A.1: Definition of Map and Reduce.

A.4 Promotion Rules

Figure A.2 defines the promotion rules for map and reduce. The two join rules are referred to as the map and reduce promotion rules respectively. The function K below is the constant valued function [Bird, 1988].

Rules	List	Bag
Empty Rules	$f * \cdot K[\,] = K[\,]$ $\oplus/ \cdot K[\,] = id_\oplus$	$f * \cdot K \wr \int = K \wr \int$ $\oplus/ \cdot K \wr \int = id_\oplus$
One-point rules	$f * \cdot [\cdot] = [\cdot] \cdot f$ $\oplus/ \cdot [\cdot] = id$	$f * \cdot \wr \cdot \int = \wr \cdot \int \cdot f$ $\oplus/ \cdot \wr \cdot \int = id$
Join rules	$f * \cdot +\!\!+/ = +\!\!+/ \cdot (f*)*$ $\oplus/ \cdot +\!\!+/ = \oplus/ \cdot (\oplus/)*$	$f * \cdot \uplus/ = \uplus/ \cdot (f*)*$ $\oplus/ \cdot \uplus/ = \oplus/ \cdot (\oplus/)*$

Figure A.2: Bird-Meertens Promotion Rules.

Appendix B

Conversion Rule Proofs

While the Bird-Meertens formalism is concerned with the independent rules for lists and bags, here we examine our extensions for convertibility between list and bag structures.

The bagify function *bag*, used in the proofs below, is defined as

$$bag = \uplus/ \cdot \Lbag \cdot \Rbag *$$

The proofs below also rely on a general rule of map. This is the distributivity rule and is defined (for any functions g and f) as

$$f * \cdot g* = (f \cdot g)*$$

B.1 Reduction Conversion Rule

$$\begin{aligned}
\oplus/\, bag &= \text{definition of bag} \\
&\quad \oplus/ \cdot \uplus/ \cdot \Lbag \cdot \Rbag * \\
&= \text{reduce promotion} \\
&\quad \oplus/ \cdot (\oplus/) * \cdot \Lbag \cdot \Rbag * \\
&= \text{distributivity of map} \\
&\quad \oplus/ \cdot (\oplus/ \cdot \Lbag \cdot \Rbag) * \\
&= \text{one-point rule}
\end{aligned}$$

$$\oplus/ \cdot (id)*$$
$$= \oplus/$$

B.2 Bag Propagation Rule through Map

$f * \cdot bag =$ definition of bag

$f * \cdot \uplus / \cdot \wr \cdot \int *$

$=$ map promotion

$\uplus/ \cdot (f*) * \cdot \wr \cdot \int *$

$=$ distributivity of map

$\uplus/ \cdot (f * \cdot \wr \cdot \int)*$

$=$ one point rule

$/uplus \cdot (\wr \cdot \int \cdot f)*$

$=$ distributivity of map

$\uplus/ \cdot \wr \cdot \int * \cdot f*$

$=$ distributivity of map

$\uplus/ \cdot \wr \cdot \int * \cdot f*$

$=$ definition of bag

$bag \cdot f*$

Appendix C

Undecidability of the Commutivity Problem

A function $f(D_1, D_1) \rightarrow D_2$ is commutative if and only if $f(a, b) = f(b, a)$. Can the question "Is this function commutative?" always be answered for all functions? If this is so, the problem (which we call the commutivity problem) is decidable.

We prove that this problem is undecidable by giving one counter example.

C.1 Proof

Consider the following function $f(D_1, D_1) \rightarrow D_2$

$f(x, y) =$ if $x<$ y then
  if *halt* $< R, X >$ then 1
  else 0
 else 1

The function *halt* determines if an arbitrary program R terminates given the input X. That is, the halting problem (which is undecidable) [Harel, 1992] has been embedded into the function f.

Without loss of generality, we restrict ourselves to cases where $a > b$.
The function f is commutative if

$f(a,b) = f(b,a) = 1 \qquad \forall a \in D_1 \wedge b \in D_2 \ , a > b$

otherwise

$f(a,b) = 1 \neq f(b,a) = 0 \quad \forall a \in D_1 \wedge b \in D_2 \ , a > b$

Therefore, the problem of commutivity is reduced to deciding if $f(b,a)$ returns a 0 or 1.

