National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

## NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

## AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

# Functional Programming Applied to Parallel Combinatorial Search

by

W. Ken Jackson

B.Sc., University of Manitoba, 1983

M.Math., University of Waterloo, 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the Department
of
Computing Science

© W. Ken Jackson 1993
SIMON FRASER UNIVERSITY
October 1993

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT A LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE A LA DISPOSITION DES PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

ISBN 0-612-01130-5

Canadä

## APPROVAL

**Name:** W. Ken Jackson

**Degree:** Doctor of Philosophy

**Title of thesis:** Functional Programming Applied to Parallel Combinatorial Search

**Examining Committee:** Dr. Robert Cameron
Chair

Dr. F. Warren Burton
Senior Supervisor

Dr. M. Stella Atkins
Supervisor

Dr. Joseph G. Peters
Supervisor

Dr. Richard Alan Frost
External Examiner

Dr. Fred Popowich
SFU Examiner

**Date Approved:** Oct. 19, 1993

## PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Functional Programming Applied to Parallel Combinatorial S. arch.

Author: ______

(signature)

William Ken Jackson

(name)

Nov 8, 1993

(date)

# Abstract

Functional programs are often more concise, more amenable to formal reasoning, and better suited to parallel execution than imperative programs. This work investigates the application of functional programming to parallel combinatorial search programs such as branch-and-bound or alpha-beta.

We develop an abstract data type called *improving intervals* that can be used to write functional search programs. Programs that use improving intervals are simple because they do not explicitly refer to pruning; all pruning occurs within the data type. The programs are also easily annotated so that different portions of the search space are searched in parallel.

The search programs are verified using *approximate reasoning*: a method of program transformation that uses both equational and approximation properties of functional programs. Approximate reasoning is also used to verify an implementation of improving intervals.

Parallel functional programs have deterministic results. In some cases, permitting some non-determinism in the functional search programs can result in more pruning. We define a restricted form of non-determinism called *partial determinism* that permits a program to return a set of possible results but requires that the set of results be consistent. Partial determinism can improve the performance of the search programs while guaranteeing consistent results. We also show how approximate reasoning can be used to reason about partially deterministic programs.

# Acknowledgments

I would like to thank Dr. Warren Burton for his supervision. He inspired many of the ideas in this thesis, gave quick and useful feedback, as well as providing financial support.

I would also like to thank the other members of my committee: Dr. Joe Peters and Dr. Stella Atkins, for their comments, suggestions, and views from outside the field of programming languages. Thanks also to Dr. Fred Popowich and Dr. Richard Frost for their careful readings of the thesis.

Thanks to sumo for her proof reading, support, and everything else! My daughter Bria has given me two years and four months of fun and interesting times (so far).

Thanks also to Ken McDonald for interesting discussions and motivation to get the thesis written. Daryl Harms showed me that a thesis could be completed and provided lots of meta-advice.

Financial support was gratefully received from Simon Fraser University and the Natural Sciences and Engineering Research Council of Canada.

# Contents

Abstract . . . . . . . . . . iii
Acknowledgments . . . . . . . . . . iv
List of Figures . . . . . . . . . . viii
1 Introduction . . . . . . . . . . 1
1.1 Combinatorial Optimization Problems . . . . . . . . . . 2
1.1.1 The 0/1 Knapsack Problem . . . . . . . . . . 3
1.1.2 Combinatorial Search . . . . . . . . . . 3
1.1.3 An Imperative Branch-and-Bound Program . . . . . . . . . . 5
1.1.4 Search Strategies . . . . . . . . . . 7
1.2 Thesis and Outline of the Dissertation . . . . . . . . . . 7
2 Background . . . . . . . . . . 10
2.1 Functional Programming . . . . . . . . . . 10
2.1.1 Miranda — Syntax . . . . . . . . . . 11
2.1.2 The Computation Model . . . . . . . . . . 13
2.1.3 Strict and Non-Strict Functions . . . . . . . . . . 15
2.1.4 Parallel Functional Programming . . . . . . . . . . 15
2.2 Non-determinism . . . . . . . . . . 19
2.2.1 Non-deterministic Operators in Functional Languages . . . . 20
2.2.2 Domains and Power Domains . . . . . . . . . . 21
2.3 Non-Sequential Functions . . . . . . . . . . 24
2.4 Equational Reasoning . . . . . . . . . . 26
3 Improving Intervals . . . . . . . . . . 28
3.1 Introduction to Improving Values . . . . . . . . . . 29
3.2 Introduction to Improving Intervals . . . . . . . . . . 30

3.3 Specification of Improving Intervals . . . . . . . . . . . . . . . . . . . 31
3.3.1 Approximate Semantics for Improving Intervals . . . . . . . 33
3.3.2 Additional Properties for Improving Intervals . . . . . . . . . 35
3.4 Implementation of Improving Intervals . . . . . . . . . . . . . . . . . . 36
3.4.1 A Depth-first Version of ii_min and ii_max. . . . . . . . . . 42
3.4.2 Correctness of the Implementation . . . . . . . . . . . . . . . 43
3.5 Branch-and-Bound with Improving Intervals . . . . . . . . . . . . . . . 52
3.5.1 Operational Behaviour . . . . . . . . . . . . . . . . . . . . . 54
3.6 Alpha-Beta Using Improving Intervals . . . . . . . . . . . . . . . . . . 60
3.7 Adding Speculative Parallelism . . . . . . . . . . . . . . . . . . . . . . 63
3.7.1 Priorities . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4 Partial Determinism . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.1 Definition of Partial Determinism . . . . . . . . . . . . . . . . . . . . . 68
4.1.1 Partial Determinism and Function Application . . . . . . . . 72
4.2 A Language with Partially Deterministic Functions . . . . . . . . . . . 73
4.2.1 A Deterministic Language . . . . . . . . . . . . . . . . . . . 73
4.2.2 A Partially Deterministic Language . . . . . . . . . . . . . . 75
4.3 Reasoning with Partial Determinism . . . . . . . . . . . . . . . . . . . 78
4.4 Implementation using Partially Mandatory Tasks . . . . . . . . . . . . 80
4.5 Partial Determinism in Branch-and-Bound . . . . . . . . . . . . . . . 83
4.5.1 A Non-Sequential Version of ii_min and ii_max . . . . . . . 83
4.5.2 A Partially Deterministic Version of ii_df_min and ii_df_max 84
4.5.3 A Partially Deterministic Version of ii_min . . . . . . . . . 86
4.6 Partial Determinism for Adapting to Memory . . . . . . . . . . . . . . 87
5 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.1 Parallel Combinatorial Search . . . . . . . . . . . . . . . . . . . . . . . 88
5.1.1 Reducing Search Overhead and Anomalies . . . . . . . . . . 90
5.2 Speculative Parallelism . . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.3 Non-Determinism . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.3.1 Non-deterministic Extensions to Functional Languages . . . 92
5.3.2 Other Approaches . . . . . . . . . . . . . . . . . . . . . . . . 93
6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
6.1 Simpler Programs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

6.2 Parallel Execution . . . . . 96
6.3 Reasoning with Programs . . . . . 97
6.4 Future Research . . . . . 98
6.4.1 Application to Other Areas . . . . . 98
6.4.2 Approximate Reasoning with Non-Deterministic Programs . 98
6.4.3 Other Partially Deterministic Functions and Primitives . . . 99
6.4.4 Scheduling with Speculative Tasks . . . . . 99
6.4.5 Performance . . . . . 99
6.5 Final Remarks . . . . . 100
Appendices
A Transforming the Implementation . . . . . 101
B Performance Testing . . . . . 104
Bibliography . . . . . 111

# List of Figures


1.1 Search Space for a Knapsack Instance . . . . . . . . . . . . . . . . . . . . . . 4

1.2 Branch-and-Bound Pseudo-code . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.1 Domain $(\mathbb{Z} \cup \{\perp\}, \sqsubseteq)$ . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

2.2 Plotkin Power Domain on $\mathbf{Z}_{\perp}$. . . . . . . . . . . . . . . . . . . . . . . . . 24

3.1 The Signature of the Improving Values Abstract Data Type . . . . . . . . . . 29

3.2 The Signature of the Improving Intervals Abstract Data Type . . . . . . . . . 31

3.3 Collected Specification For Improving Intervals . . . . . . . . . . . . . . . . . 34

3.4 Implementation of Values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

3.5 Implementation of Bounds . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

3.6 Implementation of `ii_exact` and `ii_value` . . . . . . . . . . . . . . . . . . . 39

3.7 Implementation of `ii_max` and `ii_min` . . . . . . . . . . . . . . . . . . . . . . 41

3.8 Implementation of `ii_df_max` and `ii_df_min`. . . . . . . . . . . . . . . . . . 42

3.9 A Specification for Branch-and-Bound . . . . . . . . . . . . . . . . . . . . . . 52

3.10 Best-first Branch-and-Bound Using Improving Intervals . . . . . . . . . . . . 52

3.11 Simplified Branch-and-Bound Using Improving Intervals . . . . . . . . . . . . 55

3.12 Example Search Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

3.13 Initial Program Graphs in Evaluation of `bb` . . . . . . . . . . . . . . . . . . . 56

3.14 After Expanding Node $r$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

3.15 After Propagating (5,$\infty$) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

3.16 After Expanding Node $r_1$ and Propagating (12,$\infty$) . . . . . . . . . . . . . . . 58

3.17 After Expanding Node $r_2$ and Propagating (13,$\infty$) . . . . . . . . . . . . . . . 59

3.18 Specification of the Minimax Value of a Game Tree . . . . . . . . . . . . . . . 61

3.19 Alpha-Beta Program using Improving Intervals . . . . . . . . . . . . . . . . . 62

3.20 Example Game Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
3.21 Parallel Branch-and-Bound Program — First Attempt . . . . . . . . . . . . . 64
3.22 Parallel Branch-and-Bound Program — Final Version . . . . . . . . . . . . . 66

4.1 An Expression in the Deterministic Language . . . . . . . . . . . . . . . . . . 74
4.2 Semantics for a Simple Deterministic Language . . . . . . . . . . . . . . . . . 75
4.3 Semantics for a Simple Partially Deterministic Language . . . . . . . . . . . . 76
4.4 Implementation of a Parallel Min . . . . . . . . . . . . . . . . . . . . . . . . 84
4.5 Implementation of `ii_pd_min` and `ii_pd_max`. . . . . . . . . . . . . . . . . . 85

# Chapter 1

# Introduction

Traditional programming languages such as C or Pascal are called *imperative* because a program is expressed as a series of commands that are executed one after another. A functional program has no commands. Instead, the program is expressed only by the definition of functions and the application of functions to arguments.

Advocates of functional programming claim that,

> "programs can be written quicker, are more concise, are higher level (resembling more closely traditional mathematical notation), are amenable to formal reasoning and analysis, and can be executed more easily on parallel architectures." [28, p. 360]

The features of functional programming that lead to the above advantages include:

1. Higher-order functions: functions may take functions as arguments and return functions as results.
2. Lazy evaluation: an expression is not evaluated until its result is needed.
3. Equational syntax: a functional program looks like a set of mathematical equations.
4. Polymorphic types: a strong but flexible type system based on parameterized types.
5. Data abstraction: facilities for defining abstract data types and modules.
6. Absence of side effects: evaluating an expression gives its value and does nothing else.

This dissertation explores the application of functional programming to search programs. Search programs are often used to solve combinatorial optimization problems such as the 0/1 knapsack problem, the travelling salesman problem, integer programming, and game playing. Many of these problems are difficult to solve: they are often NP-hard. Executing search programs on parallel machines provides an opportunity to reduce the time required to solve such problems.

Our approach to writing functional search programs is based on a new abstract data type, called *improving intervals*, that encapsulates the pruning behaviour that occurs in search programs. The definition of this data type relies heavily on lazy evaluation and the encapsulation of pruning leads to concise and simple programs.

Functional programs are naturally parallel because each argument of a function can be evaluated in parallel; they do not have the inherent "one after another" nature of imperative programs. Parallel imperative search algorithms typically use shared variables and involve tasks that asynchronously update the shared variables. It is not possible to express such algorithms using functional languages. However, functional search programs that use improving intervals can be executed on parallel machines using *speculative parallelism*. Speculative parallelism creates a task to evaluate an expression before its result is known to be needed.

In some cases, the performance of the search programs can be improved by permitting some non-deterministic behaviour. However, permitting non-deterministic behaviour in functional programs is difficult: functions are by definition deterministic and introducing non-deterministic constructs hampers the ability to formally reason about programs. We introduce a new concept called *partial determinism* and show how it captures the type of non-determinism required by parallel search programs while preserving the ability to formally reason about the programs.

## 1.1 Combinatorial Optimization Problems

Combinatorial optimization problems arise in various fields such as operations research, scheduling, CAD, AI, and game playing. An instance of a combinatorial optimization problem consists of a non-empty discrete set $X$ and a function $f(x)$ defined on $X$. We are asked to find an element $x^*$ of $X$ that optimizes the function $f(x)$. The function $f(x)$ is called the ***objective function*** and an optimization problem is either a minimization problem or a

maximization problem depending on whether we want to minimize or maximize $f(x)$.

### 1.1.1 The 0/1 Knapsack Problem

The 0/1 Knapsack problem is a simple maximization problem. We are given a set of items and each item has an associated profit. The goal is to fill a knapsack such that the total profit is maximized and the capacity of the knapsack is not exceeded. More precisely, the problem is defined as follows:

**Given:** A finite set of objects $U = \{u_1, u_2, \ldots, u_n\}$, a weight function $w: U \rightarrow \mathbf{Z}^+$, a profit function $p: U \rightarrow \mathbf{Z}^+$, and a capacity $C \in \mathbf{Z}^+$.

**Find:** A subset $U'$ of $U$ that maximizes $\sum_{u \in U'} p(u)$ such that $\sum_{u \in U'} w(u) \leq C$

In this problem, the set $X$ is all the subsets of $U$ whose total weight is less than or equal to the knapsack's capacity and the objective function is $f(U') = \sum_{u \in U'} p(u)$.

### 1.1.2 Combinatorial Search

In many cases, a problem instance can be divided into simpler problems. A problem instance $(X', f)$ is a sub-instance of $(X, f)$ iff $X'$ is a non-empty subset of $X$. If $f$ is clear from context, we often say $X'$ is a sub-instance of $X$. For any sub-instance $X'$ of $X$, let $f^*(X')$ be the cost of an optimal solution of $X'$. The function $f^*(X')$ may be defined as

$$f^*(X') = min\{f(x) \mid x \in X'\}.$$

Top-down search programs use a *branching* function $b(X)$ that divides an instance into a set of sub-instances. The iterative application of $b(X)$ starting with the original instance yields a tree or graph of sub-instances called the search space[1]. A leaf occurs in the tree where there is a sub-instance $X'$ with $|X'| = 1$.

For example, an instance of the 0/1 knapsack problem can be divided into two sub-instances: one that includes the first item (subject to there being sufficient capacity) and another that excludes the first item. Figure 1.1 shows part of the search space for an instance of the 0/1 knapsack problem using the above branching function.

[1] We consider only search spaces structured as trees.

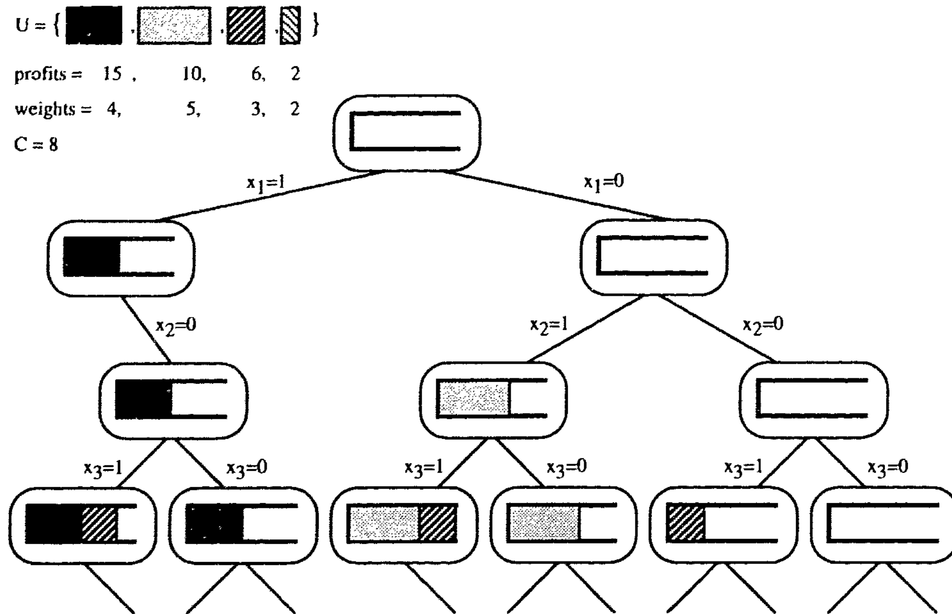


Figure 1.1: Search Space for a Knapsack Instance

Given a problem instance $(X, f)$ and a branching function $b(X)$, the cost of an optimal solution can be defined as:

$$opt(X, f, b) = \begin{cases} f(x), & \text{if } X = \{x\} \\ min\,\{opt(X', f, b)\,|\,X' \in b(X)\}, & \text{if } |X| > 1 \end{cases} \tag{1.1}$$

For many problems, the size of $X$ is large enough that it is not practical to exhaustively generate all of the search space. Branch-and-bound methods [26, 39] use bound functions to avoid generating the entire search space. Branch-and-bound subsumes many top-down search techniques [35] including the $A^*$ algorithm and the alpha-beta algorithm for searching game trees [44].

A lower bound $l(X')$ and an upper bound $u(X')$ are functions on sub-instances such that, for any sub-instance $X'$,

$$l(X') \leq f^*(X') \leq u(X')$$

For minimization problems, the utility of bound functions relies on the observation that if $X''$ and $X'$ are two sub-instances such that $u(X') < l(X'')$ then no element in $X''$ can be the optimal solution and the search space rooted at the node representing $X''$ does not need to be generated. In such cases, we say that the node representing $X''$ can be pruned.

Maximization problems can be treated as minimization problems by negating the objective function and negating and swapping the bound functions.

For many problems, bound functions can be found by relaxing some of the constraints. In the 0/1 knapsack problem, for example, a simple upper bound relaxes the 0/1 constraint. The items are sorted by their profit-to-weight ratios and the knapsack is packed with as many items as will fit plus some *fraction* of the next item.

### 1.1.3 An Imperative Branch-and-Bound Program

The pseudo-code in figure 1.2 illustrates a typical imperative branch-and-bound program for a minimization problem[2]. The function `branch_and_bound(r)` returns a node that represents a sub-instance of `r` containing an optimal solution (if `branch_and_bound(r)` returns `r*` then `r*` represents the sub-instance $\{x^*\}$). An *iteration* of branch-and-bound refers to the execution of the body of the `while` loop. Each iteration selects a node, updates the best current solution (the incumbent), and possibly expands the selected node. The main data structure is a priority queue of nodes that contains the nodes the have been created but not yet expanded; such nodes are called *open* nodes. The function initially inserts the root node `r` in the priority queue and terminates when the priority queue becomes empty.

The function `children(r)` is the branching function and it returns a list of nodes representing sub-instances of `r`. The functions `lb(r)` and `ub(r)` are the bound functions. The function `has_direct_solution(r)` is true when the node `r` represents a sub-instance that can be solved directly. In that case, `cost(r)` is the value of the objective function at the optimal solution. A node that can be solved directly is called a *solution* node. We assume that for any solution node `r` that `lb(r)` = `cost(r)` = `ub(r)`.

The *incumbent* is the best current solution node, that is, the node with the minimal cost over the set of solution nodes that have currently been expanded. The *incumbent's value* is cost of the incumbent's solution.

The program uses a variable `min_ub` to record the minimum upper bound of the expanded nodes. Any node whose lower bound is greater than `min_ub` is pruned.

There are several improvements that can be made to the above code. For example, we could check that `lb(k)<min_ub` for each child `k` before putting `k` in the priority queue. There are other enhancements such as dominance tests and equivalence tests that we have

[2]The pseudo-code is copied from [56] with some minor changes.

```
function branch_and_bound(r: Node) : Node
     pq : priority_queue of Node
     incumb : Node                          /* the incumbent */
     min_ub : int                           /* current min upper bound */

     if has_direct_solution(r) then return r
     min_ub = ub(r)
     insert(pq, r)
     while  not is_empty(pq)
        r := delete_max(pq)
        if lb(r) < min_ub
            min_ub := min(min_ub,ub(r))
            if has_direct_solution(r)
                incumb := r
            else
                /* Expand the node */
                for each k in children(r)
                    insert(pq, k)
                end
            fi
        else
            /* node r is pruned */
        fi
     end
     return(incumb)
end
```

Figure 1.2: Branch-and-Bound Pseudo-code

omitted for the sake of simplicity.

The time taken by a branch-and-bound program is often measured by the number of nodes generated. The space used by the program is measured by the maximum size of the priority queue.

### 1.1.4 Search Strategies

The branch-and-bound program above uses a priority queue to record the open nodes. The order in which nodes are expanded depends on an assignment of priorities to nodes. Different search strategies result in different time and space requirements and the two most common strategies are depth-first and best-first.

The depth-first strategy corresponds to prioritizing nodes in a manner such that each node's priority is greater than the priority of any node to its right.

The best-first strategy gives higher priority to nodes with the smaller lower bounds and ties are usually broken in favour of the node that was generated first.

The best-first strategy corresponds to the $A^*$ algorithm [44] and is known to generate the fewest number of nodes in the worst case. The space used with the best-first strategy typically grows exponentially with the depth of the search tree. This affects the amount of time taken by each iteration of the `while` loop because the time to delete/insert items from the priority queue grows as the priority queue becomes larger.

The depth-first strategy can generate more nodes but the space grows linearly with the depth of the search tree. In addition, a stack can be used to implement the priority queue so the delete and insert operations can be done in constant time. The depth-first strategy can fail to terminate if the search tree contains an infinite branch.

## 1.2 Thesis and Outline of the Dissertation

It is possible to code the standard imperative search algorithms in a functional manner [6, 46] but this does not result in simpler programs and does not lead to any new insights. Our thesis is that functional programming can be applied to combinatorial search programs so that the programs are simpler, more amenable to formal reasoning, and easily executed in parallel. The dissertation demonstrates that this can be done by:

1. Defining a new abstract data type that encapsulates pruning.

2. Using speculative parallelism to execute the programs in parallel.
3. Using partially deterministic functions to permit more non-deterministic behaviour while still guaranteeing consistent results.
4. Using approximations to formally reason about the programs.

Chapter 2 provides the background necessary to understand the rest of the dissertation and describes lazy evaluation, speculative parallelism, non-determinism, and approximate reasoning, in more detail.

Chapter 3 develops the *improving intervals* abstract data type that forms the basis for simplifying search programs. This data type is based on the observation that during execution of a branch-and-bound program, the bounds on the optimal solution value form an interval that becomes tighter (or improves) as the search tree is explored. The implementation of improving intervals relies on lazy evaluation. We give a branch-and-bound program and an alpha-beta program that use improving intervals. These programs are simple because all pruning is encapsulated within the data type.

Functional programs are amenable to equational reasoning where the algebraic properties of programs are used to reason about the correctness of the programs. We extend the idea of equational reasoning to include the use of *approximations*. Approximations are inequalities defined using the approximates relation ($\sqsubseteq$) from domain theory. We verify our search programs and also verify an implementation of improving intervals using approximate reasoning. We use approximations, rather than equations, because they result in simple proofs for the search programs and they are also appropriate for reasoning about the partially deterministic programs that are discussed later.

Section 3.7 describes how search programs can be executed in parallel using speculative parallelism. With speculative parallelism a task is initiated before knowing that the results of the task will be required. The advantage of speculative parallelism in functional languages is that we obtain parallel behaviour without affecting the results of the programs; the parallel program remains deterministic.

However, in some cases, permitting some non-determinism can result in more pruning. Consider searching distinct sub-spaces in parallel with asynchronous processes. Finding a solution in one sub-space may be sufficient to prune the other sub-space. A program that abstracts the actual order of finding solutions is non-deterministic. However, in section 2.4, we show that adding non-determinism to functional programs hampers the ability

to formally reason about the programs. Chapter 4 introduces an alternative called *partial determinism* that allows the type of non-determinism required by parallel search programs while retaining the ability to formally reason with the programs. A partially deterministic program is non-deterministic but it is restricted to give consistent results in terms of information content. The idea of "information content" arises from domain theory. Chapter 4 provides a precise definition of partial determinism using domain theory, considers the effect of extending functional languages with partial determinism, and describes how partially deterministic programs are amenable to approximate reasoning.

Chapter 5 relates our work to other work on search, non-determinism, and speculative parallelism. Finally, chapter 6 describes some conclusions and areas for future research.

# Chapter 2

# Background

This chapter begins with a review of functional programming, including a brief introduction to the syntax of the functional language Miranda[1]. We then give an intuitive description of graph reduction and lazy evaluation and show how lazy evaluation permits the definition of non-strict functions. We also cover the distinction between speculative and mandatory parallelism and define the annotations `par`, `spec`, and `priority` for parallel functional programming.

Section 2.2 clarifies the concept of non-determinism and defines the non-deterministic operators `amb`, `choose`, and `nd_merge` that have been used as extensions to functional languages. The semantics of programming languages with non-determinism involves domains and power domains. We briefly review their definitions.

Non-sequential functions are described in section 2.3. These functions are interesting because they require fair evaluation of their arguments and hence cannot be expressed in functional languages. The chapter ends with a description of equational reasoning with functional programs.

## 2.1 Functional Programming

This section gives a brief introduction to functional programming. Its aim is not to be comprehensive but instead it defines the terminology used later and places this work in the context of previous research on functional programming.

[1] Miranda is a trademark of Research Software Ltd.

### 2.1.1 Miranda — Syntax

Miranda is used as the programming notation throughout the dissertation. Expressions and function definitions appear in a typewriter font. Turner [62] gives a good overview of Miranda. The following briefly describes some aspects of the syntax that are used later.

A functional program is a set of type and function definitions. Miranda uses an equational syntax so a function definition looks like an equation. For example, the following is a definition for a function `fib` that computes the $n^{th}$ Fibonacci number.

```
> fib n = 1, if n=0 \/ n=1
>       = fib (n-1) + fib (n-2),    if n>1
>       = error "fib:of -'ve num", otherwise
```

The application of a function `f` to an argument `a` is denoted by the juxtaposition `f a` instead of the more usual `f(a)`. The right-hand side of a definition is a sequence of guarded expressions (a guard is a boolean expression) and provides one method for case analysis.

Pattern matching is an alternative way of doing case analysis in function definitions that is often more concise than using guards. Patterns occur in the argument positions on the left-hand side of a function definition. The following definition of `fib` uses the patterns `0`, `1`, and `n+2`.

```
> fib 0 = 1
> fib 1 = 1
> fib (n+2) = fib (n+1) + fib n
```

In general, a pattern `n+k` matches a numeric argument that is greater than or equal to `k` and has the effect of binding the variable `n` to the argument minus `k`. The pattern `[]` matches an empty list while the pattern `(x:xs)` matches a non-empty list and binds `x` to its head and `xs` to its tail. The pattern `(x1,x2,...xn)` matches an n-tuple and binds `x1` through `xn` to the components of the tuple.

Local definitions are possible using a `where` clause. For example, the following uses pattern matching and a local definition to define a function `fib2` such that

$$\texttt{fib2 n} = \texttt{(fib (n-1), fib n)}.$$

```
> fib2 0 = (error "fib:of -'ve num",1)
> fib2 1 = (1,1)
```

```
> fib2 (n+1) = (f2,f1+f2)
>              where
>                  (f1,f2) = fib2 n
```

The scope of the where clause is the entire right-hand side of the equation. The indentation is significant and is used by the compiler to determine the block structure of the program.

### The Type System

Miranda is strongly typed and each variable has a type that can be inferred from the program text by the compiler. The programmer is not required to specify types though it is often useful to do so for documentation. The primitive types in Miranda are booleans (`bool`), characters (`char`), and numbers (`num`). The type `num` includes both floating point numbers and integers.

The list type is typically the most important type in functional programs. `[T]` represents the type of lists whose elements are of type `T`. Thus `[num]` is the type of lists of numbers.

Other type constructors are `(T1, T2,...Tn)` for the type of n-tuples and `T1 -> T2` for the type of functions with argument type `T1` and result type `T2`.

User defined types can be defined using type synonyms, algebraic types, or abstract data types. A type synonym such as

```
> word == [char]
```

defines an alternate name for a type. Algebraic types can be used for tagged unions and for recursive types. For example, binary trees with integer leaves are defined by the following algebraic type:

```
> bin_tree ::= Leaf num | Node bin_tree bin_tree
```

`Leaf` and `Node` are called *constructors* and may be used as functions (`Leaf 1` is a valid expression) or in patterns (`Leaf x` matches a leaf node and binds `x` to its integer label).

Miranda uses a polymorphic type system. The type variables `*`, `**`, `***`, etc. can be used to represent an arbitrary type, for example `[*]` is the type of a list of elements of an arbitrary type.

The type of a variable can be explicitly stated when desired. For example, the type of the `map` function (that applies a function to every element of a list) can be stated with the following:

```
> map :: (*->**) -> [*] -> [**]
```

In Miranda, functions are *curried* so that functions of several arguments are actually functions that take a single argument and return a function. The following addition function

```
> add x y = x + y
```

appears to take two arguments but its actual type is

```
> add :: num -> (num -> num)
```

so it maps a number to a function from numbers to numbers. The brackets in the above expression are redundant because the type constructor `->` associates to the right. With currying, functions can easily be partially applied. For example, `add 1` is a valid expression and it denotes a function that adds one to its argument.

### 2.1.2 The Computation Model

A computational model for functional programs is typically based on a reduction system: an expression is executed with respect to a functional program by reducing the expression to normal form. Most functional languages can be considered as sugared syntax for the lambda calculus and so the formal definition of reduction in the lambda calculus carries over to functional programs.

Each reduction step replaces a sub-expression, called the *redex*, by an equivalent sub-expression. Redex is short for reducible expression. An expression is in *normal form* if it has no redexes. Within an expression, there may be many sub-expressions that are redexes; a reduction strategy is a method for choosing the redex to be replaced. The *normal-order* reduction strategy chooses the left-most outer-most redex. Normal-order reduction corresponds to executing the body of a function as far as possible before evaluating the function's arguments. In contrast, imperative languages typically use the *applicative-order* strategy that evaluates the arguments before executing the function's body.

Two fundamental results that carry over from the lambda calculus are:

1. If an expression $x$ reduces to the normal form $y$ and to the normal form $z$ then $y = z$.
2. If an expression $x$ can be reduced to the normal form $y$ then $x$ can be reduced to $y$ using normal-order reduction.

The first result ensures that the normal form is unique regardless of the reduction strategy. Hence, sub-expressions may be reduced in parallel without affecting the result of the program. The situation is not quite this simple because some care is required in handling sub-expressions whose evaluation may not terminate. The second result shows that the normal-order strategy finds the normal form if it exists.

Expressions can be represented as either strings or graphs. The graph representation permits sharing of sub-expressions when a variable occurs more than once in an expression. For example, in the following function definition,

```
> f x = z * (x / z)
>        where z = g x
```

the variable `z` occurs twice and is bound to the shared expression `g x`. Graph reduction reduces a shared sub-expression only once while string reduction reduces a shared sub-expression each time it occurs. The time used by the program is measured by the number of reduction steps performed to reduce an expression to normal form. The space used is the size (the number of nodes) of the largest graph that exists during reduction.

Lazy evaluation uses the normal-order reduction strategy with graph reduction. With lazy evaluation, a sub-expression is only evaluated when its value is needed and shared sub-expressions are evaluated only once. For example, with the following definition,

```
> f x y = (x-1)/x, if x ~= 0
>       = y,       otherwise
```

the expression `f (1+1) (1/0)` reduces to `0.5`. The argument `(1+1)` is reduced once even though `x` occurs three times in the body of `f`. In addition, the function returns a result even though its second argument is undefined.

In practise, a redex is selected and is itself reduced. However, the redex is not reduced all the way to normal form. An expression is in *weak head normal form* (WHNF) iff it is not a redex and cannot become a redex by reducing its sub-expressions [48]. We refer to the process of reducing an expression to WHNF as *evaluating* the expression. Later in the dissertation, we use the fact that a list of the form `x:xs` is in WHNF. The evaluation of an expression that denotes a list stops once the expression is reduced to a list cell; neither the head nor the tail is evaluated.

### 2.1.3 Strict and Non-Strict Functions

Lazy evaluation allows the definition of functions that may not need the value of an argument. Such functions are called non-strict and a simple example is the constant-one function, defined by the following:

```
> const_one x = 1
```

More precisely, a function $f(x)$, is *strict* iff $f(\perp) = \perp$, where $\perp$ denotes a non-terminating computation. Otherwise the function is called *non-strict*. This idea can be generalized to functions with more than one argument (but recall that, with currying, all functions take a single argument).

All general-purpose programming languages provide some non-strict constructs: the `if` statement is non-strict because a statement like

```
if True then c1 else ⊥
```

evaluates `c1` without evaluating $\perp$. Most languages also provide a non-strict conditional or function, we call it `cor`, that is non-strict in its second argument because

```
cor True ⊥ = True.
```

However, in most languages, user-defined functions are always strict.

In Miranda, constructor functions are used to build data structures. Constructor functions are an important class of non-strict functions. In particular, the list constructor is non-strict in both arguments so that $(\perp : \perp) \neq \perp$. Lists constructed with this non-strict constructor are called *lazy lists* (or "streams") and are used extensively in the dissertation.

### 2.1.4 Parallel Functional Programming

Two main approaches to parallel programming are: to leave the parallelism implicit and let the compiler determine what to do [21]; or to annotate the program to explicitly indicate what should be done in parallel. Functional programming provides advantages for both approaches though we consider only the latter.

We use annotations similar to those proposed by Burton [12] and Hudak [27] to explicitly indicate the parallelism in functional programs. Annotations are special functions that do not affect the meaning of the program but do affect its execution. The advantage of

such annotations is that the annotated/parallel version is correct provided that the un-annotated/sequential program is correct. Therefore, a parallel program can be developed by first developing and debugging a sequential program and then adding the appropriate annotations.

The annotation par is used to initiate the parallel evaluation of the argument to a function (the original task goes on to evaluate the function — just as with normal order reduction). For example, evaluating the expression par (add e1) e2 creates a task to evaluate e2 in parallel with the evaluation of (add e1). Semantically, par behaves as if it were defined by,

```
> par f x = f x, if x ≠ ⊥
>         = ⊥,   otherwise
```

Notice that evaluating par f x does not terminate if evaluating x does not terminate. Therefore, the function par is an annotation (it preserves meaning) only for strict functions: if f is a non-strict function then par f is not equal to f. The next section describes an annotation called spec that can be used to introduce parallelism with non-strict functions.

The par annotation can be used to build other functions for parallel programming. For example, a parallel map function that applies a function f to each element of a list in parallel can be expressed as follows:

```
> par_map f [] = []
> par_map f (x:xs) = par ((f x):) (par_map f xs)
```

The notation ((f x):) is an example of Miranda's notation for partially applying an infix function. The expression (x:) denotes a function such that (x:) xs = x:xs. Hence, ((f x):) denotes a function that appends the element (f x) to a list.

### Speculative and Mandatory Parallelism

Burton [10] and Peyton-Jones [47] make a distinction between two types of parallelism: mandatory parallelism and speculative parallelism.

Mandatory parallelism refers to evaluating an expression whose result is known to be required. With mandatory parallelism, no processor wastes its time evaluating an expression whose result may not be required. If the evaluation of the expression par f x is mandatory then the par annotation initiates a mandatory task to evaluate x.

Speculative parallelism is used to evaluate an expression whose result may not be needed. If the result is needed then time is saved by the parallel evaluation of the expression. If the result is not needed then a processor has wasted some time by evaluating the expression.

Speculative parallelism is useful with non-strict functions. For example, consider evaluating an expression such as `cor e1 e2` where `cor` is the conditional-or function. The function `cor` is strict in its first argument but non-strict in its second argument so mandatory parallelism could be used to evaluate `e1` and speculative parallelism may be used to evaluate `e2`.

The annotation `spec` is used to introduce speculative parallelism. Semantically, `spec` behaves as if it were defined by

```
> spec f x = f x
```

and so `spec` is the identity function on both strict and non-strict functions. Evaluating the expression `spec f x` creates a new speculative task to evaluate `x` while the original task proceeds with the evaluation of `f x`. If the evaluation of `f x` does not need the value of `x` then the speculative task becomes irrelevant and may be killed. Otherwise when the evaluation of `f x` requires the value of `x` then the task evaluating `x` becomes mandatory and the original task blocks until `x` is evaluated.

Tasks are either speculative or mandatory. Mandatory tasks are scheduled ahead of speculative tasks. There is always at least one mandatory task because the initial task is mandatory and a mandatory task that evaluates `spec f x` or `par f x` goes on to evaluate `f x`.

Fair (or pre-emptive) scheduling is not required. A mandatory task can always be run to completion because if the task does not terminate then the program must not terminate. A speculative task can also be run to completion. If a speculative task is scheduled then there must be a mandatory task scheduled on some other processor. If the speculative task does not terminate then the processor that is running the mandatory task may be used to complete the program.

A speculative task may become mandatory or may become irrelevant. Any task that becomes irrelevant should be killed and reclaimed through a process similar to garbage collection. A speculative task cannot initiate a mandatory task.

It is often useful to be able to place priorities on speculative tasks so that the system can schedule tasks that the programmer deems are more likely to be needed. The annotation

`priority p x` modifies the priority of a speculative task evaluating `x` to be `p`. For example, evaluating the expression

```
spec (cor e1) (priority 100 e2)
```

initiates a speculative tasks to evaluate `e2` whose priority will be set to `100`. The semantics of the `priority` annotation are given by the following definition.

```
> priority p x = x, if p ≠ ⊥
>              = ⊥, otherwise
```

We assume that priorities are numeric and that a larger number represents a higher priority. A speculative task created without a priority annotation has the same priority as its parent task if the parent is speculative or otherwise has some maximal speculative priority. Evaluating a priority annotation has no effect on a mandatory task.

Priorities are just hints to the scheduler and may be ignored. For example, we want to permit a distributed implementation where each processor has a local priority queue of tasks rather than requiring a global priority queue be shared among all the processors.

Speculative parallelism is useful with lazy data structures such as lists. The annotation `spec_list`, defined below, is the identity function on lists, but uses speculative parallelism to evaluate the spine of the list.

```
> spec_list [] = []
> spec_list (x:xs) = spec (x:) (spec_list xs)
```

Only the spine of the list, as opposed to its elements, is evaluated. For example, a task evaluating the expression `spec_list [e1,e2,e3]` terminates after reducing this expression to `e1:spec_list [e2,e3]` without evaluating `e1` and also initiates another speculative task to evaluate `spec_list [e2,e3]`.

It is easy to write a function that speculatively evaluates each element of the list by using two spec annotations:

```
> spec_list_elems [] = []
> spec_list_elems (x:xs) = spec (spec (:) x) (spec_list_elems xs)
```

The sub-expression `spec (:) x` denotes a function that takes a list and returns the list with `x` appended as the head and also initiates a speculative task to evaluate `x`.

## 2.2 Non-determinism

Non-determinism naturally arises in parallel programs where several tasks have access to a shared resource. Most programs do not precisely specify the order in which tasks access a shared resource and different results may be obtained depending on the actual order of execution.

In imperative programs, non-determinism is often implicit in the constructs of the languages. For example, in Dijkstra's guarded command language the `if` statement non-deterministically selects and executes a branch with a true guard. Languages that support message passing usually contain a `receive` statement that retrieves messages in a first-in first-out order. If two client tasks send messages to the same server, then the order in which the messages are retrieved is non-deterministic.

A major concern in writing parallel programs is the synchronization of the tasks so that non-determinism is at least somewhat controlled. Bugs may occur because of improper synchronization. Such bugs can be difficult to detect and correct because a small change in the actual execution order of the tasks can hide or reveal the bug.

It is often helpful to view execution of a program as a sequence of states in a state transition system. A deterministic program gives a linear sequence of states whereas a non-deterministic program gives a branching tree of states. Each branching point involves a choice. The choice depends on the program as well as other factors, like the actual order of execution, that are not specified in the program. There must be some method for resolving the choice and different strategies for resolving the choice lead to different types of non-determinism:

**Global or Local** With global non-determinism, the choices are made to ensure that the program as a whole terminates successfully. For example, in a non-deterministic finite automaton a string is accepted if there is *some* sequence of valid transitions from the start state to a final state. At each transition, the machine chooses to move to a new state but this choice must be made so that eventually a final state is reached.

With local non-determinism, the choices can be made locally without considering global aspects of the program. Most parallel programs use local non-determinism. For example, when two tasks access a shared resource the choice may be based on the time when a task first accessed the resource.

**Angelic, Demonic, or Erratic** There are several possibilities for the interaction of non-terminating computations and non-determinism. Angelic non-determinism always makes a choice to avoid non-termination; demonic non-determinism always chooses non-termination; while erratic non-determinism make a random choice.

**Weak or Strong** Weak non-determinism refers to a program whose execution may be non-deterministic but whose result is deterministic. Dijkstra [17] uses his guarded command language to write many non-deterministic programs that always produce the same result.

Strong non-determinism refers to programs that produce different results. Operating systems and many real-time systems are strongly non-deterministic.

Our concept of partial determinism lies between weak and strong non-determinism. A partially deterministic program is strongly non-deterministic because different results can be produced but the results are all consistent.

The various types of non-determinism can be combined in different ways. For example, in Dijkstra's guarded command languages the guarded `if` statement is a local angelic non-deterministic construct but many programs written using the guarded command language are weakly non-deterministic. The non-determinism in logic programming languages involves making a global angelic choice between clauses in a predicate definition. Logic programs are often strongly non-deterministic because they return results corresponding to different proofs of the goal.

### 2.2.1 Non-deterministic Operators in Functional Languages

In functional languages, non-determinism is often expressed by the use of pseudo-functions. They are called pseudo-functions because they do not map the same arguments to the same result. McCarthy's **amb** operator is a local angelic pseudo-function that behaves as if it were defined by the following:

```
> amb ⊥ ⊥ = ⊥
> amb a ⊥ = a
> amb ⊥ a = a
> amb a b = a or b
```

The intended behaviour of amb is that it evaluates both arguments in parallel and returns the first one to finish.

The pseudo-function choose is pseudo-function that is local erratic. It behaves as if it were defined by

```
> choose a b = a or b
```

The amb operator must return a non-bottom argument if one exists while choose may return $\perp$ if either of its arguments is $\perp$.

The operator amb can be used to write some other common pseudo-functions. The non-deterministic merge operator nd_merge, as defined below, returns a fair interleaving of two lists.

```
> nd_merge xs ys
>     = ys, if choice=1 & xs=[]
>     = xs, if choice=2 & ys=[]
>     = (hd x):nd_merge (tl xs) ys, if choice = 1 & xs~=[]
>     = (hd y):nd_merge xs (tl ys), if choice = 2 & ys~=[]
>       where
>        choice = amb (seq xs 1) (seq ys 2)
```

The amb operator is used to non-deterministically choose between 1 and 2 based on which of xs or ys evaluates first (the expression seq xs 1 evaluates to 1 after evaluating xs to WHNF: that is, the empty list or a list cell). The use of amb ensures that nd_merge is bottom-avoiding. The operational reading of nd_merge is that it evaluates both of its arguments in parallel and merges elements from the arguments in the order that they become available. The behaviour of nd_merge corresponds to the non-determinism that occurs with message passing when two tasks each send a stream of messages to the same receiver. The tasks evaluating each argument of nd_merge must be evaluated with a fair scheduler so that if one argument fails to terminate then elements from the other argument can still be used.

### 2.2.2 Domains and Power Domains

In chapter 4, denotational semantics is used as a tool for understanding some aspects of partial determinism. Denotational semantics is built on the concept of a domain. Power

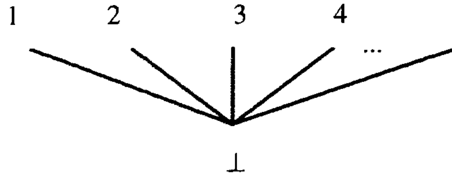


Figure 2.1: Domain $(\mathbb{Z} \cup \{\perp\}, \sqsubseteq)$

domains are used for handling non-determinism. The following reviews some of the basic definitions for domains and power domains.

A domain is a set with a partial order $\sqsubseteq$ that obeys the restriction described below. The usual reading for $\sqsubseteq$ is "approximates" where $a$ approximates $b$ iff $b$ has as much information content as $a$. Every domain contains an element $\perp$ that is considered to contain no information or to denote an undefined value. Figure 2.1 gives a pictorial representation of part of the domain $(\mathbb{Z} \cup \{\perp\}, \sqsubseteq)$ where for all $x, y \in \mathbb{Z}$, $x \sqsubseteq y$ iff $x = \perp$ or $x = y$. In more operational terms, $\perp$ stands for a value that has not been computed yet. We often use $\perp$ to stand for a non-terminating computation because a non-terminating computation is always "not computed yet". Domains like that in figure 2.1 are called flat.

The notion of approximates and information content is better illustrated with more structured domains. For example, with lists

$$1{:}\perp \sqsubseteq 1{:}2{:}\perp \sqsubseteq [1,2,3]$$

In the expression `1:⊥`, nothing is known about the tail of the list. The tail of the list `1:2:⊥` is known to start with a 2 and the tail of the list `[1,2,3]` is known to be the list `[2,3]`. The computable functions from $D_1$ to $D_2$ form a domain that is partially ordered by the following: if $f$ and $g$ are functions in $D_1 \rightarrow D_2$ then $f \sqsubseteq g$ iff for all $x$ in $D_1$, $f(x) \sqsubseteq_{D_2} g(x)$.

In a deterministic language, every expression denotes a single value. However, when non-determinism is added, an expression denotes a set of possible values. For example, the expression `choose 1 2` denotes the set $\{1, 2\}$. The sets of possible values are elements in a power domain (the analogue of a power set). The elements of a power domain are sets and the ordering on the sets combines the subset relation with the approximates relation.

We now define some of the above terms with more precision.

A *poset* $(D, \sqsubseteq)$ is a set $D$ with a binary relation $\sqsubseteq$ on $D$ that is reflexive, transitive, and anti-symmetric. A subset $S$ of $D$ is a *chain* iff for every pair of elements $a, b \in S$, either $a \sqsubseteq b$ or $b \sqsubseteq a$. Given a subset $S$ of $D$, an element $b \in D$ is an *upper bound* on $S$ iff for all

$a \in S$, $a \sqsubseteq b$. An upper bound $b$ on $S$ is the *least upper bound* iff, for every upper bound $b'$ on $S$, $b \sqsubseteq b'$. $\bigsqcup S$ denotes the least upper bound on $S$ when it exists. Similarly, $\bigsqcap S$ denotes the *greatest lower bound* on $S$ when it exists.

A poset $(D, \sqsubseteq)$ is a *domain* iff every chain of $D$ has a least upper bound in $D$. Every domain has a least element that is denoted by $\bot_D$ or just $\bot$ if $D$ is clear from context. Given a set $A$ not containing $\bot$, the domain $A_\bot = (A \cup \{\bot\}, \sqsubseteq)$, where for $a, b \in A$, $a \sqsubseteq b$ iff $a = b$ or $a = \bot$, is called a *flat domain.*

A function $f: D_1 \to D_2$ is *monotonic* iff $x \sqsubseteq_{D_1} y$ implies $f(x) \sqsubseteq_{D_2} f(y)$ for all $x, y \in D_1$ and $f$ is *continuous* iff $\bigsqcup_{D_2} \{f(x) \mid x \in X\} = f(\bigsqcup_{D_1} X)$ for all the chains $X$ of $D_1$. Every computable function is monotonic and continuous.

There are three standard power domain constructions [49, 53]: the Plotkin power domain for erratic non-determinism, the Hoare power domain for angelic non-determinism, and the Smythe power domain for demonic non-determinism. We use the Plotkin power domain to model the erratic non-determinism that occurs in parallel search programs.

When $D$ is a flat-domain, the elements of the Plotkin power domain $\mathcal{P}(D)$ are the non-empty subsets of $D$ that are either finite or contain $\bot$. Since $\bot$ can be an element in a set, non-termination is a possible result. The ordering $\sqsubseteq_{\mathcal{P}}$ is defined for all $A, B \in \mathcal{P}(D)$ as,

$$A \sqsubseteq_{\mathcal{P}} B \text{ iff } (\forall a \in A.\exists b \in B.a \sqsubseteq b) \wedge (\forall b \in B.\exists a \in A.a \sqsubseteq b).$$

The least element of $\mathcal{P}(D)$ is $\{\bot\}$ (the empty set is not an element of $\mathcal{P}(D)$ because every program at least returns $\bot$ as a result). Figure 2.2 shows part of the Plotkin power domain on $\mathbf{Z}_\bot$.

When $D$ is not a flat domain then the above power domain construction is more complicated. For completeness, we give the Plotkin power domain construction for non-flat domains following Broy [8].

The first problem is that $\sqsubseteq_{\mathcal{P}}$ is not anti-symmetric. The standard solution is to divide the subsets of $D$ into equivalence classes. For the Plotkin power domain, the equivalence relation is based on the *convex closure* of a set. For a subset $S$ of $D$, the convex closure $conv(S)$ of $S$ is defined by

$$conv(S) = \{y \mid \forall x, z \in S, x \sqsubseteq y \sqsubseteq z\}$$

For example, the convex closure of the set $\{$`1:`$\bot$`, [1,2,3]`$\}$ includes the list `1:2:`$\bot$ and the list `1:2:3:`$\bot$. Two sets $S_1$ and $S_2$ are considered equivalent iff $conv(S_1) = conv(S_2)$.

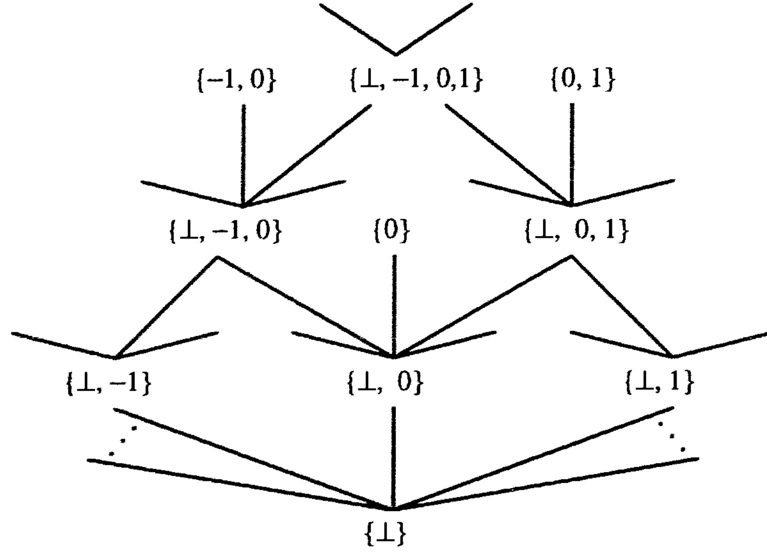


Figure 2.2: Plotkin Power Domain on $\mathbf{Z}_\perp$.