If we assume the commutivity problem is decidable then $IsComm(f)$ is $true$ when $f(b,a)$ returns 1. Conversely $IsComm(f)$ is $false$ when $f(b,a)$ returns 0 (where $IsComm$ is a function which can tell if a function is commutative).

Notice that the halting problem can be described in terms of $IsComm$ and f.

```
my_halt(R,X) = if IsComm(f) then
                    R halts on X
                 else
                    R does not halt on X
```

Consequently the commutative problem is undecidable, otherwise we have solved the halting problem (which we know is undecidable) by constructing the function f.

References

Aharoni, G., *et al.* 1992. A Run-time Algorithm for Managing the Granularity of Parallel Functional Programs. *Journal of Functional Programming*, **2**(4), 387 - 405.

Bird, R.S. 1988 (Sept.). *Lectures on Constructive Functional Programming.* Lecture notes PRG-69. Oxford University Computing Laboratory.

Bird, R.S. 1989. Algebraic Identities for Program Calculation. *The Computer Journal*, **32**(2), 122 - 126.

Burn, G.L. 1989 (June). Overview of a Parallel Reduction Machine Project II. *Pages 385–396 of: PARE 89, Springer-Verslag LNCS No 365.*

Burton, F.W. 1988. Nondeterminism with Referential Transparency in Functional Programming languages. *The Computer Journal*, **31**(3), 243–247.

Carriero, N., & Gelernter, D. 1989. Linda in Context. *Communications of the ACM*, **32**(4), 444 - 458.

Clinger, W. 1982 (August). Nondeterministic Call by Need is Niether Lazy Nor by Name. *Pages 226 - 234 of: Proc. 1982 ACM Symp. LISP and Functional Programming.*

Davidson, C. 1989. Technical Correspondence. *Communications of the ACM*, **32**(10), 1249 - 1251.

de Heer-Menlah, F.K. 1991 (October). *Analyzing Communication Flow and Process Placement in Linda Programs on Transputers.* Master's The-

sis TD91/17. Computer Science Department, Rhodes University, South Africa.

Eisenbach, S., & Sadler, C. 1988. Parallel Architecture for Functional Programming. *Information and Software Technology*, **30**(6), 355 – 364.

Field, A., & Harrison, P. 1988. *Functional Programming*. International Computer Science Series. Addison-Wesley Publishing Company.

Gelernter, D. 1988. Getting the Job Done. *Byte Magazine*, **13**(November), 301 – 308.

Harel, D. 1992. *Algorithmics: The Spirit of Computing*. Addison-Wesley Publishing Company Inc.

Hennesy, M.C.B, & Ashcroft, E.A. 1977. Parameter-Passing Mechanisms and Nondeterminism. *Pages 306–311 of: Proceedings of the ninth ACM Symposium on Theory of Computing*.

Hoogendijk, P., & Backhouse, R. 1994. Relational Programming Laws in the Tree, List, Bag, Set Hierarchy. *Science of Computer Programming*, **22**, 67 – 105.

Hudak, P. 1989. Conception, Evolution, and Application of Functional Programming Lanugaes. *ACM Computing Surveys*, **21**(3), 359 – 411.

Hudak, P., & Sundaresh, R. 1988 (December). *On the Expressiveness of Purely Functional I/O Systems*. Tech. rept. YALEU/DCS/RR665. Yale University, Department of Computer Science.

Hudak, P., *et al.* 1992. Report on the Programming Language Haskell. *ACM Sigplan Notices*, **27**(5).

Hughes, J. 1987 (March). *Backward Analysis of Functional Programs*. Research Report CSC/87/R3. Department of Computing Science, University of Glasgow.

Hughes, J. 1989. Why Functional Programming Matters. *The Computer Journal*, **32**(2), 98 -107.

Hughes, J., & Moran, A. 1995. Making Choices Lazily. *In: Functional Programming and Computer Architecture.* San Diego: ACM Press.