The second problem is that even when such equivalence classes are used some operations on power domains are not continuous. (For example, the singleton set constructor is not continuous.) This can be remedied by modifying the equivalence relation to consider finite elements of finite subsets of the base domain $D$. An element $x$ in $D$ is *finite* iff for every chain $S$ in $D$ with $x \sqsubseteq \bigsqcup S$, we have $x \sqsubseteq z$ for some $z$ in $S$ [8, p. 13]. Let $fin(D)$ be the finite elements of a domain $D$. For sets $S_1$, $S_2 \subseteq D$, the equivalence relation is defined by

$$S_1 =_{\mathcal{P}} S_2 \text{ iff } \forall S \subseteq fin(D), |S| < \infty.(S \sqsubseteq_{\mathcal{P}} S_1 \text{ iff } S \sqsubseteq_{\mathcal{P}} S_2)$$

We take the elements of the Plotkin power domain to be the $\subseteq$-maximal element of each equivalence class. That is,

$$\mathcal{P}(D) = \left(\left\{\bigcup\{S_1 \mid S_1 =_{\mathcal{P}} S\} \;\middle|\; S \subseteq D \text{ and } S \neq \{\}\right\}, \sqsubseteq_{\mathcal{P}}\right)$$

## 2.3 Non-Sequential Functions

The classic example of a non-sequential function is the parallel-or function and it can be defined by the following:

$$por(x, y) = \begin{cases} True, & \text{if } x = True \\ y, & \text{if } x = \perp \text{ or } x = False \end{cases}$$

The *por* function can return a non-bottom and non-constant result when either argument is $\perp$. That is,

$$\begin{aligned} por(x, \perp) &= x \\ por(\perp, x) &= x \end{aligned}$$

Therefore, *por* requires fair evaluation of its arguments to avoid non-termination when evaluation of one of its arguments does not terminate. The fair evaluation may be done sequentially by interleaving the evaluation of each argument or may be done in parallel by concurrently evaluating each argument. The tasks evaluating each argument are speculative in that a task becomes irrelevant if the other task evaluates to true. However, these tasks are unlike the speculative tasks described earlier because they require fair scheduling.

Another typical non-sequential function is a variant of the conditional function *if* and is defined by:

$$pif(c, x, y) = \begin{cases} x, & \text{if } c = \perp \text{ and } x \neq \perp \text{ and } x = y \\ \perp, & \text{if } c = \perp \text{ and } (x = \perp \text{ or } x \neq y) \\ x, & \text{if } c = True \\ y, & \text{if } c = False \end{cases}$$

An implementation of *pif* requires fair evaluation of the condition $c$ and the test $x = y$.

The *por* function appears to be a good candidate for writing parallel search programs because it could be used in decision problems to search the children of a node in parallel. However, any functional language with a fixed sequential reduction strategy cannot implement the *por* function because it requires fair evaluation of its arguments.

Functional languages can be extended so that the *por* function could be implemented. For example, the *por* function can be implemented using **amb** by the following:

```
> por x y = amb (cor x y) (cor y x)
```

where `cor` is the conditional or function. However, we want to avoid the overheads of fair scheduling. We also want to be able to write functional programs that can be written and debugged using existing sequential compilers and then run on a parallel machine for better performance. Therefore, we want a variant of the *por* function that is sequential but can take advantage of parallelism when it is available. Chapter 4 describes how partial determinism can be used to define an approximation to *por* that achieves the above goals.

## 2.4 Equational Reasoning

Equational reasoning is a method of program transformation that uses the algebraic properties of functional programs. Verifying a program using equational reasoning is done by transforming the program to its specification (or vice versa).

Equational reasoning is particularly suited to functional programs because the lack of side effects results in many simple algebraic properties. One simple but important property is the equivalence of identical expressions. That is, for any expression `x`,

$$\texttt{x} = \texttt{x}.$$

Note that this property does not hold in languages with side-effects (in C, for example, it is not true that `i++ == i++`). A slightly more complex example is the following property

$$\texttt{(map f) . (map g)} = \texttt{map (f . g)} \tag{2.1}$$

where `.` is an infix function denoting function composition. This property says that applying `g` to each element of a list followed by applying `f` to each element of the result is the same as applying `f . g` to the list.

Bird and Wadler [7] contains many examples of such algebraic properties and their application. Bird and Hughes [6] describe a particular relevant example. They use equational reasoning to derive a sequential alpha-beta program from its specification. However, the final program is not surprising and can be viewed as a straightforward translation of an imperative alpha-beta program.

The addition of non-deterministic operators seriously hampers equational reasoning. The equivalence of identical expressions is no longer valid. For example,

$$\texttt{amb 1 2} \neq \texttt{amb 1 2}$$

because the left-hand side may evaluate to `1` and the right-hand side may evaluate to `2`. Setsoft and Søndergaard [58] examine non-determinism in functional languages in more detail. They define a simple non-deterministic functional language and describe twelve different semantics. Equational reasoning is difficult, using each of the twelve different semantics, because the language either supports unfolding or has simple algebraic properties but not both. Unfolding replaces a function application with the function's body (with a suitable substitution of the arguments for the parameters) and is a key technique in equational reasoning.

Our approach is to reason with deterministic approximations to non-deterministic programs. *Approximate reasoning* extends equational reasoning to include reasoning with approximations of the form `e1` $\sqsubseteq$ `e2`. There are several factors that contribute to approximate reasoning. The approximates relation is similar to equality in that it is reflexive and transitive. Secondly, if `f` is any computable function, and `e1` $\sqsubseteq$ `e2` then `f e1` $\sqsubseteq$ `f e2` follows from the monotonicity of `f`.

Unlike equality, the approximates relation `e1` $\sqsubseteq$ `e2` permits the expression `e2` to contain more information that the expression `e1`. For example, if `s` is a specification and `p` is a program then showing that `s` $\sqsubseteq$ `p` is usually sufficient for correctness since it guarantees that results from `p` will be at least as informative as results required by `s`. In other words, we are usually willing to accept programs that meet or exceed their specification.

# Chapter 3

# Improving Intervals

This chapter describes the *improving intervals* abstract data type. Improving intervals are an explicit representation of a sequence of converging intervals. The data type defines minimum and maximum functions that are useful for writing functional search programs in which all pruning occurs within the data type.

Improving intervals are an extension of Burton's *improving values* [14] and the chapter starts with a review of improving values as an introduction to the basic concepts. The improving values data type encapsulates operations on lower bounds while the improving intervals data type handles both lower and upper bounds.

Section 3.3 describes a specification for improving intervals that uses non-sequential functions. However, we do not want to enforce non-sequential behaviour for the reasons described in section 2.3. Therefore, we weaken the specification to allow sequential implementations. An implementation for improving intervals is given in section 3.4. The implementation relies on lazy evaluation and uses lazy lists to represent the sequence of converging intervals. Approximate reasoning is used to prove that the implementation is correct with respect to the specification.

Sections 3.5 and 3.6 include two examples of search programs that use improving intervals: a best-first branch-and-bound program and an alpha-beta program. For each program, we give a specification and use approximate reasoning to show that the program meets its specification.

Improving intervals help to simplify the coding and verification of a search program but it can be difficult to understand the program's behaviour. Section 3.5.1 includes a comparison

```
> abstype impvalue *
>   with
>    iv_exact :: * -> impvalue *
>    iv_lb :: * -> impvalue * -> impvalue *
>    iv_value :: impvalue * -> *
>    iv_min :: impvalue * -> impvalue * -> impvalue *
```

Figure 3.1: The Signature of the Improving Values Abstract Data Type

between the behaviour of a functional branch-and-bound program and imperative branch-and-bound.

Section 3.7 describes how speculative parallelism can be used to execute the functional branch-and-bound program in parallel. The addition of `spec` annotations yields parallel programs but without some care the behaviour of the parallel programs can be unexpectedly poor.

## 3.1 Introduction to Improving Values

Improving values were conceived by Burton [14] as a way of expressing parallel search programs in a functional language. Consider the execution of a branch-and-bound program on some problem instance $(X, f)$. As the search tree is explored, we obtain better-and-better bounds on $f^*(X)$ (the cost of an optimal solution). Such a sequence of bounds can be explicitly represented by a lazy list. For example, the list `[3, 5, 10]` might be generated by a program that first found that $3 \leq f^*(X)$, then found the better bound $5 \leq f^*(X)$, and finally found that $f^*(X) = 10$. Improving values encapsulate operations on lists of lower bounds.

Figure 3.1 shows the signature of the improving value abstract data type. The following describes some of the intuition behind the operations on improving values.

`iv_exact a` is an improving value whose value is exactly `a`. For example, `iv_exact 10` is represented by the list `[10]`.

`iv_lb a x` is an improving value with an initial lower bound of `a` and whose subsequent value is defined by `x`. The improving value `iv_lb 3 [5,10]` is represented by the list `[3,5,10]`. The function `iv_lb` is non-strict in its second argument so that `iv_lb 3` $\perp$

is the list `3:⊥`.

**`iv_value x`** is the exact value in the improving value `x`. For example, `iv_value [3,5,10]` is `10`. This is defined only for improving values having a finite total representation.

**`iv_min x y`** returns an improving value that represents the minimum of the improving values `x` and `y`. The function `iv_min` may return a result without examining all the bounds in its arguments: `iv_min [3] (5:⊥)` may return `[3]` without examining $\perp$.

Throughout this dissertation, the variables `a`, `b`, `c`, etc. are used for values while `x`, `y`, `z`, are used for improving values (or improving intervals).

## 3.2 Introduction to Improving Intervals

The improving intervals abstract data type extends improving values to handle both upper and lower bounds. An improving interval represents a value by a sequence of successively tighter intervals that bound the value. For example, the sequence of intervals,

$$[3, \infty], [3, 10], [5, 10], [7, 7]$$

represents the value 7. We try to avoid some confusion between the notation $[a, b]$ for the inclusive interval of values from $a$ to $b$ and the notation `[a,b]`, for a list with two elements by typesetting the interval notation in a math font while a typewriter font is used for the list notation. As with improving values, the sequence of intervals can be explicitly represented by a lazy list.

Section 3.3 defines a specification for the functions on improving intervals but we give a informal description below. The functions on improving intervals are similar to those on improving values but there are also some additional functions on improving intervals:

**`ii_ub a x`** is an improving interval whose initial upper bound is `a` and whose subsequent value is defined by `x`.

**`ii_max x y`** is an improving interval that is the maximum of the improving intervals `x` and `y`.

The signature of improving intervals is shown in figure 3.2 and is similar to the signature for improving values with the addition of the two new functions.

```
> abstype impint *
>   with
>     ii_exact :: * -> impint *
>     ii_value :: impint * -> *
>     ii_lb :: * -> impint * -> impint *
>     ii_ub :: * -> impint * -> impint *
>     ii_min :: impint * -> impint * -> impint *
>     ii_max :: impint * -> impint * -> impint *
```

Figure 3.2: The Signature of the Improving Intervals Abstract Data Type

The functions **ii_exact**, **ii_lb**, and **ii_ub** are used to construct improving intervals. For example,

1. **ii_exact 7** is denotes a sequence with the single interval $[7, 7]$.
2. **ii_lb 5 (ii_ub 10 (ii_exact 7))** denotes the sequence $[5, \infty], [5, 10], [7, 7]$.

The functions **ii_min** and **ii_max** return an improving interval that is the minimum or maximum of its arguments.

## 3.3 Specification of Improving Intervals

A natural approach to specifying improving intervals is to equate their operations to functions on intervals. However, this is difficult because the natural functions on intervals are non-sequential. Instead, the specification is given using approximations. The approximations constrain the operations on improving intervals with respect to minimum and maximum functions on values. We call this an *approximate semantics*.

The section is organized as follows. We start with a more precise definition of intervals, the approximates relation $\sqsubseteq_{\mathcal{I}}$ on intervals, and the minimum and maximum functions on intervals. We then give the specification of improving intervals using approximations. Finally we derive some additional properties about improving intervals from the specification.

The domain of intervals is constructed from a flat domain $V$ with a linear ordering $\leq$. We assume that $\leq$ is strict and monotonic. For any subset $S$ of non-bottom elements of $V$, let $MIN(S)$ be the greatest lower bound on $S$ with respect to $\leq$ and let $MAX(S)$ be the least upper bound on $S$ with respect to $\leq$. For reasons described later, we extend $V$ to

include the $MIN(S)$ and $MAX(S)$ of any subset of non-bottom elements of $V$. That is,

$$V^* = V \cup \{MIN(S) \mid S \subseteq V - \{\bot\}\} \cup \{MAX(S) \mid S \subseteq V - \{\bot\}\}$$

Let $-\infty = MIN(V - \{\bot\})$ and $\infty = MAX(V - \{\bot\})$. The ordering $\leq$ extends to $V^*$ in the obvious way. For example, that rational numbers $\mathbb{Q}$ would be extended as $\mathbb{Q}^* = \mathbb{R} \cup \{-\infty, \infty\}$. We call $V^*$ the set of *values.*

The domain $\mathcal{I}$ of intervals is the set

$$\{[a, b] \mid a, b \in V^* - \{\bot\} \text{ and } a \leq b\},$$

partially ordered by

$$[a, b] \sqsubseteq_{\mathcal{I}} [c, d] \text{ iff } a \leq c \text{ and } b \geq d.$$

Hence, for intervals $i_1$ and $i_2$, we can read $i_1 \sqsubseteq_{\mathcal{I}} i_2$ as $i_2$ is tighter or equal to $i_1$. The least element of $\mathcal{I}$ is $\bot_{\mathcal{I}} = [-\infty, \infty]$. The maximum and minimum functions on intervals are defined point-wise:

$$\begin{aligned} min_{\mathcal{I}}([a, b], [c, d]) &= [\min(a, c), \min(b, d)] \\ max_{\mathcal{I}}([a, b], [c, d]) &= [\max(a, c), \max(b, d)] \end{aligned}$$

where min and max are minimum and maximum functions on values with respect to $\leq$. The functions $min_{\mathcal{I}}$ and $max_{\mathcal{I}}$ are monotonic because max and min are monotonic. That is, if $i_1 \sqsubseteq_{\mathcal{I}} i_1'$ and $i_2 \sqsubseteq_{\mathcal{I}} i_2'$ then $max_{\mathcal{I}}(i_1, i_2) \sqsubseteq_{\mathcal{I}} max_{\mathcal{I}}(i_1', i_2')$.

If we used $V$ rather than $V^*$ in the definition of $\mathcal{I}$ then some chains in $\mathcal{I}$ would not have a least upper bound. For example, if we let $V$ be the rational numbers $\mathbb{Q}$ then the least upper bound of the chain of intervals

$$\left\{ \left[ 1 + \sum_{i=1}^{j} \frac{1}{i!}, 3 \right] \middle| \; j = 1, 2, \ldots \right\}$$

is the interval $[e, 3]$ (where $e = 2.71828$) but $[e, 3] \notin \mathcal{I}$ because $e$ is not in $\mathbb{Q}$. With any data type having a fixed size representation, and hence only a finite number of values, $V$ and $V^*$ are equivalent.

A natural specification for improving intervals defines a function that maps the operations on improving intervals to functions on intervals. For example, the following equations

partially define such a function called $\alpha$.

$$\alpha(\texttt{ii_exact a}) = \begin{cases} [a,a], & \text{if } a \neq \bot \\ \bot_{\mathcal{I}}, & \text{otherwise} \end{cases}$$

$$\alpha(\texttt{ii_max x y}) = max_{\mathcal{I}}(\alpha(\texttt{x}), \alpha(\texttt{y}))$$

$$\alpha(\texttt{ii_lb a y}) = \begin{cases} max_{\mathcal{I}}(\alpha(\texttt{ii_exact a}), \alpha(\texttt{y})), & \text{if } a \neq \bot \\ \bot_{\mathcal{I}}, & \text{otherwise} \end{cases}$$

The function `ii_lb` is specified using $max_{\mathcal{I}}$ because for any lower bound $l$ on a value $x$ it follows that $\max(l, x) = x$.

An implementation of improving intervals would be correct if it satisfied the above equations. However $max_{\mathcal{I}}$ is a non-sequential function (in the sense defined in section 2.3). For example, because

$$max_{\mathcal{I}}([3,3], \bot_{\mathcal{I}}) = [3, \infty]$$

and

$$max_{\mathcal{I}}(\bot_{\mathcal{I}}, [3,3]) = [3, \infty],$$

an implementation of $max_{\mathcal{I}}$ must fairly evaluate both its arguments in order to avoid a possible non-terminating computation. Any implementation of **ii_max** in a traditional functional language must be sequential and can not satisfy the above specification.

Two possible remedies are to extend the functional language so that non-sequential functions are implementable or to weaken the abstract model (intervals plus $min_{\mathcal{I}}$ and $max_{\mathcal{I}}$) so that they corresponds to sequential functions. Both these approaches have problems. Extending the functional language introduces the problems with non-sequential functions mentioned in section 2.3. On the other hand, $min_{\mathcal{I}}$ and $max_{\mathcal{I}}$ are simple and natural functions on intervals[1].

Our approach is to avoid the above dilemma: We weaken the specification by replacing the equations with approximations.

### 3.3.1 Approximate Semantics for Improving Intervals

The approximate semantics for improving intervals re-writes the previous equations using approximation relations. For example,

$$\alpha(\texttt{ii_max x y}) \sqsubseteq_{\mathcal{I}} max_{\mathcal{I}}(\alpha(x), \alpha(y))$$

[1] $max_{\mathcal{I}}$ and $min_{\mathcal{I}}$ are associative, commutative and have absorption and distributive properties.

$$\alpha(\texttt{ii_exact a}) = \begin{cases} [a,a], & \text{if } a \neq \bot \\ \bot_{\mathcal{I}}, & \text{otherwise} \end{cases} \quad (3.1)$$

$$\texttt{ii_value } \bot_{\mathcal{I}} = \bot \quad (3.2)$$

$$\alpha(\texttt{ii_exact (ii_value x)}) \sqsubseteq_{\mathcal{I}} \alpha(\texttt{x}) \quad (3.3)$$

$$\texttt{ii_value (ii_exact a)} = \texttt{a} \quad (3.4)$$

$$\alpha(\texttt{ii_max x y}) \sqsubseteq_{\mathcal{I}} max_{\mathcal{I}}(\alpha(x), \alpha(y)) \quad (3.5)$$

$$\alpha(\texttt{ii_min x y}) \sqsubseteq_{\mathcal{I}} min_{\mathcal{I}}(\alpha(x), \alpha(y)) \quad (3.6)$$

$$\alpha(\texttt{ii_lb a y}) = \begin{cases} max_{\mathcal{I}}(\alpha(\texttt{ii_exact a}), \alpha(\texttt{y})), & \text{if } a \neq \bot \\ \bot_{\mathcal{I}}, & \text{otherwise} \end{cases} \quad (3.7)$$

$$\alpha(\texttt{ii_ub a y}) = \begin{cases} min_{\mathcal{I}}(\alpha(\texttt{ii_exact a}), \alpha(\texttt{y})), & \text{if } a \neq \bot \\ \bot_{\mathcal{I}}, & \text{otherwise} \end{cases} \quad (3.8)$$

$$\texttt{max2 (ii_value x)(ii_value y)} \sqsubseteq \texttt{ii_value (ii_max x y)} \quad (3.9)$$

$$\texttt{min2 (ii_value x)(ii_value y)} \sqsubseteq \texttt{ii_value (ii_min x y)} \quad (3.10)$$

$$\texttt{max2 a (ii_value x)} \sqsubseteq \texttt{ii_value (ii_lb a x)} \quad (3.11)$$

$$\texttt{min2 a (ii_value x)} \sqsubseteq \texttt{ii_value (ii_ub a x)} \quad (3.12)$$

Figure 3.3: Collected Specification For Improving Intervals

This approximation is not strong enough by itself because it is satisfied by an implementation for **ii_max** that always returns $\bot_{\mathcal{I}}$. The specification can be strengthened by giving an approximation that relates **ii_max** to the strict maximum functions. That is,

$$\texttt{max2 (ii_value x)(ii_value y)} \sqsubseteq \texttt{ii_value (ii_max x y)}$$

where **max2** is the strict maximum functions on values. The above approximation constrains the **ii_max** of two improving intervals to be at least as informative as applying **max2** to the values represented by the improving intervals. The use of $\sqsubseteq$ allows **ii_max** to be less strict than **max2** so that **ii_max** can do pruning where **max2** does not.

Figure 3.3 contains the collected specification for improving intervals. The functions **ii_value** and **ii_exact** convert values to and from improving intervals so their composition is related to the identity function, as specified by approximations 3.3 and 3.4. Section 3.4 describes an implementation that meets the above specification.

### 3.3.2 Additional Properties for Improving Intervals

The rest of this section describes some additional properties about improving intervals that follow from the specification in the previous section.

Adding a lower bound to an improving interval does not change its value.

**Lemma 3.13** If a $\leq$ `ii_value x` then `ii_value x` $\sqsubseteq$ `ii_value (ii_lb a x)`

**Proof**

| | | | |
|---|---|---|---|
| `ii_value x` | $=$ | `max2 a (ii_value x)` | {since a $\leq$ `ii_value x`} |
| | $\sqsubseteq$ | `ii_value (ii_lb a x)` | {by eqn 3.11} |

□

Similarly, adding an upper bound does not change the value of an improving interval.

**Lemma 3.14** If a $\geq$ `ii_value x` then `ii_value x` $\sqsubseteq$ `ii_value (ii_ub a x)`

The proof is similar to that for lemma 3.13.

Approximations 3.9 and 3.10 can be extended to lists of improving intervals.

**Lemma 3.15** If `xs` is a non-empty list then

`max (map ii_value xs)` $\sqsubseteq$ `ii_value (foldl1 ii_max xs)`

**Proof**

Since `max (x:xs) = foldl max2 x xs`, we prove that

`foldl max2 (ii_value x)(map ii_value xs)` $\sqsubseteq$ `ii_value (foldl ii_max x xs)`

by induction on `xs`.

**Case** `xs=[]`. Then both the left-hand side and right-hand side are `ii_value x` by the definition of `foldl`.

**Case** `xs=x1:xs'`.

```
foldl max2 (ii_value x)(map ii_value (x1:xs'))
  =  foldl max2 (max2 (ii_value x)(ii_value x1)) (map ii_value xs')
     {defn of foldl)}
  ⊑  foldl max2 (ii_value (ii_max x x1)) (map ii_value xs')
     {by eqn 3.9}
  ⊑  ii_value (foldl ii_max (ii_max x x1) xs')          {induction}
  ⊑  ii_value (foldl ii_max x (x1:xs'))                 {defn of foldl}
```

□

**Lemma 3.16** If `xs` is a non-empty list then

```
min (map ii_value xs)  ⊑  ii_value (foldl1 ii_min xs)
```

The proof is similar to that for lemma 3.15.

Results similar to lemmas 3.15 and 3.16 that use `foldr1` instead of `foldl1` also hold. The proofs are straightforward.

## 3.4 Implementation of Improving Intervals

This section describes an implementation that represents an improving interval by a lazy list of bounds. Each bound is a tuple consisting of the lower bound and the upper bound. The implementation emphasizes clarity and simplicity. Appendix A describes how to transform the implementation into a more efficient version (though the improvement in efficiency is a constant factor). We describe some preliminary experiments on the performance of this approach in appendix B.

First, we define a type to represent values. The main purpose in defining this type is to provide two identifiers that explicitly represent $-\infty$ and $\infty$. The type `values *` is defined in figure 3.4. The identifiers `Neginf` and `Inf` represent $-\infty$ and $\infty$. We assume that `<`, `=`, `>` are functions of type `* -> * -> bool` that correspond to the linear order on values[2]. The functions `v_lt`, `v_leq`, `v_gt`, etc. extend this ordering to the type `value *`. If `f` is a function then `$f` is Miranda's notation for the infix operator that corresponds to `f`. The functions `v_max` and `v_min` take two `values` and return the maximum or minimum based on the above ordering.

[2] Miranda automatically defines `<`, `=`, and `>` for all non-function types.

```
> value * ::= Neginf | V * | Inf

> v_leq Neginf y = True
> v_leq (V a) Neginf = False
> v_leq (V a) (V b) = (a<=b)
> v_leq (V a) Inf = True
> v_leq Inf Inf = True
> v_leq Inf x = False

> a $v_lt b = a $v_leq b & a~=b
> a $v_geq b = b $v_leq a
> a $v_gt b = b $v_lt a

> v_max x y = y, if x $v_leq y
>           = x, otherwise
> v_min x y = x, if x $v_leq y
>           = y, otherwise
```

Figure 3.4: Implementation of Values

A bound is a tuple consisting of the lower bound and the upper bound. Figure 3.5 gives the code that defines a type `bnd *` for bounds and some useful functions on bounds. Each component of a bound has the type `value *` so that `(V a,Inf)` is a bound that represents the interval $[a, \infty]$. We say a bound `(l,u)` is *valid* iff $\mathtt{l} \leq \mathtt{u}$. Hence, neither component of a valid bound can be $\perp$.

The functions `lb` and `ub` take a bound and return the lower bound and upper bound, respectively. The constant `b_bot` is the least informative bound. The functions `b_isExact` and `b_nonExact` check if a bound represents an interval with a single value. The function `b_tighter_or_eq` returns `True` if its first argument is a bound that is tighter or equal to its second argument. Finally, `b_min` and `b_max` are the minimum and maximum functions on bounds. There is an obvious correspondence between bounds and intervals. The key difference is that `b_min` and `b_max` are strict in both arguments while $max_{\mathcal{I}}$ and $min_{\mathcal{I}}$ are non-strict.

An improving interval is represented by an infinite list of bounds where the bounds are monotonically tighter (or equal). The bounds in the list are constant once an exact bound

```
> bnd * == (value *, value *)

> ub (l,u) = u
> lb (l,u) = l

> b_bot = (Neginf, Inf)
> b_isExact b = (lb b = ub b)
> b_nonExact b = ~ (b_isExact b)

> b_tighter_or_eq b1 b2 = (lb b1) $v_geq (lb b2) &
>                         (ub b1) $v_leq (ub b2)

> b_min b1 b2 = (v_min (lb b1) (lb b2), v_min (ub b1) (ub b2))
> b_max  b1 b2  = (v_max (lb b1) (lb b2), v_max (ub b1) (ub b2))
```

Figure 3.5: Implementation of Bounds

is found. For example, the list

```
(V 1,Inf):(V 1,V 10):(V 3,V 5):(V 3,V 3):(V 3,V 3):...
```

represents the sequence of intervals

$$[1,\infty]\ ,\ [1,10]\ ,\ [3,5]\ ,\ [3,3]$$

Not all lists are valid representations. In particular, a finite list is not valid. The function *valid*, defined below, defines the valid representations more precisely.

$$\begin{aligned}
\mathit{valid}(\bot) &= \mathit{True} \\
\mathit{valid}(\texttt{b:}\bot) &= \texttt{(lb b) \$v_leq (ub b)} \\
\mathit{valid}(\texttt{b1:b2:x}) &= \texttt{(lb b1) \$v_leq (ub b1)}\ \text{and} \\
&\quad\ \texttt{b2 \$b_tighter_or_eq b1}\ \text{and} \\
&\quad\ \mathit{valid}(\texttt{b2:x}) \\
\mathit{valid}(\texttt{[]}) &= \mathit{False} \\
\mathit{valid}(\texttt{[b]}) &= \mathit{False}
\end{aligned}$$

An infinite list is valid if it is the least upper bound of a chain of valid partial lists. Note that the function *valid* is not computable.

```
> impint * == [bnd *]

> ii_exact a = seq (force a) (cycle [(V a,V a)])
> cycle xs = xs' where xs' = xs ++ xs'

> ii_value (b:x) = (v_out (lb b)), if b_isExact b
>                = ii_value x,     otherwise
>                  where v_out (V a) = a
```

Figure 3.6: Implementation of `ii_exact` and `ii_value`

If at some point an improving interval is represented by the list `(b:bs)` then we call `b` the *current* bound and the bounds in `bs` the *subsequent* bounds. The restriction to infinite lists is not crucial but does simplify some of the code and some of the proofs.

Figure 3.6 gives the code that defines a type, `impint *`, to represent improving intervals and the functions `ii_exact` and `ii_value`. Evaluating `ii_exact a` produces an infinite list where each element is `(V a,V a)`. It uses the function `cycle` that takes a (finite) list and turns it into an infinite one. The expression `seq (force a) x` is a Miranda idiom that fully evaluates `a` before returning `x`. It is used to make `ii_exact` strict and to ensure that its argument is either $\perp$ or fully defined. This helps to ensure that the argument to `ii_exact` comes from a flat domain.

The function `ii_value` returns the value represented by an exact bound in the list if it exists. If a list `x` does not contain an exact bound (for example, when the list is partial) then the evaluation of `ii_value x` does not terminate.

The implementation of `ii_max` and `ii_min` is based on "zipping" the arguments into a list of tuples and then mapping `b_max` or `b_min` on this list. The complete code for `ii_max` and `ii_min` is shown in figure 3.7. It uses a more general version of the `zipwith` function called `zipwithord`. The normal `zipwith` function behaves as follows:

```
zipwith f [x1,x2, ...][y1,y2, ...] = [f x1 y1, f x2 y2, ...]
```

If one of the list arguments is longer, excess elements are discarded from it. The `zipwithord` function has an extra parameter, called a *reducing function*, that controls how the lists are combined. For example, the normal `zipwith` could be defined as follows.

```
> zipwith f x y = zipwithord tails f x y
```

```
>                     where tails (x:xs) (y:ys) = (xs, ys)
```

The reducing function takes two lists and returns a pairs of lists. A reducing function must discard a non-exact bound from, at least, one of its arguments. Thus if `red` is a reducing function then `red x y` returns either:

1. `(x,y)`
2. `(tl x,y)`
3. `(x,tl y)`
4. `(tl x,tl y)`

If `red (bx:x) (by:y) = (x',y')`, then we say that `red` *discards* the bound `bx` if `x'` = `x` (similarly for `by` and `y`). Some further restrictions on reducing functions are described in section 3.4.2. These are necessary to ensure that the implementation is correct.

The implementation of `zipwithord` is shown in figure 3.7. The second equation of `zipwithord` handles finite lists. It is not required for implementing improving intervals but it is include to strengthen the analogy with the standard `zipwith` function.

The function `ii_min` uses the reducing function `lb_red` such that `lb_red x y` discards a bound from the argument that has the smaller current lower bound (assuming that neither is an exact bound). For example,

`lb_red ((V 5,V 10):x) ((V 3, V 7):y)` = `((V 5,V 10):x, y)`

because the lower bound of 3 in the second argument is less than the lower bound of 5 in the first argument. The iterative application of `lb_red`, as is done by `zipwithord` (defined in figure 3.7), causes further evaluation of the list with the smaller lower bound. This is how `ii_min` corresponds to the best-first search strategy. The function `ii_max` is similar but uses a reducing function based on the largest upper bound.

Versions of `ii_max` and `ii_min` that correspond to different search strategies can be implemented by using a different reducing function. For example, section 3.4.1 defines versions of `ii_max` and `ii_min` that correspond to the depth-first strategy.

The functions `ii_lb` and `ii_ub` are implemented using `ii_max` and `ii_min`, just as in the specification. The code for `ii_lb` and `ii_ub` is included in figure 3.7. The implementation of

```
> ii_max x y = zipwithord ub_red b_max x y

> ub_red (bx:x) (by:y)
>      = (bx:x,by:y),  if b_isExact bx & b_isExact by
>      = (bx:x, y),    if b_isExact bx & b_nonExact by
>      = (x, by:y),    if b_nonExact bx & b_isExact by
>      = (x, by:y),    if ubx $v_gt uby
>      = (x, y),       if ubx = uby
>      = (bx:x, y),    if ubx $v_lt uby
>        where ubx = ub bx
>              uby = ub by

> ii_min x y = zipwithord lb_red b_min x y

> lb_red (bx:x) (by:y)
>      = (bx:x,by:y),  if b_isExact bx & b_isExact by
>      = (bx:x, y),    if b_isExact bx & b_nonExact by
>      = (x, by:y),    if b_nonExact bx & b_isExact by
>      = (x, by:y),    if lbx $v_lt lby
>      = (x,y),        if lbx = lby
>      = (bx:x,y),     if lbx $v_gt lby
>        where lbx = lb bx
>              lby = lb by

> ii_lb a x = ii_max (ii_exact a) (b_bot:x)
> ii_ub a x = ii_min (ii_exact a) (b_bot:x)

> zipwithord red f (bx:x) (by:y)
>     = f bx by : zipwithord red f x' y'
>       where (x',y') = red (bx:x)(by:y)
> zipwithord red f x y = []
```

Figure 3.7: Implementation of ii_max and ii_min

```
> ii_df_max x y = zipwithord df_red b_max x (b_bot:y)
> ii_df_min x y = zipwithord df_red b_min x (b_bot:y)

> df_red (bx:x) (by:y)
>     = ((bx:x),(by:y)), if b_isExact bx & b_isExact by
>     = (x,(by:y)),      if b_nonExact bx
>     = ((bx:x),y),      otherwise
```

Figure 3.8: Implementation of `ii_df_max` and `ii_df_min`.

`ii_lb a x` applies `ii_max` to `b_bot:x` instead of just `x` so that `ii_lb` is non-strict in its second argument. For example, evaluating `ii_lb 3` ⊥ returns the partial list `(V 3, Inf):`⊥. If the bound `(V 3,Inf)` is sufficient then further evaluation of `x` is not required, otherwise `x` is evaluated to find subsequent bounds.

The functions `ii_max` and `ii_min` are strict in both arguments but can produce an exact value without fully evaluating their arguments. For example, evaluating the expression

`ii_min (ii_exact 5) (ii_ub 3 ⊥)`

produces the list

`(V 5,V 5):zipwithord lb_red b_min (ii_exact 5) ((Neginf,V 3):⊥)`

without evaluating ⊥. This is how pruning occurs in search programs that use improving intervals.

### 3.4.1 A Depth-first Version of `ii_min` and `ii_max`.

Section 3.5 defines a best-first branch-and-bound program that uses the functions `ii_min` and `ii_max`. In this section, we define two new functions on improving intervals that are useful in depth-first search programs. Section 3.6 describes an alpha-beta program that uses these new functions.

The new functions are called `ii_df_max` and `ii_df_min`. They have the same type and specification as `ii_max` and `ii_min`. An implementation for `ii_df_max` and `ii_df_min` is shown in figure 3.8. The implementation of `ii_df_max` uses `zipwithord` with the reducing function `df_red`. This reducing function discards bounds from its first argument until an exact bound is found and then it starts discarding bounds from its second argument.

Therefore, `ii_df_max` completely evaluates its first argument before evaluating its second argument.

The functions `ii_df_max` and `ii_df_min` should be non-strict in their second arguments. However, the function `zipwithord` is strict in both arguments. We make `ii_df_max` and `ii_df_min` non-strict by applying `zipwithord` to `b_bot:y` instead of just `y`.

### 3.4.2 Correctness of the Implementation

This section proves that the above implementation is correct by showing that it satisfies the specification given in figure 3.3. The proof is done in three steps. First, we show that the implementation preserves valid representations of improving intervals. Then we define the function $\alpha$ that maps a representation to an abstract interval. Finally, we show that the approximations in figure 3.3 hold. Throughout the section, we give the proofs only for `ii_lb` and `ii_max`. The proofs for `ii_ub` and `ii_min` are analogous.

The function `ii_max` is implemented using `zipwithord` with the reducing function `ub_red`. Our approach is to prove some general results about `zipwithord` with any reducing function. The results for `ii_max` are just special cases of these more general results. This approach makes it easy to verify variants of `ii_max` (for example `ii_df_max`) that use a different reducing function. Our intention is that a reducing function must progress (by returning the tail of one of its arguments) until both arguments are exact. Let `red` be a function from two lists of bounds to a pair of lists of bounds. Given two lists of bounds `x` and `y`, we say that the function `red` *discards* the bound `b` from `x` if

$$\texttt{b} = \texttt{hd x} \text{ and } \texttt{fst (red x y)} = \texttt{tl x}$$

Similarily, `red` discards the bound `b` from `y` if

$$\texttt{b} = \texttt{hd y} \text{ and } \texttt{snd (red x y)} = \texttt{tl y}$$

The function `red` is a *reducing function* iff for any lists of bounds `x` and `y`: If the current bound on either `x` or `y` is non-exact then there exists a non-exact bound `b` such that `red` discards `b` from `x` or `red` discards `b` from `y`. In other words, applying a reducing function discards a non-exact bound from one of its arguments, unless the current bound on both arguments is exact. The functions `ub_red`, `lb_red`, and `df_red` are obviously reducing functions.

### Proving Valid Representations

We show that the implementation preserves valid representations by showing that `ii_exact`, `ii_lb` and `ii_max` each produce a valid representation when their arguments are valid representations.

**Lemma 3.17** *valid*(`ii_exact a`)

**Proof**

If `a` is $\perp$ then the use of `seq` and `force` causes `ii_exact a` to evaluate to $\perp$ and $\perp$ is a valid representation. Otherwise, `ii_exact a` evaluates to `cycle [(V a,V a)]` which is also a valid representation.

□

**Lemma 3.18** If `red` is a reducing function and `x` and `y` are valid representations then

*valid*(`zipwithord red b_max x y`).

**Proof**

We prove the lemma by structural induction on `x` and `y`.

The base case is when `x` $=\perp$ or `y` $=\perp$ in which case the `zipwithord` application evaluates to $\perp$ which is valid.

Otherwise, let `bx` = `hd x`, `by` = `hd y` and `(x',y')` = `red x y`. Both `x'` and `y'` are valid representations because `x` and `y` are valid.

*valid*(`zipwithord red b_max x y`)
  = *valid*(`b_max bx by:zipwithord red b_max x' y'`)    {by `zipwithord`.1}

The bound `b_max bx by` is valid because `bx` and `by` are valid bounds.

Let `e` = `zipwithord red b_max x' y'`. The expression `e` is valid by induction so either:

1. `e` = $\perp$ and `b_max bx by:`$\perp$ is valid.

2. `e = b_max (hd x') (hd y'):e'` where `e'` is a valid representation. But `hd x'` is tighter or equal to `bx` and `hd y'` is tighter or equal to `by` because `x` and `y` are valid representations. Therefore, `b_max (hd x')(hd y')` is tighter or equal to `b_max bx by`. Hence, `b_max bx by:e` is a valid representation.

□

**Lemma 3.19** If `x` and `y` are valid representations then *valid*(`ii_max x y`).

**Proof**

Follows directly from lemma 3.18.

□

**Lemma 3.20** If `x` is a valid representation then *valid*(`ii_lb a x`).

**Proof**

If `a` = ⊥ then `ii_lb a x` is ⊥ which is valid.

Otherwise, both `ii_exact a` and `b_bot:x` are valid representations. Therefore, the expression `ii_lb a x` is valid because of lemma 3.19 and because

```
ii_lb a x  =  ii_max (ii_exact a) (b_bot:x)
```

□

### Defining the $\alpha$ Function

The $\alpha$ function for improving intervals is easily defined using an auxiliary function $\alpha_b$ that maps a valid bound (`bnd *`) to an interval ($\mathcal{I}$).

$$\begin{aligned}
\alpha_b(\bot) &= [-\infty, \infty] \\
\alpha_b(\texttt{(Neginf,Inf)}) &= [-\infty, \infty] \\
\alpha_b(\texttt{(Neginf,V a)}) &= [-\infty, a] \\
\alpha_b(\texttt{(V a,V b)}) &= [a, b] \\
\alpha_b(\texttt{(V a,Inf)}) &= [a, \infty]
\end{aligned}$$

The implementation never constructs the bounds `(Neginf,Neginf)` or `(Inf,Inf)` so these case have been omitted from the definition of $\alpha_b$. If `b1` and `b2` are valid bounds then the following equations hold.

$$\alpha_b(\texttt{b_max b1 b2}) = max_{\mathcal{I}}(\alpha_b(b1), \alpha_b(b2)) \quad (3.21)$$

$$\alpha_b(\texttt{b_min b1 b2}) = min_{\mathcal{I}}(\alpha_b(b1), \alpha_b(b2)) \quad (3.22)$$

Now, the function $\alpha$ maps valid representations of improving intervals as follows:

$$\alpha(\bot) = [-\infty, \infty] \tag{3.23}$$

$$\alpha(\texttt{b:x}) = \alpha_b(\texttt{b}) \sqcup_{\mathcal{I}} \alpha(\texttt{x}) \tag{3.24}$$

For an infinite representation x, the value of $\alpha(\texttt{x})$ is defined to be the least upper bound of the chain of partial lists that approximate x. Note that $\sqcup_{\mathcal{I}}$ is interval intersection and $\bot$ is the universal interval.

The following corollaries holds from the definition of $\sqcup_{\mathcal{I}}$ .

**Corollary 3.25** If x is a valid representation of an improving interval and x' is any suffix of x then $\alpha(\texttt{x'}) \sqsubseteq_{\mathcal{I}} \alpha(\texttt{x})$.

**Corollary 3.26** If b:x is a valid representation of an improving interval then

$$\alpha_b(\texttt{b}) \sqsubseteq_{\mathcal{I}} \alpha(\texttt{b:x}).$$

The following corollary holds for a representation b:x because if the bound b is exact then all subsequent bounds in x are equal to b. Otherwise, the implementation of ii_value discards b.

**Corollary 3.27** If b:x is a valid representation of an improving interval then

$$\texttt{ii_value (b:x)} = \texttt{ii_value (x)}$$

### Proving the Approximations

We now show that all the equations and approximations in figure 3.3 are satisfied by the implementation.

**Lemma 3.28** The implementation of ii_exact satisfies equation 3.1. That is,

$$\alpha(\texttt{ii_exact a}) = \begin{cases} [a, a], & \text{if } a \neq \bot \\ \bot_{\mathcal{I}}, & \text{otherwise} \end{cases}$$

**Proof**

If a $= \bot$ then the use of `seq` and `force` causes `ii_exact a` to evaluate to $\bot$. Otherwise, `ii_exact a` evaluates to `cycle [(V a,V a)]` and

$$\alpha(\texttt{(V a,V a):(V a,V a):...}) \quad = \quad [a,a]$$

□

**Lemma 3.29** The implementation of `ii_value` satisfies equation 3.2. That is,

$$\texttt{ii_value}\ \bot_{\mathcal{I}} \quad = \quad \bot$$

**Proof**

The implementation of `ii_value x` is strict because it matches its argument with the pattern `(b:x)`.

□

**Lemma 3.30** The implementation of `ii_value` and `ii_exact` satisfy approximation 3.3. That is,

$$\alpha(\texttt{ii_exact (ii_value x)}) \quad \sqsubseteq_{\mathcal{I}} \quad \alpha(\texttt{x})$$

**Proof**

**Case** x $= \bot_{\mathcal{I}}$. The left-hand side is $\bot_{\mathcal{I}}$ so the approximation holds.

**Case** x = `(b:x')` and b is an exact bound.

Then `ii_exact (ii_value (b:x'))` evaluates to `cycle [b]`. And

$$\alpha(\texttt{cycle [b]}) = \alpha(\texttt{b:x'})$$

because all the subsequent bounds in `x'` must be equal to b.

**Case** x = `(b:x')` and b is not an exact bound.

Then `ii_value (b:x')` evaluates to `ii_value x'` and the result holds by induction.

□

**Lemma 3.31** The implementation of ii_value and ii_exact satisfy the equation 3.4. That is,

$$\texttt{ii_value (ii_exact a)} \;=\; \texttt{a}$$

**Proof**

If x = $\perp$ then both sides of the equation are $\perp$. Otherwise,

```
ii_value (ii_exact a)
  =  ii_value ((V a,V a):(V a,V a):...)        {by ii_exact.1}
  =  a                                          {by ii_value.1}
```

$\square$

**Lemma 3.32** If red is a reducing function and x and y are improving intervals then

$$\alpha(\texttt{zipwithord red b_max x y}) \sqsubseteq_{\mathcal{I}} max_{\mathcal{I}}(\alpha(\texttt{x}), \alpha(\texttt{y}))$$

**Proof**

We prove the approximation by induction on the structure of x and y.

The base case is when x or y is $\perp$ in which case the left-hand side is $\perp_{\mathcal{I}}$ and so the approximation is trivially true.

Otherwise, let bx = hd x and by = hd y and (x',y') = red x y. The following approximations hold by corollaries 3.26 and 3.25.

$$\alpha_b(\texttt{bx}) \sqsubseteq_{\mathcal{I}} \alpha(\texttt{x}),\ \alpha_b(\texttt{by}) \sqsubseteq_{\mathcal{I}} \alpha(\texttt{y})$$
$$\alpha(\texttt{x'}) \sqsubseteq_{\mathcal{I}} \alpha(\texttt{x}),\ \alpha(\texttt{y'}) \sqsubseteq_{\mathcal{I}} \alpha(\texttt{y})$$

Now consider the left-hand side of the approximation.

$\alpha(\texttt{zipwithord red b_max x y}))$

$= \alpha(\texttt{b_max bx by:zipwithord red b_max x' y'})$ {by zipwithord.1}

$= \alpha_b(\texttt{b_max bx by}) \sqcup_{\mathcal{I}} \alpha(\texttt{zipwithord red b_max x' y'})$ {by defn of $\alpha$}

The result holds because both arguments to $\sqcup_{\mathcal{I}}$ approximate $max_{\mathcal{I}}(\alpha(\texttt{x}), \alpha(\texttt{y}))$. First,

$\alpha_b(\texttt{b_max bx by}) \sqsubseteq_{\mathcal{I}} max_{\mathcal{I}}(\alpha_b(\texttt{bx}),\alpha_b(\texttt{by}))$ {by equation 3.21}

$\sqsubseteq_{\mathcal{I}} max_{\mathcal{I}}(\alpha(\texttt{x}),\alpha(\texttt{y}))$ {by monotonicity of $max_{\mathcal{I}}$}

Also,

$$\begin{array}{lll} \alpha(\texttt{zipwithord red b_max x' y'}) & & \\ \quad \sqsubseteq_{\mathcal{I}} & max_{\mathcal{I}}(\alpha(\texttt{x'}), \alpha(\texttt{y'})) & \{\text{by induction}\} \\ \quad \sqsubseteq_{\mathcal{I}} & max_{\mathcal{I}}(\alpha(\texttt{x}), \alpha(\texttt{y})) & \{\text{by monotonicity of } max_{\mathcal{I}}\} \end{array}$$

□

**Lemma 3.33** The implementation of `ii_max` satisfies approximation 3.5.

**Proof**

Follows directly from lemma 3.32 because `ii_max x y` = `zipwithord ub_red b_max x y`.

□

**Lemma 3.34** The implementation of `ii_lb` satisfies approximation 3.7.

**Proof**

If `a` is $\perp$ then `ii_lb a x` is $\perp_{\mathcal{I}}$.

Otherwise,

$$\begin{array}{lll} \texttt{ii_lb a x} & & \\ \quad = & \texttt{ii_max (ii_exact a) (b_bot:x)} & \{\text{by } \texttt{ii_lb.1}\} \\ \quad \sqsubseteq_{\mathcal{I}} & max_{\mathcal{I}}([a,a], \alpha(\texttt{b_bot:x})) & \{\text{by lemma 3.33}\} \\ \quad = & max_{\mathcal{I}}([a,a], \alpha(\texttt{x})) & \{\text{because } \alpha(\texttt{x}) = \alpha(\texttt{b_bot:x})\} \end{array}$$

□

**Lemma 3.35** If `red` is a reducing function and `x` and `y` are improving intervals then

$$\texttt{max2 (ii_value x)(ii_value y)} \sqsubseteq \texttt{ii_value (zipwithord red b_max x y)}$$

**Proof**

If `x` or `y` does not contain an exact bound then the left-hand side is $\perp$ and the approximation trivially holds.

Otherwise, both `x` and `y` must contain an exact bound and there are a finite number of bounds before the exact bound. We prove this case by induction on the number of non-exact

bounds in x and y. Applying the reducing function to x and y must decrease the number of non-exact bounds.

Assume (wlog) that

$$\texttt{ii_value x} \geq \texttt{ii_value y}$$

and so the left-hand side of the approximation is `ii_value x`. Using the zipwithord.1, the right-hand side evaluates to

```
ii_value (b_max bx by:zipwithord red b_max x' y')
```

where `hd x` = `bx`, `hd y` = `by`, and `red x y` = `(x',y')`. Let `b` = `b_max bx by`.

The inequality, `lb b` $\leq$ `ii_value x`, holds because `ii_value y` $\leq$ `ii_value x`. Also, `ii_value x` $\leq$ `ub b` holds because `ii_value x` $\leq$ `ub bx`.

The base case occurs when both bx and by are exact bounds and then b must be an exact bound. If b is an exact bound then `lb b` = `ub b` and therefore `lb b` = `ii_value x` = `ub b`.

When bx is a non-exact bound or by is a non-exact bound then b can either be an exact bound (and the above argument applies) or a non-exact bound. Hence, induction is used when either bx is a non-exact bound or by is a non-exact bound and b is a non-exact bound. In this case, the right-hand side further evaluates to

```
ii_value (zipwithord red b_max x' y')
```

where a non-exact bound has been discarded from either x or y to obtain x' and y'. Therefore, by induction,

$$\texttt{max2 (ii_value x')(ii_value y')} \sqsubseteq \texttt{ii_value (zipwithord red b_max x' y')}$$

but `ii_value x'` = `ii_value x` and `ii_value y'` = `ii_value y` so the left-hand side is equal to

$$\begin{array}{lll} \multicolumn{2}{l}{\texttt{max2 (ii_value x)(ii_value y)}} & \\ \sqsubseteq & \texttt{ii_value (zipwithord red b_max x' y')} & \{\text{above}\} \\ = & \texttt{ii_value (b:zipwithord red b_max x' y')} & \{\text{by corollary 3.27}\} \\ = & \texttt{ii_value (zipwithord red b_max x y)} & \end{array}$$

□

**Lemma 3.36** The implementation of ii_max satisfies approximation 3.9.

**Proof**

Follows directly from lemma 3.35 because `ii_max x y` = `zipwithord ub_red b_max x y`.