Hughes, J., & O'Donnell, J. 1989. Expressing and reasoning about Non-deterministic Functional Programs. *Pages 308 – 328 of:* Davis, K., & Hughes, J (eds), *Functional Programming : Proceedings of the 1989 Glasgow Workshop.*

Jackson, W., & Burton, F. 1994. A Definite and Unfoldable, Partially Deterministic Language. *The Computer Journal*, **37**(8), 711 – 714.

Jagannathan, S. 1991a. Customization of First-Class Tuple-Spaces in a Higher-Order Language. *In: Parallel Architectures and Languages Europe.* Springer-Verlag LNCS 506.

Jagannathan, S. 1991b. Expressing Fine-Grained Parallelism Using Distributed Data Structures. *In: Workshop on Research Directions in High-Level Parallel Programming Languages.* LNCS, vol. 574. Springer-Verlag.

Kahn, K.M., & M.S.Miller. 1989. Technical Correspondence. *Communications of the ACM*, **32**(10), 1253 – 1255.

Kale, L.V. 1989. Technical Correspondence. *Communications of the ACM*, **32**(10), 1252 – 1253.

Kuchen, H., & Gladitz, K. 1992. Implementing Bags on a Shared Memory MIMD-Machine. *In: Proceedings of the 4th INT. Workshop on the Parallel Implementation of Functional Languages.*

Lock, H., & Jahnichen, S. 1990 (Octoboer). Linda Meets Functional Programming. *In: IEEE 2nd Workshop on Future Trends in Distrubuted Computation.*

MacLennan, B.J. 1990. *Functional Programming: Practice and Theory.* Addison-Wesley Publishing Company, Inc.

Main, M.G. 1987. A Powerdomain Primer. *Bulletin of the EATCS*, **33**, 115 – 147.

Manna, Z. 1974. *Mathematical Theory of Computation.* McGraw-Hill Computer Science Series. McGraw-Hill book Company.

Meertens, L.G.L.T. 1986. Algorithmics - Towards Programming as a Mathematical Activity. *Pages 289 – 334 of: Proceedings CWI Symposium on Mathematics and Computer Science.*

Melton, A., *et al.* 1994 (June). *Denotational Semantics and Domain Theory.* Lecture notes. Department of Computer Science, Michigan Technological University. Presented at WOFACS'94. Hosted by the Departments of Mathematics and Computer Science at the University of Cape Town, South Africa.

Michaelson, G. 1989. *An Introduction to Functional Programming through Lambda Calculus.* International Computer Science Series. Addison-Wesley Publishing Company.

Nielson, H.R., & Nielson, F. 1992. *Semantics with Applications : A Formal Introduction.* Wiley Professional Computing. John Wiley and Sons.

Norvell, T.S., & Hehner, C.R. 1993 (June/July). Logical Specifications for Functional Programs. *In: Mathematics of Program Constructs, Second International Conference.* Lecture Notes in Computer Science, no. 669.

Peyton Jones, S.L. 1987. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science. Prentice-Hall International.

Peyton Jones, S.L. 1989. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, **32**(2), 175 –186.

Reade, C. 1991. *Elements of Functional Programming.* International Computer Science Series. Addison-Wesley Publishing Company.

Rehmet, G.M. 1994 (September). *Remora - Implementing Adaptive Parallelism on a Heterogeneous Cluster of Networked Workstations.* M.Phil. thesis, Computer Science Department, Rhodes University, South Africa.

Schmidt, D.A. 1988. *Denotational Semantics : A Methodology for language development.* Wm. C. Brown Publishers.

Schreiner, W. 1993 (December). *Parallel Functional Programming: An Annotated Bibliography.* Tech. rept. RISC-Linz.

Scientific Computing Associates Inc. 1993. *C-Linda User's Guide and Reference Manual.* Scientific Computing Associates Inc.

Shapiro, E. 1989. Technical Correspondence. *The Communications of the ACM*, **32**(10), 1244 - 1249.

Sondergaard, H., & Sestof, P. 1990. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, **27**, 505–517.

Sondergaard, H., & Sestoft, P. 1992. Non-determinism in Functional Languages. *The Computer Journal*, **35**(5), 514–523.

Wadler, P. 1990. Comprehending Monads. *Pages 61 - 79 of: ACM Conference on Lisp and Functional Programming.*