□

**Lemma 3.37** The implementation of `ii_lb` satisfies approximation 3.11.

**Proof**

```
max2 a (ii_value x)
  =  max2 (ii_value (ii_exact a))(ii_value x)              {by eqn 3.4}
  =  max2 (ii_value (ii_exact a))(ii_value (b_bot:x))      {by corollary 3.27}
  ⊑  ii_value (ii_max (ii_exact a)(b_bot:x))               {by lemma 3.36}
  ⊑  ii_value (ii_lb a x)                                  {by ii_lb.1}
```

□

**Correctness of The Depth-first Versions**

The functions `ii_df_max` and `ii_df_min` are depth-first versions of the functions `ii_max` and `ii_min`. Section 3.4.1 described an implementation of `ii_df_min` and `ii_df_max` using `zipwithord` with the reducing function `df_red`. The function `df_red` is a reducing function so lemmas 3.18, 3.32, and 3.35 can be used to prove that the approximations that hold for `ii_max` and `ii_min` also hold for `ii_df_max` and `ii_df_min`. That is,

$$\alpha(\texttt{ii_df_min x y}) \sqsubseteq_{\mathcal{I}} min_{\mathcal{I}}(\alpha(x), \alpha(y)) \quad (3.38)$$

$$\alpha(\texttt{ii_df_max x y}) \sqsubseteq_{\mathcal{I}} max_{\mathcal{I}}(\alpha(x), \alpha(y)) \quad (3.39)$$

and

$$\texttt{max2 (ii_value x)(ii_value y)} \sqsubseteq \texttt{ii_value (ii_df_max x y)} \quad (3.40)$$

$$\texttt{min2 (ii_value x)(ii_value y)} \sqsubseteq \texttt{ii_value (ii_df_min x y)} \quad (3.41)$$

The extension of approximations 3.40 and 3.41 to lists also holds.

**Lemma 3.42** If `xs` is a non-empty list then

$$\texttt{max (map ii_value xs)} \sqsubseteq \texttt{ii_value (foldl1 ii_df_max xs)}$$

```
> opt r = (cost r, r),                   if has_direct_solution r
>       = min (map opt (children r)), otherwise
```

Figure 3.9: A Specification for Branch-and-Bound

```
> bb r = ii_value (bb' r)
> bb' r
>    = ii_exact (cost r, r),        if has_direct_solution r
>    = (ii_lb l . ii_ub u) (exp r), otherwise
>      where
>         l = (lb r, l_node)
>         u = (ub r, u_node)
>         exp r = foldl1 ii_min (map bb' (children r))
```

Figure 3.10: Best-first Branch-and-Bound Using Improving Intervals

**Lemma 3.43** If `xs` is a non-empty list then

$$\texttt{min (map ii_value xs)} \sqsubseteq \texttt{ii_value (foldl1 ii_df_min xs)}$$

## 3.5 Branch-and-Bound with Improving Intervals

A specification for branch-and-bound on minimization problems is shown in figure 3.9. It is based on the function $opt(X, f, b)$ defined in section 1.1 but maps a search tree node to a tuple consisting of the cost an optimal solution and the optimal solution node. The specification relies on ordering tuples as: `(a,b)<(c,d)` iff `a<c` or (`a=c` and `b<d`). It also assumes that every leaf node is a solution node[3]. The specification is executable but executing `opt r` generates the entire search tree.

The program in figure 3.10 has the same form as the definition of `opt` but uses improving intervals. The auxiliary function `bb' r` returns an improving interval that represents the sequence of bounds obtained by exploring the sub-tree rooted at `r`. The function `ii_value` is used at the root to extract the optimal cost and solution node.

[3]If a leaf node n is not a solution node then we can set cost $r=\infty$.

The local definitions `l` and `u` are the lower and upper bounds on the node `r`. We assume that `l_node` and `u_node` are two dummy nodes such that for any nodes `r` in the search space,

$$\texttt{l_node} < \texttt{r} < \texttt{u_node}$$

This ensures that `l` < `opt r` < `u`.

When a node `r` does not have a direct solution then `exp r` is the result of expanding `r`. The definition of `exp r` is similar to the second equation of `opt` but substitutes `foldl1 ii_min` for `min`.

The theory of improving intervals requires that the values be elements from a flat domain. In the above program, the values are tuples and tuples are not a flat domain. However, the tuples are of the form `(cost r,r)` and such tuples are isomorphic to a flat domain provided that `r` is always fully defined and `cost r` is defined when `r` is not bottom.

The properties in section 3.3 can be used to prove that `bb` meets the specification `opt`. First, note that

$$\texttt{ii_value (exp r)} \sqsubseteq \texttt{ii_value ((ii_lb l . ii_ub u) (exp r))} \qquad (3.44)$$

follows from lemmas 3.13 and 3.14 because `l` ≤ `exp r` ≤ `u`.

**Theorem 3.45** For any node `r`, `opt r` ⊑ `bb r`

**Proof**

If `r` is not a finite tree then `opt r` is ⊥ and the result trivially holds. Otherwise, `r` has finite height and the proof is by induction on the height of `r`. There are two cases depending on whether `r` has a direct solution.

**Case** If `r` has a direct solution.

```
bb r  =  ii_value (bb' r)                      {by bb.1}
      =  ii_value (ii_exact (cost r, r))       {by bb'.1}
      =  (cost r, r)                           {by eqn 3.4)}
      =  opt r                                 {by opt.1}
```

**Case** If `r` does not have a direct solution then let `kids` = `children n`. Note that `kids`≠`[]` because every leaf node is a solution node.

```
opt r  =  min (map opt kids)                                {by opt.2}
       ⊑  min (map bb kids)                                 {induction}
       =  min (map (ii_value . bb') kids)                   {by bb.1}
       =  min (map ii_value (map bb' kids))                 {map law}
       ⊑  ii_value (foldl1 ii_min (map bb' kids))           {by lemma 3.16}
       =  ii_value (exp r)                                  {by exp.1}
       ⊑  ii_value ((ii_lb l . ii_ub u) (exp r))            {by eqn 3.44}
       =  bb r                                              {by bb.1}
```

□

The proof of correctness relies only on the specification of improving intervals and does not rely on any details of the implementation. With the implementation in section 3.4, `bb` strictly exceeds its specification (that is, `opt` $\sqsubset$ `bb`) because there are infinite search trees for which `bb` terminates but `opt` does not.

It also follows immediately that the obvious depth-first version of this algorithm, using `ii_df_min`, is correct.

### 3.5.1 Operational Behaviour

The previous section proved that `bb` was correct in that it meets the specification `opt`. The operational behaviour of `bb` may not be clear: When and how does pruning occur? How is the least-cost node chosen without using a priority queue? What are the time and space requirements of `bb` compared to imperative branch-and-bound?

The questions are addressed by looking at the behaviour of `bb` on a particular example. A correspondence between the search tree and the functional program graph is used to show that the execution of `bb` goes through a number of iterations where each iteration does the following:

1. Finds the least-cost node.

2. Expands the least-cost node.

3. Propagates a bound to the root of the graph.

Each iteration corresponds to an iteration in the imperative branch-and-bound program. However, we also show that the time taken for some steps within the iteration may be greater than the time for the step in the imperative program.

```
> bb r = ii_value (bb' r)
> bb' r
>    = ii_exact (cost r),     if has_direct_solution r
>    = ii_lb (lb r) (exp r), otherwise
>      where
>         [k1,k2] = children r
>          exp r = ii_min (bb' k1) (bb' k2)
```

Figure 3.11: Simplified Branch-and-Bound Using Improving Intervals

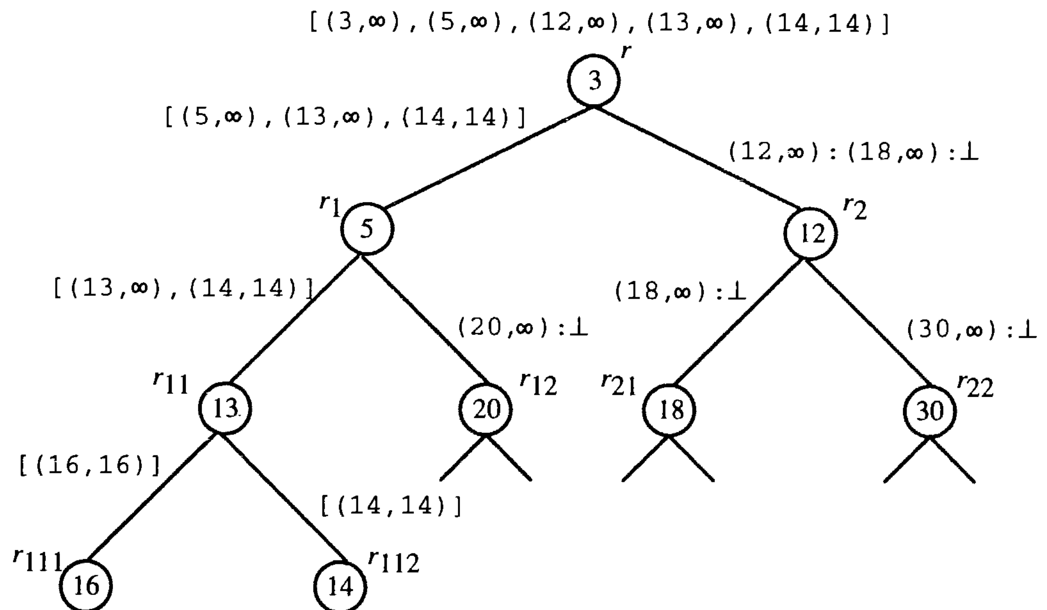


Figure 3.12: Example Search Tree

To simplify the presentation, the slightly different version of bb shown in figure 3.11 is used. The new version of bb differs from the previous one in that each non-solution node must have exactly two children, only lower bounds are used, and just the cost of the optimal solution is returned (rather than the cost plus solution node).

The behaviour of bb is demonstrated on the search tree shown in figure 3.12. The lower bounds are shown within each node. The cost of a leaf node is the same as its lower bound. The optimal solution occurs at node $r_{112}$ with cost 14. With a best-first strategy, the imperative branch-and-bound algorithm expands the nodes in the order: $r$, $r_1$, $r_2$, $r_{11}$ and the nodes $r_{12}$, $r_{21}$, and $r_{22}$ are pruned.

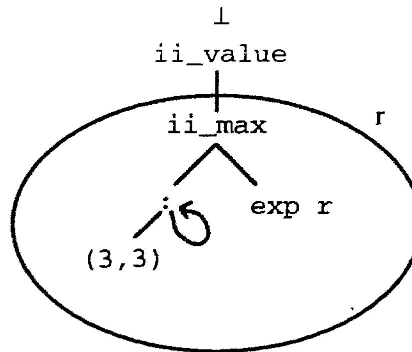


Figure 3.13: Initial Program Graphs in Evaluation of bb

Consider the expression `bb r` where `r` is an expression corresponding to the root of the search tree in figure 3.12. Figure 3.13 to 3.17 show the program graph during reduction of `bb r`. Figure 3.13 shows the program graph after a few reductions. The oval encloses the sub-graph that corresponds to node `r`. The expression at the top of the figure is the list of bounds that have been computed so far (in figure 3.13 no bounds have been computed yet). To further evaluate this graph the expression `exp r` must be evaluated and this evaluation corresponds to expanding the node $r$.

Figure 3.14 shows the result of reducing the expression `exp r` to weak-head normal form. The graph now contains sub-graphs corresponding to the nodes $r_1$ and $r_2$. The bound $(5,\infty)$[4] on $r_1$ has been computed and propagated up to node $r$. The current least-cost node is $r_1$. This iteration is finished when bound $(5,\infty)$ propagates up through the root of the graph and the result is shown in figure 3.15.

The next iteration then finds the least cost node by starting at the root and following the path down the graph while avoiding any node represented by a list-cell. In figure 3.15, for example, we follow the path from the root $r$ to $r_1$ and avoid $r_2$ because it is represented by a list-cell.

The normal order reduction strategy implicitly does this traversal because `ii_min` is strict in both arguments. However, any argument that is a list-cell is in WHNF and therefore it does not need to be evaluated further. This is how the functional program avoids the explicit use of a priority queue. Essentially, each node records which child contains the open node of least cost. Obviously, the time to traverse down the tree to the least-cost node is proportional to the height of the tree.

[4]For readability, we show the bound as $(5,\infty)$ rather than `(V 5, Inf)`.

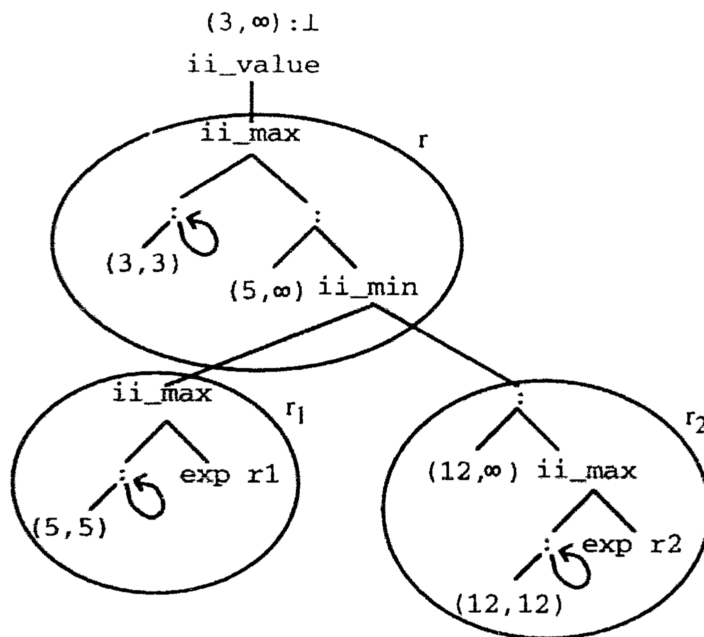


Figure 3.14: After Expanding Node $r$

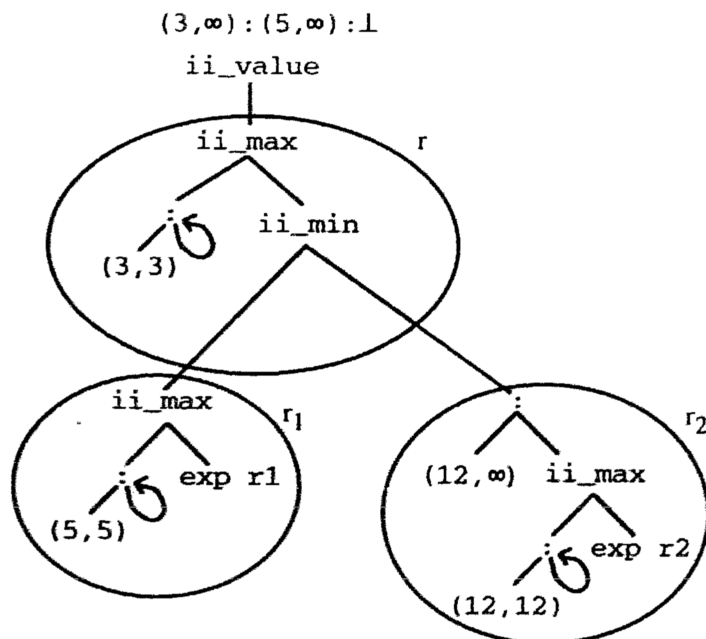


Figure 3.15: After Propagating (5,∞)

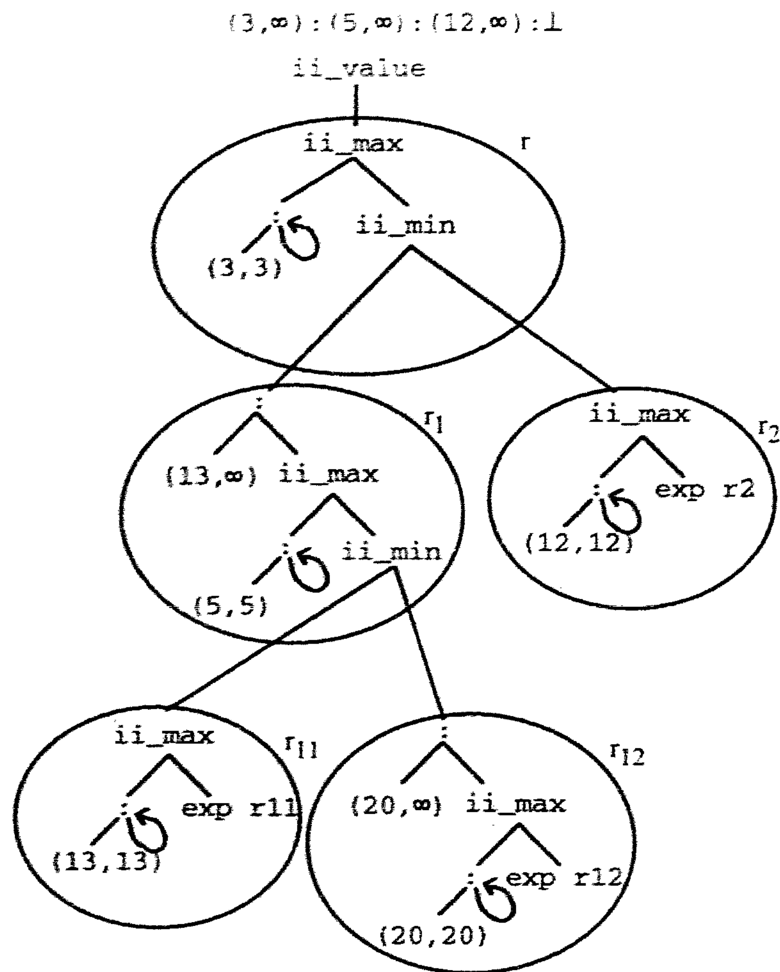


Figure 3.16: After Expanding Node $r_1$ and Propagating (12, ∞)

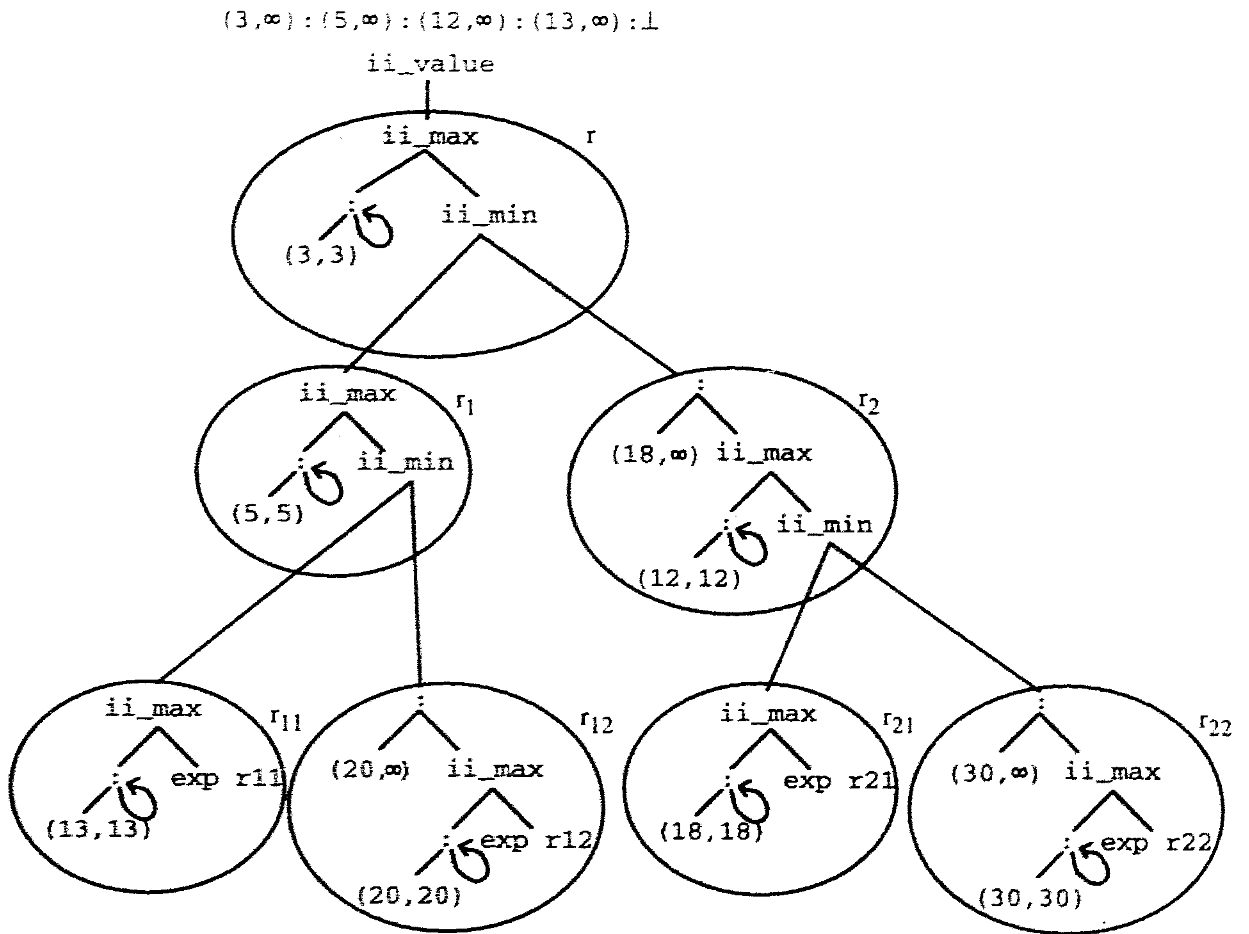


Figure 3.17: After Expanding Node $r_2$ and Propagating $(13, \infty)$

Figure 3.16 shows the program graph after expanding the node $r_1$. This produces the bound of (13,$\infty$) on $r_1$. However, the bound of 12 on $r_2$ is smaller and is propagated through the root making $r_2$ the new least-cost node.

The third iteration expands $r_2$ then propagates the lower bound of 13 from $r_1$ through the root. Figure 3.17 show the program graph after the third iteration. The final iteration expands $r_{11}$ and finds the exact bound (14,14) on $r_{11}$. The exact value propagates up to the root, but since 14 is less than the lower bound of 18 at $r_2$, the sub-tree rooted at $r_2$ is pruned. The expressions associated with the pruned sub-tree become garbage and will be reclaimed. Therefore, the exact bound on the root $r$ is (14,14) and the application of `ii_value` converts it to the value 14.

There are a couple of points about the time and space requirements worth emphasizing. First, the number of iterations taken by `bb` and the imperative program are the same. However, the time taken by each iteration of `bb` can be greater than an iteration in an imperative branch-and-bound program. We assume that the time to compute the bounds and generate the children is the same in each case but consider the time to find the least cost node in a search tree whose height is currently $h$ ($h \geq 0$) and whose branching factor is $b$. Typically, the search tree grows exponentially with its height so $b > 1$. In this case, the number of leaves is $O(b^h)$ so the time to remove the least cost node from the priority queue is $O(h)$ — the same as in the `bb` program. However, it is possible that the priority queue has less than $\Omega(b^h)$ nodes and then the time to remove the least cost node would be less than $h$.

Secondly, the lists used to represent improving intervals never grow long. After an iteration, each non-least-cost open node has one element in its list representation while the least-cost open nodes and closed nodes have no elements in their representation. Therefore, the space used by `bb` should be of the same order as that used by an imperative branch-and-bound program.

## 3.6 Alpha-Beta Using Improving Intervals

The alpha-beta algorithm is the key algorithm in many programs for two-player games. It searches a *game tree* to find the best-move from the current position. Nodes in a game tree represent positions and the children of a node are nodes that represent positions that result from making a single move. The leaf nodes represent positions that end the game.

```
> minmax r
>   = (cost r, r),                      if has_direct_solution r
>   = max (map minmax (children r)), if my_turn r
>   = min (map minmax (children r)), otherwise
```

Figure 3.18: Specification of the Minimax Value of a Game Tree

The alpha-beta algorithm is based on the *minimax* value of a game tree. We assume a game with two players, myself and an opponent. The minimax value of a game tree is the value of the best position in the game tree and is defined recursively as: the value of a leaf node, the maximum value of the children when it is my turn or the minimum value of the children when it is the opponent's turn.

The function **minmax** defined in figure 3.18 is an executable specification for the minimax value of a game tree. It also serves as a specification for an alpha-beta program. We assume some functions on game tree nodes that are similar to the functions used in our branch-and-bound programs:

`has_direct_solution r` is true iff the game position represented by `r` is an obvious win or loss.

`cost r` is the value of the game position `r` when `has_direct_solution r` is true.

**children r** is a list of nodes that represent the positions that result from making a single move from `r`.

**my_turn** `r` is true when `r` is a position where it is my turn to move and is false when it is the opponent's turn to move.

In practise, the size of game trees makes it impractical to consider complete game trees. Game trees are typically cutoff past some depth. We can implement this by defining `has_direct_solution r` to be true if the depth of `r` is greater than some threshold and then `cost r` is an estimate of the value of the game position represented by `r`.

A simple alpha-beta program is constructed from the definition of **minmax** by replacing **max** with **foldl1 ii_df_max** and **min** with **foldl1 ii_df_min**. The resulting program is shown in figure 3.19 The functions **ii_df_min** and **ii_df_max** are used because alpha-beta uses a depth-first strategy to search the game tree.

```
> ab r = ii_value (ab' r)
> ab' r
>  = ii_exact (cost r, r),                        if has_direct_solution r
>  = foldl1 ii_df_max (map ab' (children r)), if my_turn r
>  = foldl1 ii_df_min (map ab' (children r)), otherwise
```

Figure 3.19: Alpha-Beta Program using Improving Intervals

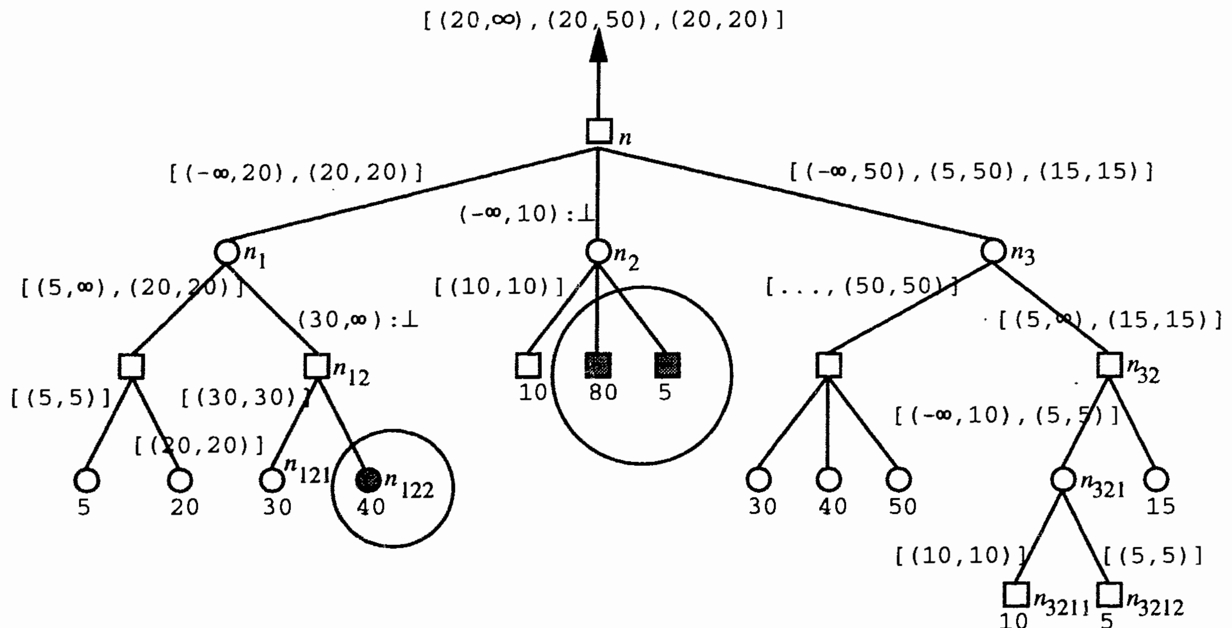


Figure 3.20: Example Game Tree

At a leaf node the value of the node is returned as an improving interval by using `ii_exact` and at the root the value of the game tree is converted from an improving interval to a value by using `ii_value`. The handling of bounds and pruning is encapsulated in the operations on improving intervals.

Figure 3.20 shows a game tree and the improving intervals that are computed at each node. Nodes where it is my turn are drawn with a square box while nodes where it is the opponent's turn are drawn with a circle. The circled portions of the game tree are pruned during the search. Node $n_{122}$ is pruned because the lower bound of 30 from node $n_{121}$ is sufficient to determine that the value of node $n_1$ is exactly 20. Other nodes are pruned in a similar manner.

However, the program misses an opportunity for pruning the node labeled $n_{3212}$. The reasoning that supports the pruning of this node is that we have a lower bound of 20 at the root node; in order to get a better result at the root each of the nodes: $n_3$, $n_{32}$ and $n_{321}$ would have to be greater than 20; but we know $n_{321}$ must be less than 10 without examining $n_{3212}$; therefore $n_{3212}$ can be pruned. This is known as a *deep cutoff*.

The program can be modified to handle deep cutoffs but this would complicate the program. The simplicity of the program in figure 3.19 arises partially because searching a sub-tree is independent of searching any of its siblings[5]. In order to handle deep cutoffs, searching a sub-tree must be made dependent on the result of searching siblings to its left. We prefer the simplicity of the program in figure 3.19 considering that deep cutoffs have not been found to be a major factor in practise [5].

The program `ab` meets its specification, that is,

$$\texttt{minmax} \sqsubseteq \texttt{ab}$$

because

```
max (map ii_value xs) ⊑ ii_value (foldl1 ii_df_max xs)
```

and

```
min (map ii_value xs) ⊑ ii_value (foldl1 ii_df_min xs)
```

This shows that `ab` is correct with respect to `minmax` but it does not show that `ab` does all the pruning that is done by the standard alpha-beta algorithm. In fact, we have shown that `ab` misses deep cutoffs. The pruning behaviour of improving intervals can only be understood by carefully examining the behaviour of the implementation. Approximate reasoning and functional programming do not help with this task.

## 3.7 Adding Speculative Parallelism

Programs written using improving intervals can be made to run in parallel using speculative parallelism. We add `spec` annotations (defined in section 2.1.4) to the branch-and-bound program from figure 3.10 to explicitly indicate the parallelism. However, this section also shows that without some care the parallel programs may not behave as expected.

The program shown in figure 3.21 is a first attempt at a parallel best-first branch-and-bound program and is based on the search programs that appeared in [14] and [32]. The

[5]This also make it easier to execute in parallel.

```
> spec_bb r = ii_value (spec_bb' r)
> spec_bb' r
>     = ii_exact (cost r, r),            if has_direct_solution r
>     = spec (ii_lb l . ii_ub u) (exp r), otherwise
>       where
>        l = (lb r, l_node)
>        u = (ub r, u_node)
>        exp r = foldr1 ii_min (map spec_bb' (children r))
```

Figure 3.21: Parallel Branch-and-Bound Program — First Attempt

`spec` annotation initiates the expansion of a node in parallel with computing the node's bounds. The parallelism is speculative because the bounds may be sufficient to prune the node without expanding it. The program is highly parallel because with an unbounded number of processors all the nodes in the search space are expanded in parallel. The parallel execution occurs because expanding a node applies `spec_bb'` to each child and the recursive application of `spec_bb'` initiates an additional speculative task to expand the child.

A minor problem is that the speculative task that expands a node sequentially evaluates the bounds of each child. This can be easily fixed by changing the definition of `exp r` to use `par_map` rather than `map`.

Another problem is that bounds are produced in parallel but are consumed sequentially. A speculative task that expands a node `r` produces a single bound on `r` then terminates because its result is in WHNF. The single mandatory task is left with the work of consuming all the subsequent bounds. There are likely to be many speculative tasks so it is likely that bounds would be produced at a rate that exceeds the rate that the mandatory task can consume them. In the worst case, the complete search tree is expanded in parallel by speculative tasks that then die and leave a single mandatory task to perform all the remaining work.

A fix is to use the `spec_list` annotation to initiate consumers for subsequent bounds. However, the `spec_list` operates on lists and the list representation of improving intervals is hidden in the abstract data type. We define a function `ii_spec` that uses `spec_list` to initiate a speculative consumer for each bound in an improving interval. Semantically, `ii_spec` is an annotation that denotes the identity function on improving intervals. That is,

```
> ii_spec :: impint * -> impint *
> ii_spec x = x
```

The type specification for `ii_spec` should be added to the signature of improving intervals. Operationally, `ii_spec` initiates the speculative evaluation of all the bounds in an improving interval. The implementation of improving intervals is extended with the following definition for `ii_spec`.

```
> ii_spec xs = spec_list xs
```

Applying `ii_spec` to an improving interval `ii_min x y` causes the bounds produced by the speculative tasks evaluating `x` and `y` to be consumed. Here is the branch-and-bound program, modified to use `ii_spec` rather than `spec`:

```
> spec_bb' r
>     = ii_exact (cost r, r),              if has_direct_solution r
>     = spec (ii_lb l . ii_ub u) (ii_spec (exp r)), otherwise
>       where
>        l = (lb r, l_node)
>        u = (ub r, u_node)
>        exp r = foldr1 ii_min (par_map spec_bb' (children r))
```

A node `r` in the search tree is now associated with a sequence of speculative tasks each of which generates a new bound by consuming a bound from a child of `r`. Thus, each speculative task is both a consumer and a producer and there is now better balance between the consumers and the producers.

The above solution only works well if the speculative tasks at `r` have a higher priority than the speculative tasks at the children of `r` so that the consumers have higher priorities than the producers. The next section adds speculative priorities to ensure this is so.

### 3.7.1 Priorities

It is usually necessary to add priorities to indicate the relative likelihood of each speculative task. This section considers the addition of priorities to speculative tasks using the `priority` annotation from 2.1.4.

A simple priority scheme for the best-first branch-and-bound strategy with a minimization problem is to use the negation of the lower bound as the priority for the tasks associated

```
> spec_bb r = ii_value (spec_bb' r)
> spec_bb' r
>    = ii_exact (cost r, r),                    if has_direct_solution r
>    = spec (ii_lb l . ii_ub u)
>           (priority (-(lb r)) (ii_spec (exp r))), otherwise
>      where
>       l = (lb r, l_node)
>       u = (ub r, u_node)
>       exp r = foldr1 ii_min (par_map spec_bb' (children r))
```

Figure 3.22: Parallel Branch-and-Bound Program — Final Version

with a node. A node with a smaller lower bound is then associated with tasks with a higher priority. This idea can be expressed by re-writing `spec_bb'` as shown in figure 3.22. If the lower bound function is strictly monotonic (for any node `r` and descendant `k` of `r`, `(lb r)<(lb k)`) then the priorities of tasks associated with a node is greater than the priorities of tasks associated with its descendants.

There are two, possibly competing, methods that dictate the order for expanding nodes:

1. The search strategy is determined by the use of `ii_min` versus `ii_df_min` in the program. The use of `ii_min` gives a best-first strategy while the use of `ii_df_min` gives a depth-first strategy. Mixed strategies are possible by combining `ii_min` and `ii_df_min` in the same program. These functions determine the order in which tasks become mandatory.

2. The assignment of priorities to speculative tasks. Nodes with higher speculative priorities may be expanded sooner.

Many variations are possible by using combinations of the above orderings.

For example, a program could use a best-first search strategy but assign speculative priorities in a depth-first manner. Branch-and-bound programs that use mixed strategies have not been examined and it is potential area for future research.

# Chapter 4

# Partial Determinism

This chapter defines partial determinism and demonstrates its use in branch-and-bound programs. A partially deterministic program is non-deterministic: It denotes a set of possible results. However, the set of possible results is restricted so that all the elements in the set are consistent. Section 4.1 defines partial determinism more precisely and includes a few simple examples. Section 4.2 describes the semantics of a simple functional language extended with partially deterministic functions. A major result is that all expressions in the language are partially deterministic, that is, unrestricted non-determinism does not occur.

In section 4.3 we show that partially deterministic programs can be developed and verified using a deterministic program that approximates the partially deterministic program. This is another application of approximate reasoning.

Partially deterministic functions are often useful as approximations to non-sequential functions. They do not require fair evaluation but can take advantage of parallel evaluation when it is available. Section 4.4 introduces a new type of task, called a *partially mandatory* task, for implementing partially deterministic functions.

The application of partial determinism to branch-and-bound programs is discussed in section 4.5. We show how a partially deterministic version of the `ii_df_min` function can prune more of the search space in a parallel depth-first branch-and-bound program. We also show how partial determinism is useful in a sequential branch-and-bound program by describing a branch-and-bound strategy that dynamically adapts to the memory available.

## 4.1 Definition of Partial Determinism

A partially deterministic set is a set whose greatest lower bound and least upper bound exist within the set. Let $\mathcal{P}(D)$ be the Plotkin power domain, as defined in section 2.2.2, over a domain $D$.

**Definition 4.1** A set $S$ in $\mathcal{P}(D)$ is partially deterministic iff $\sqcap S \in S$ and $\bigsqcup S \in S$.

For example, the set $\{\bot, 1\}$, in $\mathcal{P}(\mathbb{N}_\bot)$, is partially deterministic because $\sqcap\{\bot, 1\} = \bot$ and $\bigsqcup\{\bot, 1\} = 1$. The set $\{\bot, 1, 2\}$, however, is not partially deterministic because it does not have an upper bound. The existence of the $\bigsqcup S$ guarantees that there is an element that is consistent with all the elements in the set. The existence of $\sqcap S$ guarantees that all the elements in $S$ are at least as well defined as $\sqcap S$.

The above definition could be applied to any of the three standard power domains (Hoare, Smythe, or Plotkin). However, some interesting simplifications occur when the Plotkin power domain is used. For the rest of the dissertation we consider only partially deterministic sets that are elements of a Plotkin power domain.

The Plotkin power domain divides the subsets of a domain $D$ into equivalences classes based on the equivalence relation:

$$S_1 =_{\mathcal{P}} S_2 \text{ iff } \forall S \subseteq \mathit{fin}(D), |S| < \infty.(S \sqsubseteq_{\mathcal{P}} S_1 \text{ iff } S \sqsubseteq_{\mathcal{P}} S_2)$$

where $S_1$ and $S_2$ are subsets of $D$. The simplification that occurs with the Plotkin power domain is that a partially deterministic set $S$ is equivalent to a two-element set that consists of $\sqcap S$ and $\bigsqcup S$.

**Lemma 4.1** If a set $S$ in $\mathcal{P}(D)$ is partially deterministic then $S =_{\mathcal{P}} \{\sqcap S, \bigsqcup S\}$.

**Proof**

Let $S'$ be any finite subset of $\mathit{fin}(D)$.

First, assume that $S' \sqsubseteq_{\mathcal{P}} S$. Then,

$$\forall a \in S'.\exists b \in S.a \sqsubseteq b$$

But for any such $b$, $b \sqsubseteq \bigsqcup S$ so

$$\forall a \in S'.\exists b \in \{\sqcap S, \bigsqcup S\}.a \sqsubseteq b$$

Similarly,

$$\forall b \in \{\sqcap S, \sqcup S\}.\exists a \in S'.a \sqsubseteq b$$

because

$$\forall b \in S.\exists a \in S'.a \sqsubseteq b$$

and $\sqcap S \in S$ and $\sqcup S \in S$. Therefore $S' \sqsubseteq_{\mathcal{P}} \{\sqcap S, \sqcup S\}$.

Now, assume that $S' \sqsubseteq_{\mathcal{P}} \{\sqcap S, \sqcup S\}$. Then for all $a$ in $S'$, $a \sqsubseteq \sqcup S$. Since $S$ is partially deterministic $\sqcup S$ is in $S$. Therefore,

$$\forall a \in S'.\exists b \in S.a \sqsubseteq b$$

Similarly, there exists an $a$ in $S'$ such that $a \sqsubseteq \sqcap S$ and for all $b$ in $S$, $\sqcap S \sqsubseteq b$ so

$$\forall b \in S.\exists b \in S.a \sqsubseteq b$$

Therefore $S' \sqsubseteq_{\mathcal{P}} S$.

□

If the set $S$ is partially deterministic then we write $\ll \sqcap S, \sqcup S \gg$ for a set that represents the equivalence class that includes $S$. If $\sqcap S = \sqcup S$ then we abbreviate this notation to $\ll \sqcup S \gg$.

The approximates relation on partially deterministic sets is also simplified.

**Lemma 4.2** If $S_1$ and $S_2$ are partially deterministic sets in $\mathcal{P}(D)$ then

$$S_1 \sqsubseteq_{\mathcal{P}} S_2 \text{ iff } \sqcap S_1 \sqsubseteq \sqcap S_2 \text{ and } \sqcup S_1 \sqsubseteq \sqcup S_2$$

**Proof**

Assume that $S_1 \sqsubseteq_{\mathcal{P}} S_2$. Then, $\sqcup S_1 \in S_1$ so $\exists b \in S_2$ such that

$$\sqcup S_1 \sqsubseteq b \sqsubseteq \sqcup S_2$$

Also, $\sqcap S_2 \in S_2$ so $\exists a \in S_1$ such that

$$\sqcap S_1 \sqsubseteq a \sqsubseteq \sqcap S_2$$

Assume that $\sqcap S_1 \sqsubseteq \sqcap S_2$ and $\sqcup S_1 \sqsubseteq \sqcup S_2$. Then

$$\forall a \in S_1.a \sqsubseteq \sqcup S_1 \sqsubseteq \sqcup S_2$$

and $\bigsqcup S_2 \in S_2$. Also,

$$\forall b \in S_2 . \sqcap S_1 \sqsubseteq \sqcap S_2 \sqsubseteq b$$

and $\sqcap S_1 \in S_1$. Therefore, $S_1 \sqsubseteq_{\mathcal{P}} S_2$.

□

It is often useful to consider a partially deterministic set of functions. The set

$$\ll \lambda x.\bot, \lambda x.x \gg$$

is partially deterministic because $\lambda x.\bot \sqsubseteq \lambda x.x$. Section 4.2 describes a simple programming language with identifiers that denote partially deterministic sets of functions. For example, the identifier **maybe** denotes the set $\ll \lambda x.\bot, \lambda x.x \gg$.

There is strong correspondence between a partially deterministic set of functions and a function that returns a partially deterministic set. If $F$ is a partially deterministic set of functions then

$$\lambda x. \ll (\sqcap F)(x), (\bigsqcup F)(x) \gg$$

is the corresponding function that returns a partially deterministic set. We use the term *partially deterministic function* to mean either a partially deterministic set of functions or a function that returns a partially deterministic set.

There are some simple partially deterministic functions that are variations of the boolean *or* function. The following truth table defines a partially deterministic version of the strict or function.

| | $pd_sor(x, y)$ | $y$: $\bot$ | $y$: *False* | $y$: *True* |
|---|---|---|---|---|
| $x$ | $\bot$ | $\ll \bot \gg$ | $\ll \bot \gg$ | $\ll \bot, True \gg$ |
| | *False* | $\ll \bot \gg$ | $\ll False \gg$ | $\ll True \gg$ |
| | *True* | $\ll \bot, True \gg$ | $\ll True \gg$ | $\ll True \gg$ |

It is like the strict or function except for the cases when one argument is $\bot$ and the other argument is *True*. In these cases, the result of *pd_sor* can be $\bot$ or *True*. Thus, *pd_sor* is non-deterministic, but only to the extent that its result is more or less defined. The function *pd_sor* corresponds to the partially deterministic set $\ll sor, por \gg$ where *sor* is the strict or function and *por* is the parallel or function.

The function *pd_sor* is only well-defined using the Plotkin power domain. With the Hoare power domain, the set $\{\perp, True\}$ is equivalent to $\{True\}$ so *pd_sor* is isomorphic to *sor*. With the Smythe power domain, the set $\{\perp, True\}$ is equivalent to $\{\perp, True, False\}$.

The following defines the function *pd_cor* as a partially deterministic version of the conditional or function.

| | $pd_cor(x, y)$ | $y$: $\perp$ | $y$: *False* | $y$: *True* |
|---|---|---|---|---|
| | $\perp$ | $\ll \perp \gg$ | $\ll \perp \gg$ | $\ll \perp, True \gg$ |
| $x$ | *False* | $\ll \perp \gg$ | $\ll False \gg$ | $\ll True \gg$ |
| | *True* | $\ll True \gg$ | $\ll True \gg$ | $\ll True \gg$ |

The *pd_cor* function corresponds to the partially deterministic set $\ll cor, por \gg$ where *cor* is the conditional or function. The difference between *pd_sor* and *pd_cor* occurs when the first argument is *True* and the second argument is $\perp$: $pd_sor(True, \perp) = \ll \perp, True \gg$ while $pd_cor(True, \perp) = \ll True \gg$. The different versions of the or function are related in the following way.

$$\ll sor \gg \sqsubseteq_{\mathcal{P}} \begin{matrix} \ll sor, por \gg \\ \ll cor \gg \end{matrix} \sqsubseteq_{\mathcal{P}} \ll cor, por \gg \sqsubseteq_{\mathcal{P}} \ll por \gg$$

Hence, the functions *pd_sor* and *pd_cor* are approximated by the deterministic functions *sor* and *cor* respectively and approximate the deterministic parallel or function *por*. Unlike the parallel or function, they do not require fair evaluation. They can, however, take advantage of parallelism when it is available. For example, $pd_cor(\perp, True)$ might evaluate its arguments sequentially and return $\perp$ or it might evaluate its arguments in parallel and return *True*.

Both *pd_sor* and *pd_cor* are useful partially deterministic functions. For example, the language Eiffel [42] uses an or function that denotes *pd_sor*. This lets the compiler use commutativity to perform some optimizations but also lets the compiler generate code that "short-circuits" evaluation of boolean expressions[1].

The *pd_cor* function is useful in parallel search programs. A backtracking program for a decision problem can be structured as nested applications of the conditional or function. Speculative parallelism can be used to evaluate the second argument of each *cor* application

[1] In this example the non-deterministic choice is resolved by the compiler so there is no choice to be made at run-time.

in parallel with the evaluation of the first argument. However, a result produced by a speculative task can not be used until the first argument has been evaluated. If *pd_cor* is used in place of *cor* then the program can take better advantage of parallelism by returning *True* as soon as the speculative task produces *True*. Section 4.5 extends this idea to the minimum and maximum functions on improving intervals.

Another interesting partially deterministic function is a partially deterministic version of the conditional.

$$pd_if(c, x, y) = \begin{cases} \ll \bot, x \gg, & \text{if } c = \bot \text{ and } x = y \\ \ll \bot \gg, & \text{if } c = \bot \text{ and } x \neq y \\ \ll x \gg, & \text{if } c = True \\ \ll y \gg, & \text{if } c = False \end{cases}$$

The function *pd_if* corresponds to the partially deterministic set $\ll if, pif \gg$ where *if* is the standard conditional and *pif* is the parallel conditional. Like the parallel conditional, $pd_if(\bot, x, y)$ can return *True* when $x = y$. However, it is not required to return *True* and therefore does not require fair evaluation.

### 4.1.1 Partial Determinism and Function Application

The most important result concerning partial determinism is that it is closed under function application.

**Theorem 4.3** If $F$ in $\mathcal{P}(D_1 \rightarrow D_2)$ is a partially deterministic set of monotonic functions and $S$ in $\mathcal{P}(D_1)$ is a partially deterministic set then the set $\{f(x) \mid f \in F, x \in S\}$ is partially deterministic.

**Proof**

Consider, any $f \in F$ and $x \in S$. We show that $(\sqcap F)(\sqcap S) \sqsubseteq_{D_2} f(x)$. First, $\sqcap F$ is monotonic because $\sqcap F \in F$. Therefore, since $\sqcap S \sqsubseteq_{D_1} x$ we obtain

$$(\sqcap F)(\sqcap S) \sqsubseteq_{D_2} (\sqcap F)(x) \sqsubseteq_{D_2} f(x)$$

Similarly,

$$f(x) \sqsubseteq_{D_2} (\sqcup F)(x) \sqsubseteq_{D_2} (\sqcup F)(\sqcup S)$$

□

The set $\{f(x) \mid f \in F, x \in S\}$ is equivalent to $\ll(\sqcap F)(\sqcap S), (\sqcup F)(\sqcup S)\gg$ by lemma 4.1.

Results analogous to monotonicity also hold for the application of partially deterministic functions.

**Corollary 4.4** If $F$ and $F'$ in $\mathcal{P}(D_1 \rightarrow D_2)$ are partially deterministic sets of monotonic functions such that $F \sqsubseteq_{\mathcal{P}(D_1 \rightarrow D_2)} F'$ and $S$ and $S'$ in $\mathcal{P}(D_1)$ are partially deterministic sets such that $S \sqsubseteq_{\mathcal{P}(D_1)} S'$ then

$$\ll(\sqcap F)(\sqcap S), (\sqcup F)(\sqcup S)\gg \sqsubseteq_{\mathcal{P}(D_2)} \ll(\sqcap F')(\sqcap S'), (\sqcup F')(\sqcup S')\gg$$

## 4.2 A Language with Partially Deterministic Functions

This section shows how partially deterministic functions can be added to a simple programming language. First, we define a simple deterministic functional language by describing its syntax and denotational semantics. We then show how to extend the language by adding some builtin partially deterministic functions. All expressions in the extended language denote partially deterministic sets.

### 4.2.1 A Deterministic Language

The syntax of the language defines an expression as either: a variable, an atomic (builtin) value, a lambda-expression, an application, or a let-expression (possibly recursive). The abstract syntax of a term is described by the following data type:

```
> expr ::=      Var [char] |
>               Atom [char] |
>               Lam [char] expr |
>               App expr expr |
>               Let [char] expr expr
```

We assume that the atomic values include: integer constants (`-2, -1, 0, 1`, etc.), boolean constants (`True, False`), and some standard functions on these types (`if, +, =`, etc.). For example, the expression shown in figure 4.1 might be written more succinctly as

```
let f x = if x = 0 then 1 else (x * (x-1)) in f 5
```

```
Let "f"
  (Lam "x"
    (App (App (App (Atom "if")
      (App (App (Atom "=") (Var "x")) (Atom "0")) )
      (Atom "1") )
      (App (App (Atom "*") (Var "x"))
            (App (App (Atom "-")(Atom "1")) (Atom "x")) ) ) )
  (App (Var "f") (Atom "5"))
```

Figure 4.1: An Expression in the Deterministic Language

The denotational semantics of the language is based on a domain called *value* consisting of functions in *value* $\rightarrow$ *value*, integers, and booleans. Thus, *value* is a domain that satisfies the following domain equation.

$$value = [value \rightarrow value] + \mathbf{Z}_{\perp} + \{True, False\}_{\perp}$$

where + is the coalesced union (sum) on domains[2]. The following functions are used for conversions to and from the domain *value*.

$$\begin{aligned} inF &: (value \rightarrow value) \rightarrow value \\ outF &: value \rightarrow (value \rightarrow value) \\ inN &: \mathbf{Z}_{\perp} \rightarrow value \\ outN &: value \rightarrow \mathbf{Z}_{\perp} \\ inB &: \{True, False\}_{\perp} \rightarrow value \\ outB &: value \rightarrow \{True, False\}_{\perp} \end{aligned}$$

The semantics use an *environment*, a partial mapping of names to values, to record the variable bindings. The empty environment is denoted by $\rho_0$ while $\rho$ denotes an arbitrary environment. The two functions for dealing with environments are:

1. *lookup*: *environment* $\times$ *name* $\rightarrow$ *value*

2. *update*: *environment* $\times$ *name* $\times$ *value* $\rightarrow$ *environment*

[2]The least element of coalesced union, $D_1 + D_2$, is $\perp$ and is equated with $\perp_{D_1}$ and $\perp_{D_2}$.

$$
\begin{aligned}
\mathcal{M}[\![\, \texttt{Var x} \,]\!]\rho &= \mathit{lookup}(\rho, \texttt{x}) \\
\mathcal{M}[\![\, \texttt{Atom n} \,]\!]\rho &= \mathcal{A}[\![\, \texttt{n} \,]\!] \\
\mathcal{M}[\![\, \texttt{Lam v e} \,]\!]\rho &= \mathit{inF}(\lambda x.\mathcal{M}[\![\, \texttt{e} \,]\!](\mathit{update}(\rho, \texttt{v}, x))) \\
\mathcal{M}[\![\, \texttt{App e1 e2} \,]\!]\rho &= \mathit{outF}(\mathcal{M}[\![\, \texttt{e1} \,]\!]\rho)(\mathcal{M}[\![\, \texttt{e2} \,]\!]\rho) \\
\mathcal{M}[\![\, \texttt{Let v e1 e2} \,]\!]\rho &= \mathcal{M}[\![\, \texttt{e2} \,]\!](\mathit{fix}(\lambda \rho'.\mathit{update}(\rho, \texttt{v}, \mathcal{M}[\![\, \texttt{e1} \,]\!]\rho')))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}[\![\, \texttt{"1"} \,]\!] &= \mathit{inN}(1) \\
\mathcal{A}[\![\, \texttt{"2"} \,]\!] &= \mathit{inN}(2) \\
&\;\vdots \\
\mathcal{A}[\![\, \texttt{"True"} \,]\!] &= \mathit{inB}(\mathit{True}) \\
\mathcal{A}[\![\, \texttt{"False"} \,]\!] &= \mathit{inB}(\mathit{False}) \\
\mathcal{A}[\![\, \texttt{"+"} \,]\!] &= \mathit{inF}(\lambda x.\mathit{inF}(\lambda y.\mathit{inN}(\mathit{outN}(x) + \mathit{outN}(y)))) \\
&\;\vdots
\end{aligned}
$$

Figure 4.2: Semantics for a Simple Deterministic Language

where $\mathit{lookup}(\rho, v)$ is the value bound to the variable $v$ in the environment $\rho$ and $\mathit{update}(\rho, v, x)$ is a new environment that is the same as $\rho$ except that variable $v$ is bound to the value $x$.

The main semantic function $\mathcal{M}$ maps an expression and an environment to a element of the domain *value*. It is common to write the application of $\mathcal{M}$ to an expression `e` and an environment $\rho$ as $\mathcal{M}[\![\, \texttt{e} \,]\!]\rho$. Figure 4.2 defines $\mathcal{M}$ using an auxiliary function $\mathcal{A}$ that maps an atomic name to a *value*. Recursive definitions in `Let` expressions are permitted and the semantics of a `Let` expression uses the function *fix* to find fixed points. For simplicity, the semantics does not explicitly handle errors such as trying to apply a number to a number — such invalid expressions are mapped to $\perp$.

### 4.2.2 A Partially Deterministic Language

We now look at extending the deterministic language defined in the previous section to include partially deterministic functions. Our approach is to include some identifiers that denote partially deterministic sets of functions. The syntax of the language is extended to include these identifier by the addition of a constructor `Pd`.

```
> expr ::=      Var [char] |
>               Atom [char] |
```

$$
\begin{aligned}
\mathcal{M}[\![\, \texttt{Var x} \,]\!]\rho &= lookup(\rho, \texttt{x}) \\
\mathcal{M}[\![\, \texttt{Atom n} \,]\!]\rho &= \ll \mathcal{A}[\![\, \texttt{n} \,]\!] \gg \\
\mathcal{M}[\![\, \texttt{Pd n} \,]\!]\rho &= \mathcal{D}[\![\, \texttt{n} \,]\!] \\
\mathcal{M}[\![\, \texttt{Lam v e} \,]\!]\rho &= \ll inF(f), inF(g) \gg \\
&\quad \text{where} \quad f(x) = \sqcap(\mathcal{M}[\![\, \texttt{e} \,]\!](update(\rho, \texttt{v}, \ll x \gg))) \\
&\quad \phantom{\text{where} \quad} g(x) = \sqcup(\mathcal{M}[\![\, \texttt{e} \,]\!](update(\rho, \texttt{v}, \ll x \gg))) \\
\mathcal{M}[\![\, \texttt{App e1 e2} \,]\!]\rho &= \ll (\sqcap F)(\sqcap X), (\sqcup F)(\sqcup X) \gg \\
&\quad \text{where} \quad F = outF(\mathcal{M}[\![\, \texttt{e1} \,]\!]\rho) \\
&\quad \phantom{\text{where} \quad} X = \mathcal{M}[\![\, \texttt{e2} \,]\!]\rho \\
\mathcal{M}[\![\, \texttt{Let v e1 e2} \,]\!]\rho &= \mathcal{M}[\![\, \texttt{e2} \,]\!](fix(\lambda\rho'.update(\rho, \texttt{v}, \mathcal{M}[\![\, \texttt{e1} \,]\!]\rho'))) \\
\\
\mathcal{A}[\![\, \texttt{"1"} \,]\!] &= inN(1) \\
&\vdots \\
\mathcal{D}[\![\, \texttt{"maybe"} \,]\!] &= \ll inF(\lambda x.\bot), inF(\lambda x.x) \gg \\
\mathcal{D}[\![\, \texttt{"pd_cor"} \,]\!] &= \ll inF(vcor), inF(vpor) \gg \\
&\quad \text{where} \quad vcor = \lambda x.inF(\lambda y.inB(cor(outB(x), outB(y))) \\
&\quad \phantom{\text{where} \quad} vpor = \lambda x.inF(\lambda y.inB(por(outB(x), outB(y)))
\end{aligned}
$$

Figure 4.3: Semantics for a Simple Partially Deterministic Language

```
>             Pd [char] |
>             Lam [char] expr |
>             App expr expr |
>             Let [char] expr expr
```

For example, an expression like

```
App (App (Pd "pd_cor") (Atom "True"))(Atom "False")
```

denotes *pd_cor*(*True*, *False*).

The semantics maps expressions to sets of values, that is, elements in the power domain $\mathcal{P}(value)$. The environment is modified to map identifiers to elements in $\mathcal{P}(value)$. The semantics is shown in figure 4.3. It uses a semantic function $\mathcal{D}$ that maps an expression of the form `Pd n` to a partially deterministic set of functions. For example,

$$\mathcal{D}[\![\, \texttt{Pd pd_cor} \,]\!] = \ll cor, por \gg$$

The function $\mathcal{D}$ plays the same role for partially deterministic functions as the function $\mathcal{A}$ plays for deterministic functions.

An important result is that unrestricted non-determinism is not introduced by the presence of partially deterministic functions. We assume that if `n` is an identifier that denotes a partially deterministic set then $\mathcal{D}[\![\ \mathtt{n}\ ]\!]$ is a partially deterministic set.

**Theorem 4.5** If $\rho$ is an environment such that for all names `n`, $lookup(\rho, \mathtt{n})$ is a partially deterministic set then for any expression `e`, $\mathcal{M}[\![\ \mathtt{e}\ ]\!]\rho$ is a partially deterministic set.

**Proof**

We prove the theorem by induction on the structure of the expression `e`.

**Case** e=`Atom n`. The result follows because $\mathcal{M}[\![\ \mathtt{Atom\ n}\ ]\!]\rho$ is a singleton set.

**Case** e=`Var n`. The result follows because $lookup(\rho, \mathtt{n})$ is a partially deterministic set.

**Case** e=`Pd n`. The result follows because $\mathcal{D}[\![\ \mathtt{n}\ ]\!]$ is a partially deterministic set.

**Case** e=`Lam v e1`. By induction, for any value $x$, $\mathcal{M}[\![\ \mathtt{e1}\ ]\!](update(\rho, \mathtt{v}, \ll x \gg))$ is partially deterministic so its greatest lower bound and least upper bound exist. Therefore, $\mathcal{M}[\![\ \mathtt{e}\ ]\!]\rho$ is a partially deterministic function.

**Case** e=`Ap e1 e2`. By induction, $\mathcal{M}[\![\ \mathtt{e1}\ ]\!]\rho$ is a partially deterministic function and $\mathcal{M}[\![\ \mathtt{e2}\ ]\!]\rho$ is a partially deterministic set. Therefore, the result follows by theorem 4.3.

**Case** e=`Let v e1 e2`. First, the environment

$$fix(\lambda\rho'.update(\rho, \mathtt{v}, \mathcal{M}[\![\ \mathtt{e1}\ ]\!]\rho'))$$

is the least fixed point of the chain of environments $\{\rho_0, \rho_1, \rho_2, \ldots\}$, where

$$\begin{aligned} \rho_0 &= \lambda i.\bot \\ \rho_1 &= update(\rho, \mathtt{v}, \mathcal{M}[\![\ \mathtt{e1}\ ]\!]\rho_0) \\ \rho_2 &= update(\rho, \mathtt{v}, \mathcal{M}[\![\ \mathtt{e1}\ ]\!]\rho_1) \\ &\vdots \end{aligned}$$

By induction, each of these environments binds `v` to a partially deterministic set and the least fixed point also bind `v` to a partially deterministic set. Therefore, also by induction,

$$\mathcal{M}[\![\ \mathtt{e2}\ ]\!](fix(\lambda\rho'.update(\rho, \mathtt{v}, \mathcal{M}[\![\ \mathtt{e1}\ ]\!]\rho')))$$

is a partially deterministic set.

□

From the above theorem, it follows immediately that, for any expression e, $\mathcal{M}[\![$ e $]\!]\rho_0$ is partially deterministic.

## 4.3 Reasoning with Partial Determinism

Section 2.4 outlined why equational reasoning is difficult with non-deterministic programs. These difficulties can be avoided with a partially deterministic program by reasoning with a deterministic approximation.

We consider a deterministic program P to be correct with respect to a specification S iff $\mathcal{M}[\![$ S $]\!]\rho_0 \sqsubseteq \mathcal{M}[\![$ P $]\!]\rho_0$. A program is partially deterministic if it contains an occurrence of `Pd n` where n is an identifier that denotes a partially deterministic function. A partially deterministic program P is correct with respect to a specification S iff $\mathcal{M}[\![$ S $]\!]\rho_0 \sqsubseteq_{\mathcal{P}} \mathcal{M}[\![$ P $]\!]\rho_0$. We abbreviate the above relation as S $\sqsubseteq_{\mathcal{P}}$ P.

Our approach for reasoning with a partially deterministic program P is to construct a deterministic program $P_{det}$ such that $P_{det} \sqsubseteq_{\mathcal{P}}$ P and then apply equational reasoning to show that S $\sqsubseteq P_{det}$. We call $P_{det}$ a *deterministic approximation* to the program P. The deterministic approximation $P_{det}$ cannot contain an occurrence of `Pd n` so $\mathcal{M}[\![$ $P_{det}$ $]\!]\rho_0$ equals $\ll x \gg$ for some value $x$. Therefore, $P_{det} \sqsubseteq_{\mathcal{P}}$ P iff $\sqcap P_{det} \sqsubseteq \sqcap$P. Hence, for demonstrating correctness, we are not normally interested in the greatest upper bounds. The existence of the greatest upper bounds ensures consistency which is the key to the simplicity of the semantics of partial determinism.

Fortunately, it is easy to construct a deterministic approximation to any partially deterministic program by replacing each occurrence of a partially deterministic function by a deterministic approximation to the function. For example, if a partially deterministic program contains an occurrence of `Pd "pd_cor"` then a deterministic approximation can be constructed by substituting `Atom "cor"` for `Pd "pd_cor"`. In the following, we use the notation `[x/y]` for a substitution and `e[x/y]` means the simultaneous substitution of x for y in the expression e (we assume that appropriate care is taken to avoid name clashes). The following theorem shows, for the language defined in section 4.2, the substitution `[A n'/Pd n]` can be used to construct a deterministic approximation to a partially deterministic program.

**Theorem 4.6** If P is a program and $\mathcal{A}[\![$ `A n'` $]\!] \sqsubseteq_{\mathcal{P}} \mathcal{D}[\![$ `Pd n` $]\!]$ then P`[A n'/Pd n]` $\sqsubseteq_{\mathcal{P}}$ P

**Proof**

Assume that $\mathcal{A}[\![\ \texttt{A n'}\ ]\!] \sqsubseteq_{\mathcal{P}} \mathcal{D}[\![\ \texttt{Pd n}\ ]\!]$. Let $\phi$ be the substitution [A n'/Pd n]. The proof is by induction on the structure of P.

**Case** P = Var x or Atom n. Then P$\phi$ = P so the theorem is trivially true.

**Case** P = Pd f. If f $\neq$ n then P$\phi$ = P. Otherwise, f = n and P$\phi$ = A n' $\sqsubseteq_{\mathcal{P}}$ P.

**Case** P = Lam v e1.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{M}[\![\ \texttt{(Lam v e1)}\phi\ ]\!]\rho} \\
= & \mathcal{M}[\![\ \texttt{Lam v e}\phi\ ]\!]\rho \\
= & \ll inF(f), inF(g) \gg \\
 & \quad \text{where} \quad f = \lambda x.\sqcap(\mathcal{M}[\![\ \texttt{e1}\phi\ ]\!](update(\rho, \texttt{v}, \ll x \gg))) \\
 & \quad \phantom{\text{where}} \quad g = \lambda x.\sqcup(\mathcal{M}[\![\ \texttt{e1}\phi\ ]\!](update(\rho, \texttt{v}, \ll x \gg))) \\
\sqsubseteq_{\mathcal{P}} & \ll inF(f'), inF(g') \gg \qquad \{\text{induction}\} \\
 & \quad \text{where} \quad f' = \lambda x.\sqcap(\mathcal{M}[\![\ \texttt{e1}\ ]\!](update(\rho, \texttt{v}, \ll x \gg))) \\
 & \quad \phantom{\text{where}} \quad g' = \lambda x.\sqcup(\mathcal{M}[\![\ \texttt{e1}\ ]\!](update(\rho, \texttt{v}, \ll x \gg))) \\
= & \mathcal{M}[\![\ \texttt{Lam v e1}\ ]\!]\rho
\end{array}
$$

**Case** P = Ap e1 e2.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{M}[\![\ \texttt{(Ap e1 e2)}\phi\ ]\!]\rho} \\
= & \mathcal{M}[\![\ \texttt{Ap e1}\phi\ \texttt{e2}\phi\ ]\!]\rho \\
= & \ll (\sqcap outF(\mathcal{M}[\![\ \texttt{e1}\phi\ ]\!]\rho))(\sqcap \mathcal{M}[\![\ \texttt{e2}\phi\ ]\!]\rho), \\
 & \quad (\sqcup outF(\mathcal{M}[\![\ \texttt{e1}\phi\ ]\!]\rho))(\sqcup \mathcal{M}[\![\ \texttt{e2}\phi\ ]\!]\rho) \gg \\
\sqsubseteq_{\mathcal{P}} & \ll (\sqcap outF(\mathcal{M}[\![\ \texttt{e1}\ ]\!]\rho))(\sqcap \mathcal{M}[\![\ \texttt{e2}\ ]\!]\rho), (\sqcup outF(\mathcal{M}[\![\ \texttt{e1}\ ]\!]\rho))(\sqcup \mathcal{M}[\![\ \texttt{e2}\ ]\!]\rho) \gg \\
 & \{\text{induction and theorem 4.3}\} \\
= & \mathcal{M}[\![\ \texttt{Ap e1 e2}\ ]\!]\rho
\end{array}
$$

**Case** P = Let v e1 e2.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{M}[\![\ \texttt{(Let v e1 e2)}\phi\ ]\!]\rho} \\
= & \mathcal{M}[\![\ \texttt{Let v e1}\phi\ \texttt{e2}\phi\ ]\!]\rho \\
= & \mathcal{M}[\![\ \texttt{e2}\phi\ ]\!]\ (fix(\lambda\rho'.update(\rho, \texttt{v}, \mathcal{M}[\![\ \texttt{e1}\phi\ ]\!]\rho'))) \\
\sqsubseteq_{\mathcal{P}} & \mathcal{M}[\![\ \texttt{e2}\ ]\!]\ (fix(\lambda\rho'.update(\rho, \texttt{v}, \mathcal{M}[\![\ \texttt{e1}\ ]\!]\rho'))) \\
 & \{\text{induction and corollary 4.4}\} \\
= & \mathcal{M}[\![\ \texttt{Let v e1 e2}\ ]\!]\rho
\end{array}
$$

□

The deterministic program $P_{det}$ is not only useful for proofs of correctness but also because tools developed for deterministic programs (debuggers, compilers, etc.) can be use with $P_{det}$ as an aid in understanding the partially deterministic program P. In fact, we suspect that P would often be developed by first developing a deterministic program and then replacing some of the deterministic functions by analogous partially deterministic functions.

As an example, consider a small search problem where we are given a finite tree with labeled leaves and we are asked if a particular value occurs as the label of any leaf. A simple specification for this problems is:

```
> tree * ::= Leaf * | Node [tree *]
> occurs v (Leaf l)
>        = True,  if v=l
>        = False, otherwise
> occurs v (Node kids)
>        = foldr sor False (map (occurs v) kids)
```

Two correct partially deterministic programs can be developed by replacing `sor` with either `pd_sor` or `pd_cor`.

## 4.4 Implementation using Partially Mandatory Tasks

Partially deterministic functions can be defined using the non-deterministic `choose` operator. For example, the function *pd_cor* could be defined as:

```
> pd_cor x y =  choose (cor x y) (por x y)
```

However, the above definition requires a non-sequential function and does not capture the intended behaviour of our partially deterministic functions. Our intention is that evaluating `pd_cor x y` might evaluate `x` and `y` in parallel without the requirement of fair evaluation. In a sequential setting, or in a program where the amount of parallelism exceeds the number of processors, `x` maybe evaluated before evaluating `y`. In a parallel setting, evaluating `x` may be done concurrently with evaluating `y`. This section describes how partially deterministic functions can be implemented using a new type of task called a *partially mandatory* task.

The evaluation of pd_cor x y initiates a partially mandatory task to evaluate x while a speculative task may be used to evaluate y. The partially mandatory task must be scheduled ahead of the speculative task but may become irrelevant if y evaluates to *True*.

A partially mandatory task is a task that can either become mandatory or become irrelevant. A partially mandatory task becomes irrelevant, like a speculative task, if its result is not required. However, a partially mandatory task may become mandatory at any point during its execution, a speculative task becomes mandatory only when its results are required. We define a new annotation called pmand for creating partially mandatory tasks. The meaning of the annotation is defined as follows.

```
> pmand f x = ⊥ or f x, if x = ⊥
>           = f x,      otherwise
```

The evaluation of pmand f x creates a partially mandatory task to evaluate x in parallel with the evaluation of f x. If a partially mandatory task does not terminate then the program may or may not terminate.

A partially mandatory task can be used to define a new non-deterministic function called pd_choose.

```
> pd_choose x y =  pmand (amb x) y
```

The pseudo-function pd_choose takes two arguments and non-deterministically returns one of them. The above definition causes pd_choose to behave as follows.

```
> pd_choose ⊥ ⊥ = ⊥
> pd_choose x ⊥ = x
> pd_choose ⊥ y = ⊥ or  y
> pd_choose x y = x or  y
```

The function pd_choose is like amb except that it is bottom avoiding only in its second argument. Therefore, it does not require fair evaluation of its arguments. Evaluating pd_choose x y initiates a partially mandatory task to evaluate x and a speculative task to evaluate y. Each sub-task evaluates its argument to WHNF and the original task returns the result of the first sub-task to complete. The expression pd_choose x y may be evaluated sequentially by returning the result of evaluating x.

The function choose can be defined using two partially mandatory tasks.

```
> choose x y = pmand (pmand amb x) y
```

Evaluation of `choose x y` initiates a partially mandatory task to evaluate each argument and the original task returns the result of the first sub-task to complete.

Many partially deterministic functions can be implemented using `pd_choose`. We describe the implementation of `pd_cor` and `pd_sor` below. Section 4.5.2 describes an implementation of a partially deterministic version of `ii_df_min` using `pd_choose`. Equational reasoning can be preserved if `pd_choose` is not available to the programmer, but is restricted to use in defining standard partially deterministic functions.

The function `pd_cor` can be implemented as[3]

```
> pd_cor x y = pd_choose (cor x y)(cor y x)
```

The evaluation of `pd_cor x y` initiates a partially mandatory task to evaluate `cor x y` and speculative task to evaluate `cor y x`. These tasks must evaluate `x` and `y` respectively because `cor` is strict in its first argument. The partially mandatory task is run in preference to the speculative task but may become irrelevant if `y` evaluates to `True`. An interesting case occurs when `x` is $\perp$ and `y` is `True`. In this case, the partially mandatory task does not terminate and the result depends on whether or not the speculative task is executed. If the speculative task is executed and finds that `y` is `True` then the result is `True`. Otherwise, the evaluation of `pd_cor` does not terminate.

The partially deterministic function `pd_sor` can be implemented using two partially mandatory tasks.

```
> pd_sor x y = choose (cor x y)(cor y x)
```

As soon as either tasks returns a result, the other task becomes irrelevant. However, non-termination by either task may result in a non-terminating program, unlike the case with `pd_cor`.

Priorities can be used with partially mandatory tasks. However, the priority of a partially mandatory task is always higher than the priority of a speculative task.

---

[3] This is very similar to the following definition of `por` using `amb` from section 2.3.

```
por x y = amb (cor x y) (cor y x)
```

## 4.5 Partial Determinism in Branch-and-Bound

This section shows how partial determinism is useful in branch-and-bound programs that use improving intervals. Recall from chapter 3 that the function `ii_df_min` returns the minimum of two improving intervals by evaluating its first argument before evaluating its second argument. A speculative task can be used to evaluate the second argument in parallel with the evaluation of the first argument.

However, consider the following expression

```
ii_value (spec (ii_df_min r1) (ii_spec r2))
  where  r1  =  (ii_lb 5 (ii_ub 10 ⊥))
         r2  =  (ii_lb 4 (ii_ub 6 (ii_exact 4)))
```

This expression mimics depth-first branch-and-bound on a node with two children, `r1` and `r2`. The node `r1` has a lower bound of 5 and an upper bound of 10 and is very time consuming to expand (expanding `r1` does not terminate). The node `r2` has a lower bound of 4, an upper bound of 5, and expands quickly to a solution node whose cost is 4.

The `spec` and `ii_spec` annotations initiate the speculative evaluation of `r2` in parallel with the evaluation of `r1`. One might think that the node `r1` can be pruned after `r2` is expanded because the lower bound of 5 on `r1` is greater than the optimal cost of 4 on `r2`. However, the use of `ii_df_min` forces `r1` to be completely evaluated before the bounds from `r2` are used. Therefore, evaluation of the above expression does not terminate.

We would like a version of the minimum and maximum functions on improving intervals that can use bounds from either argument as soon as they are available. The next section describes how the non-deterministic function `amb` can be used to implement such functions.

### 4.5.1 A Non-Sequential Version of `ii_min` and `ii_max`

Figure 4.4 contains an implementation for the minimum and maximum functions on improving intervals that can use bounds as soon as they become available. Like `ii_max` and `ii_min`, the functions `ii_pmax` and `ii_pmin` are implemented using `zipwithord`. However, they use a reducing function `nd_red` where `nd_red x y` non-deterministically chooses to discard a non-exact bound from either `x` or `y` (provided that both are not exact). If `x` $= \bot$ then `nd_red` returns `(bx:x,y)` while if `y` $= \bot$ then `nd_red` returns `(x,by:y)`.

The function `ii_pmin` can return different representations when applied to the same

```
> ii_pmax x y = zipwithord nd_red b_max (b_bot:x) (b_bot:y)
> ii_pmin x y = zipwithord nd_red b_min (b_bot:x) (b_bot:y)

> nd_red (bx:x)(by:y)
>     = (bx:x,by:y),        if b_isExact bx & b_isExact by
>     = (bx:x,y),           if b_isExact bx & b_nonExact by
>     = (x,by:y),           if b_nonExact bx & b_isExact by
>     = amb (seq x (x,by:y)) (seq y (bx:x,y)), otherwise
```

Figure 4.4: Implementation of a Parallel Min

arguments. Consider evaluating the expression

ii_pmin (ii_lb 5 (ii_ub 10 ⊥)) (ii_lb 4 (ii_ub 6 (ii_exact 4))).

If amb chooses its first argument until it is $\perp$ then the result is

[( $-\infty,\infty$ ),( $-\infty$,10),(4,10),(4,6),(4,4)].

If amb chooses its second argument until it is exact then the result is

[( $-\infty,\infty$ ),( $-\infty$,6),( $-\infty$,4),(4,4)].

However, both results are equivalent in that they both denote the interval $[4,4]$. The use of amb in the implementation of ii_pmin allows ii_pmin to use bounds as soon as they are produced.

The function ii_pmin is non-sequential because it avoids bottom in either argument. For example,

ii_pmin ⊥ (ii_exact 4) = ( $-\infty$,4):⊥ = ii_pmin (ii_exact 4) ⊥

Therefore an implementation of ii_pmin requires fair evaluation of its arguments.

### 4.5.2 A Partially Deterministic Version of ii_df_min and ii_df_max

This section describes a partially deterministic version of ii_df_min called ii_pd_min. The function ii_pd_min is like ii_pmin in that it can use bounds as soon as they are available. However, unlike ii_pmin, it does not require fair evaluation of its arguments.

```
> ii_pd_min x y = zipwithord pd_red b_min x y
> ii_pd_max x y = zipwithord pd_red b_max x y

> pd_red (bx:x) (by:y) =
>     = (bx:x,by:y),           if b_isExact bx & b_isExact by
>     = (bx:x,y),              if b_isExact bx & b_nonExact by
>     = (x,by:y),              if b_nonExact bx & b_isExact by
>     = pd_choose (seq x (x,by:y)) (seq y (bx:x,y)), otherwise
```

Figure 4.5: Implementation of ii_pd_min and ii_pd_max.

The code for the partially deterministic version of ii_df_min is shown in Figure 4.5. The function pd_red is similar to nd_red except that it uses pd_choose instead of amb. If neither bx nor by is an exact bound then evaluating the expression

$$\texttt{pd_red (bx:x) (by:y)}$$

creates a partially mandatory task to evaluate x and creates a speculative task to evaluate y.

The function ii_pd_min is partially deterministic. First,

$$\texttt{ii_pd_min} \sqsubseteq_{\mathcal{P}} \texttt{ii_pmin}$$

because pd_choose $\sqsubseteq_{\mathcal{P}}$ amb so pd_red $\sqsubseteq_{\mathcal{P}}$ nd_red. Furthermore, the only case where

$$\texttt{df_red (bx:x) (by:y)} \neq \texttt{pd_red (bx:x) (by:y)}$$

is when $\texttt{y} \neq \bot$ with

$$\texttt{df_red (bx:x)(by:y)} = \texttt{(x,by:y)}$$

and

$$\texttt{pd_red (bx:x)(by:y)} = \texttt{(bx:x,y)}$$

But then the following hold

$$\begin{aligned} \alpha(\texttt{by:y}) &= \alpha(\texttt{y}) \\ \alpha(\texttt{x}) &\sqsubseteq_{\mathcal{I}} \alpha(\texttt{bx:xs}) \end{aligned}$$

Therefore, in terms of their denotations, ii_df_min approximates ii_pd_min.

The function ii_pd_min is non-deterministic. For example, evaluating the expression

```
ii_pd_min (ii_lb 5 (ii_ub 10 ⊥)) (ii_lb 4 (ii_ub 6 (ii_exact 4)))
```

can result in any of the representations:

1. (−∞,∞):(−∞, 10):⊥
2. (−∞,∞):(−∞, 10):(4,10):⊥
3. (−∞,∞):(−∞, 10):(4,10):(4,6):⊥
4. [(−∞,∞),(−∞, 10),(4,10),(4,6),(4,4)]
5. [(−∞,∞),(−∞,6),(−∞,4),(4,4)]
6. (−∞,∞):(−∞,6):(4,6):⊥
7. (−∞,∞):(4,∞):(4,10):⊥
8. [(−∞,∞),(4, ∞),(4,6):(4,4)]

etc. depending upon when the bounds are produced. Each result denotes an interval that is approximated by $[-\infty, 10]$ and approximates $[4, 4]$.

### 4.5.3 A Partially Deterministic Version of ii_min

The previous sections have shown the utility of a partially deterministic version of **ii_df_min**. Is there useful partially deterministic version of **ii_min** as well?

Partial determinism does not yield the same benefits with the best-first strategy. We can imagine a partially deterministic version of ii_min whose behaviour is between that of ii_min and ii_pmin. That is,

$$\texttt{ii_min} \sqsubseteq_{\mathcal{P}} \texttt{ii_pd_min'} \sqsubseteq_{\mathcal{P}} \texttt{ii_pmin}$$

The main advantage of ii_pd_min x y over ii_df_min x y is that by using a bound from y, ii_pd_min x y may return an exact bound sooner than ii_df_min x y.

This advantage does not occur with ii_min. Consider an expression

```
ii_min x y
```

where the current bound on `x` is `bx` (a non-exact bound) and the current bound on `y` is `by` and `(lb bx)` < `(lb by)`. If `by'` is any subsequent bound in `y` then

$$\texttt{(lb bx)} < \texttt{(lb by)} \leq \texttt{(lb by')} \leq \texttt{(ub by')}$$

Therefore `b_min bx by'` cannot be an exact bound. This is closely related to the well known result that the best-first search strategy never expands a node that another strategy could avoid expanding (assuming that all the bounds are distinct) [44].

## 4.6 Partial Determinism for Adapting to Memory

The previous sections have discussed partial determinism in the context of parallel programs and non-sequential functions. Partial determinism is also useful in strictly sequential situations.

Consider a branch-and-bound program that adapts its search strategy to the memory available. Initially the program uses the best-first strategy but when memory becomes tight it switches to a depth-first strategy. The program's behaviour is non-deterministic: It depends on the amount of memory available. The memory-adaptive program is partially deterministic because its behaviour falls between that of depth-first branch-and-bound and best-first branch-and-bound.

If memory is always tight then executing the memory-adaptive program is equivalent to executing a depth-first program. On the other hand, if memory is never tight then executing the memory-adaptive program is equivalent to executing a best-first program. The interesting cases occur somewhere between these two extremes. We hypothesize that when the amount of available memory is sufficiently large then the memory-adaptive strategy will expand fewer nodes than the depth-first strategy but run in less space than the best-first strategy. The above hypothesis remains to be verified and we do not know what "sufficiently large" really means.

# Chapter 5

# Related Work

This dissertation combines ideas from the fields of functional programming, combinatorial optimization, parallel programming, and non-determinism. There is related work from each of these fields and this chapter gives a brief overview of some of the related work.

## 5.1 Parallel Combinatorial Search

There has been much recent interest in applying parallel processing to search programs. Roucairol [51] and Grama and Kumar [22] survey parallel processing in branch-and-bound while Marsland and Campbell [41] survey work on the parallel search of game trees. The main goal in the above is to improve the performance of parallel search programs while our work has focussed on simplifying parallel search programs.

Searching for solutions is just one approach to solving combinatorial optimization problems. We have not considered other approaches such as: dynamic programming, simulated annealing, or approximation algorithms.

There are three main approaches in writing parallel search programs:

1. Search distinct parts of the search tree in parallel.

2. Apply parallel processing to node operations such as computing the bounds or generating the children. Ebeling [19] describes the application of this approach to chess.

3. In parallel window search[4, 41], each processor explores the search tree using a different window. The lower and upper bound on the root of the tree defines the initial

search window. With $P$ processors, the initial window is split into $P$ subwindows and each processor searches within a distinct subwindow. The speedup using parallel window search in game playing has been found to be 5 or 6 regardless of the number of processors [41].

Our programs use the first approach and the rest of this section discusses the first approach in more detail.

A straightforward method of parallelizing branch-and-bound on $P$ processors is to modify the simple branch-and-bound program in Figure 1.2 such that each iteration expands $P$ nodes in parallel. We call this approach *synchronous* because the processors synchronize after each iteration. Li and Wah [40] use this approach to analyze the number of iterations taken by parallel branch-and-bound.

Parallel branch-and-bound programs can exhibit speedup anomalies [38]: the program can exhibit speedups that are greater than $P$ or less than 1. Anomalies are possible because the parallel program may explore a different search tree than the sequential program. The part of the search tree that is expanded by the parallel program but not by the sequential program is called *search overhead.* Search overhead is analogous to the irrelevant work that may be done by programs that use speculative parallelism. There is a tradeoff between processor utilization and search overhead: using a single processor results in no search overhead but a utilization of only $\frac{1}{P}$. As more processors are used, it is more likely that some of the processors expand nodes that are not expanded by the sequential program. The same tradeoff occurs with speculative parallelism: it increases processor utilization but may result in irrelevant work.

The synchronous approach may perform poorly in practise because of contention for the shared priority queue and delays in waiting for all processors to complete the iteration. In many programs [36, 37, 50], the processors are given their own local priority queues and operate more asynchronously by selecting and expanding nodes from their local priority queues. Some communication between processors is needed so that all processors are aware of the current best solution.

Various load balancing schemes are used to ensure that each processor has nodes to expand and to ensure that promising nodes are fairly distributed among the processors. Large search overhead can occur if some processor only has nodes that are likely to be irrelevant. Two general strategies for load balancing are:

1. Queue splitting: When a processor's priority queue becomes empty it requests work from some target processor. The target processor splits its priority queue and gives part of it to the requesting processor. Many variations are possible depending on how a target is selected and how the queue is split[37].

2. Node splitting: When a node is expanded its children are distributed among the processors. Shu and Kalé [55] use this approach and distribute the children randomly among the processors.

Queue splitting typically involves less communication overhead than node splitting because the processors communicate less often.

Our use of improving intervals results in programs that do not have an explicit priority queue. Instead, the priority queue is implicit in the search tree. It is therefore impossible for the programmer to dictate a queue splitting strategy. However, the run-time system of a parallel functional language must support some load balancing strategy. For example, the Grip system [47] has a local queue of tasks on each processors and exports some tasks from a processor's queue when its local task pool exceeds a fixed size[1]. Hence, our approach simplifies programs by handing over control of load balancing to the run-time system. While this does make the programs simpler, the performance of the programs depends on the run-time system and it is more difficult to experiment with different load balancing strategies.

The node splitting strategy could be programmed within our approach if an additional annotation were available to initiate a task on a specific processor. Both Burton [9] and Hudak [27] proposes annotations for this purpose. For example, the expression `on p f e` behaves like `spec f e` but initiates the speculative task on the processor named `p`.

### 5.1.1 Reducing Search Overhead and Anomalies

Search overhead can be kept low by keeping the number of nodes expanded by the parallel algorithm close to the number of nodes expanded by the sequential algorithm. Kalé and Salvetor [34] describe a parallel depth-first search program that does not exhibit speedup anomalies. They assign priorities to nodes in a manner that is consistent with a depth-first ordering of the search tree. Their program ensures that during the expansion of a node $n_1$, another node of lower priority may be expanded in parallel only if there would otherwise

[1]Grip does not currently support prioritized tasks so the load is balanced only in the sense of balancing the number of tasks on each machine.

be a processor left idle. This is similar to what happens when using speculative parallelism with a depth-first strategy. Any node expanded by the sequential program will eventually become mandatory and when it does its evaluation will be scheduled ahead of speculative tasks. Priorities should also be placed on the speculative task such that the speculative priorities are consistent with a depth-first ordering of the tree.

Akl et al. [1] use a mandatory-work-first scheme for an alpha-beta program. Their strategy exploits the idea that for perfectly ordered game trees, there is a unique minimal search tree that must be expanded by the sequential alpha-beta algorithm. Their algorithm expands this unique search tree in parallel before expanding other nodes. Steinberg and Solomon [59] refine this approach to get better processor utilization. Our approach could be adapted to utilize these strategies by using a combination of mandatory and speculative parallelism.

## 5.2 Speculative Parallelism

There has been relatively little work done on speculative parallelism. Languages for parallel computing do not in general support speculative parallelism very well and few programs explicitly use speculative parallelism. Burton [11] introduces the term "speculative computation" and we use annotations similar to those proposed in [12].

Osborne's thesis [45] describes how speculative parallelism can be supported in Multilisp. He views the major requirements for speculative parallelism as: a priority scheme for directing computation to more promising work and a means to abort and reclaim irrelevant speculative tasks. Osborne proposes a sophisticated priority scheme that supports dynamically changing priorities, priority transitivity (if a task $T_1$ demands the result of a task $T_2$ then priority of $T_2$ should be greater than or equal the priority of $T_1$), and modular priorities. His scheme is certainly more expressive than the simple scheme that we use but it is not clear that the gain in expressiveness out-weighs the overheads involved in its implementation.

Osborne used the travelling salesman problem as a sample problem and describes a simple branch-and-bound program to solve it. The program speculatively expands each node and uses the speculative priorities to control the order of node expansions. However, the program explicitly handles pruning and a global variable is used to record the incumbent. The use of side-effects to update the incumbent makes the program non-deterministic and

not purely functional.

Soley [57] describes the implementation of speculative parallelism in the dataflow language Id. In Soley's programs, `speculate f xs` evaluates the application of `f` on each element in the list `xs` in parallel and returns the result of `f` applied to some element in the list (typically the first to evaluate). Hence, `speculate` is non-deterministic and can be viewed as an extension of `amb` to lists.

Grit and Page [23] and Baker and Hewitt [3] describe methods for reclaiming speculative tasks that have become irrelevant.

Parallel logic programming systems [54] may use or-parallelism to evaluate each clause of a predicate definition in parallel. Or-parallelism is speculative because once any clause is found to be true, the tasks evaluating other clauses become irrelevant. However, these systems do not yet support priorities for controlling speculative parallelism.

Witte [63] shows how speculative parallelism can be applied to simulated annealing. Simulated annealing iteratively makes local improvements and occasionally a random mutation to an initial solution (the probability of a mutation decreases over time). The algorithm is inherently serial because each iteration depends on the solution from the previous iteration. However, speculative parallelism can be used to initiate a next iteration on both the old solution and a new candidate, before it is known which will be chosen. One benefit of using speculative parallelism is that it preserves the sequential sequence of solutions.

Schaeffer [52] describes an interesting use of speculative parallelism in game playing. The speculative tasks are specialists in searching for specific game positions that are particularly beneficial. Often, a specialist will fail to find such a position but when it does the rewards are great. In experiments with chess, Schaeffer found that the addition of speculative specialists increased the search depth by two or more plys.

## 5.3 Non-Determinism

### 5.3.1 Non-deterministic Extensions to Functional Languages

Adding non-deterministic operators hampers equational reasoning with functional programs but functional languages extended with non-deterministic operators have been used in practice to build operating systems [24, 33] and user interfaces [15, 18].

There have also been several proposals to add non-deterministic constructs in ways that preserve equational reasoning. Burton's [13] approach using oracles, and Hughes and

O'Donnell's [31] approach using sets are described below. Both of these approaches permit more general non-determinism but they also have their own shortcomings.

Burton proposes a non-deterministic operator `choice` that is like `amb` but has an extra argument called an oracle that determines how the choice should be made. An oracle can either have the value `One` or the value `Two` and the `choice` function behaves as if it were defined by the following.

```
> choice One x y = x
> choice Two x y = y
```

The oracle is set by the run-time system when the choice must be made and may be set by examining factors not available to the program (such as the actual time when an argument becomes evaluated). Equational reasoning is preserved provided that every instance of `choice` uses a new oracle. Burton proposes that every functional program be given an *infinite* supply of oracles by the addition of an argument that is an lazy infinite tree of oracles.

Hughes and O'Donnell elevate the set of possible results of a non-deterministic choice to the language level. Their functional language includes built-in functions to create and manipulate these sets. Set union is the basic non-deterministic operator and an `amb`-like operator can be defined as:

```
> amb x y = {x} ∪ {y}
```

Equational reasoning is preserved because the program explicitly deals with sets of results. However, these sets are actually represented by a single element so for example the result of `{1} ∪ {2}` is a set represented by either `1` or `2`. They go on to show how these sets are useful for writing and verifying parallel programs. However, with their approach, equational reasoning only demonstrates partial correctness and termination must be demonstrated by other means[2].

### 5.3.2 Other Approaches

Milner's work on CCS [43] and Hoare's work on CSP [25] give more general theories of parallel systems and non-determinism. Their approaches are algebraic and include calculi for reasoning about non-deterministic programs.

[2]Hughes and O'Donnell attempt to develop a calculus for reasoning about termination.

Many treatments of non-determinism include an assumption or requirement of fairness. Apt and Olderheog [2] model parallel programs as non-deterministic programs imbedded in a fair scheduler. Pure logic programming systems must assume each clause is executed fairly so that a proof is found if one exists. The UNITY language [16] assumes that no statement will be ignored infinitely often. We have deliberately avoided a fairness assumption because it introduces overheads in scheduling and makes it more difficult to execute programs sequentially.

There are other approaches in dealing with the difficulties caused by non-determinism. Emrath and Padua [20] try to automate the detection of unwanted non-determinism in parallel programs. Tolmach and Appel [61] instrument non-deterministic programs to produce traces of their execution including the outcome of all non-deterministic choices. The traces are used to re-play the program so that the results are repeatable. This approach can help with debugging a non-deterministic program once a bug has been detected but it does not address the problem of finding bugs in non-deterministic programs.

# Chapter 6

# Conclusions

This dissertation has explored the application of functional programming to combinatorial search programs. Our thesis is that functional programming leads to programs that are simple, are amenable to formal reasoning, and are easy to execute on parallel machines. The dissertation supports the above thesis with some qualifications. This chapter starts with a review of what has been accomplished and describes why the thesis is only partially supported. We follow with a description of some future work and then conclude with a short discussion of the main contributions of this work.

## 6.1 Simpler Programs

We have developed an abstract data type called improving intervals whose minimum and maximum operations encapsulate the pruning behaviour found in many search programs. Search programs that use improving intervals are simple because all pruning is handled within the data type. Lazy evaluation is the key aspect of functional programming that allows pruning to be modularized in an abstract data type. This supports Hughes' claim that lazy evaluation is a new type of "glue" whereby

> "one can modularize programs in new and exciting ways." [30]

However, we have also shown that improving intervals can be implemented using lazy lists and there are a number of languages (Scheme, ML, SISAL) that support lazy lists without being fully lazy. Improving intervals could be implemented and used in these languages as well.

One of the reasons why our functional search programs are simple is that they do not include enhancements found in more complex search programs. For example, our programs assume that the search space is structured as a tree rather than a graph and our programs do not use enhancements such as dominance relations. Including such enhancements would make the programs more complicated and may pose additional difficulties for functional programming.

We have also not considered the details of particular combinatorial problems. In many cases, the code needed to compute bounds, etc., for instances of the problem is more complex than the code for the search program. For example, integer programming problems are often solved using a branch-and-bound program that computes bounds on a sub-instance by solving a linear programming problem. The amount of code required to solve the linear programming problem is to be likely much larger than the amount of branch-and-bound code.

## 6.2 Parallel Execution

Search programs that use improving intervals are easily annotated with a `spec` annotation to initiate speculative tasks that explore distinct sub-trees in parallel. The annotations do not affect the meaning of a program so annotations can be stripped from a functional program without changing its results. This means that programs can be written and debugged on a sequential machine and then executed on a parallel machine for better performance.

The parallelism in search programs is speculative because a task searching a sub-tree may become irrelevant if a solution is found in a different part of the tree. We have assumed that the irrelevant tasks are automatically reclaimed. This leads to simpler programs because the programmer is freed from the details of detecting and killing irrelevant tasks much as garbage collection frees the programmer from the details of memory management.

However, programs using `spec` annotations are deterministic and we have shown that this determinism can prevent some pruning in parallel search programs. We defined the concept of partial determinism to allow programs with some non-determinism. The use of non-determinism means that the programs are not purely functional and it raises the possibility of unpredictable results. Partial determinism restricts non-determinism so that only consistent results can be produced. We have shown that partial determinism is closed under function application so that every expression in a functional language extended with

partially deterministic functions is partially deterministic.

We also demonstrated how partially deterministic functions can be viewed as approximations to non-sequential functions that can use parallelism when it is available but do not require fair evaluation. This means that partially deterministic functions are easy to evaluate sequentially.

We have assumed that some details that make parallel programming difficult, for example mapping tasks onto processors and load balancing, are handled by the functional language. We have also not considered the details of task granularity or communication costs. An advantage of functional programming is that these details can often be ignored or left to the run-time system. However, without considering these factors, the performance of the programs may be poor.

## 6.3 Reasoning with Programs

Approximate reasoning is an extension of equational reasoning to include approximations such as e1 $\sqsubseteq$ e2 in addition to equations. We have shown how approximate reasoning is useful for formally reasoning about search programs. We have used approximate reasoning to verify an implementation of improving intervals and to verify two search programs.

Approximate reasoning is applicable to partially deterministic programs because for a partially deterministic program P, there is often a deterministic program P' that approximates it, that is P' $\sqsubseteq$ P. This means that programs can be written and debugged using the deterministic approximation and we can avoid the problems associated with the unpredictability of non-deterministic programs.

Approximate reasoning can only demonstrate the correctness of programs and it does not help in understanding their behaviour. Understanding the behaviour of our functional search programs is difficult for a number of reasons. First, understanding search programs that prune parts of the search space is difficult because the behaviour depends on what parts of the search space are pruned and when they are pruned. Understanding the behaviour of lazy functional programs, including the time and space requirements, is difficult because the order in which expressions are evaluated is not readily apparent from the program's source code; rather it depends on when the results of an expression are needed. The use of speculative or partially mandatory parallelism complicates this by allowing some expressions to be evaluated before they are demanded.

## 6.4 Future Research

### 6.4.1 Application to Other Areas

We have used the combination of improving intervals, speculative parallelism, and partial determinism to write functional search programs. While search programs are applicable to a wide range of problems, it is interesting to consider whether or not the above can be applied to other areas.

One possible area is numerical analysis where many programs compute a sequence of better and better approximations to some value. If a program computes the sequence $a_1, a_2, \ldots$ that approximates a value $a$ then typically the sequence is improving in that the errors, $|a_1 - a|, |a_2 - a|, \ldots$, are monotonically decreasing. Such a sequence of approximations can be explicitly represented by a lazy list [30] and it is easy to define functions that add/subtract/multiply/etc. such lists.

As with the minimum and maximum functions on improving intervals, there are different strategies for evaluating the arguments of these functions. For example, a function `df_add` might evaluate its first argument, to within some tolerance, before evaluating the second argument while a function `bf_add` might evaluate the argument with the largest current error. Speculative parallelism can be used to evaluate the arguments in parallel and partial determinism is useful to use results as soon as they become available.

The approach seems particularly suited to an algorithm such as adaptive quadrature that computes an estimate for a definite integral $\int_r^l f(x)dx$. It operates by computing an initial approximation to the integral using the trapezoid rule and recursively divides the interval into sub-intervals if the initial approximation is not good enough. Hence, the overall structure of an adaptive quadrature program is similar to the structure of a search program.

### 6.4.2 Approximate Reasoning with Non-Deterministic Programs

We have shown that reasoning with a partially deterministic program can be done using a deterministic program that approximates it. This technique is applicable to any non-deterministic program that is approximated by a deterministic program, not just to partially deterministic programs. It would be interesting to look for non-deterministic programs that are not partially deterministic but have a deterministic approximation.

### 6.4.3 Other Partially Deterministic Functions and Primitives

We have described a few partially deterministic functions that are useful in writing search programs. Section 6.4.1 above suggests that there are other useful partially deterministic functions. Ideally, there would exist some set of primitive partially deterministic functions that could be used to build other partially deterministic functions. We have not found any set of primitive partially deterministic functions and it seems like a difficult task. In theory, a function like `pd_cor` or `pd_if` might be sufficient to express the possible results of program. However, in practise, the method for choosing between the results may vary. We have described examples where the choice is made based on the number of processors available or is made based on the amount of memory available.

### 6.4.4 Scheduling with Speculative Tasks

We have used a simple priority scheme for directing the scheduling of speculative tasks towards tasks that are more likely to be needed. However, scheduling with speculative tasks involves the probability that a task will be needed, the utility of the result of the task, and the amount of work required to compute the result. For example, we may schedule a task that is likely to be irrelevant if the utility of its result is high enough or if it requires very little work. There is a large body of research on scheduling tasks for execution on parallel machines but this work mostly ignores the possibility of speculative tasks.

### 6.4.5 Performance

We have not addressed the question of how our functional search programs perform in practice. There are a number of interesting possibilities for work in this direction.

1. How does the performance of the functional programs compare with imperative programs, both in the sequential and parallel case?
2. How much does speculative parallelism speedup the execution of the programs?
3. How much does the performance improve when partial determinism is used in contrast with just speculative parallelism?

Measuring the performance of search programs is difficult because the performance is often very dependent on the particular problem and data sets used. Parallel search programs have additional unpredictability due to non-determinism.

Comparisons between functional and imperative implementations depend on the particular language and compiler used. In the sequential case, good compilers for functional languages are starting to appear though the state of the art is changing rapidly and there is a risk of obsolescence. In the parallel case, there are no current compilers that support speculative parallelism well enough to make a valid comparison.

There is some question as to whether functional languages are an appropriate choice for implementing search programs. Search programs are computationally intensive and can consume large amounts of space and time (even for small problems). In some cases, such as integer programming. there are hand-crafted and highly optimized imperative programs available. A functional program can not compete with the performance of these programs. Functional programming may still be useful to prototype and experiment with new search programs.

## 6.5 Final Remarks

Our attempt to apply functional programming to search programs has highlighted both strengths and weaknesses of functional programming. Our functional search programs are simple, easy to verify and can be easily executed in parallel. However, it can be difficult to understand their behaviour and some of the programs require non-deterministic behaviour.

The main contributions of the work are:

1. The definition of improving intervals and their use in simplifying functional search programs.

2. The concept of partial determinism and its use in programs that can take advantage of parallel evaluation without requiring fair evaluation.

3. The utility of approximate reasoning with search programs and partially deterministic programs.

Our approach also suggests some interesting search strategies for branch-and-bound programs. There is a lot of research that remains to be done. The improving intervals approach needs to be tested on real problems. Futher work on partial determinism is needed to discover its applicability to other areas.

# Appendix A

# Transforming the Implementation

This section describes some simple improvements that can make the implementation of improving intervals run more quickly. Our approach is to use equational reasoning to transform the implementation from chapter 3 to a more efficient version.

The first optimization is to remove duplicate bounds from the list of bounds. The following definition of the function ii_rmdup removes duplicate bounds, and ensures that once an exact bound is found, the list is turned into an infinite list (that uses a constant amount of space).

```
> ii_rmdup (b:bs)
>     = b:rmdup' b bs, if b_nonExact b
>     = cycle [b]      , otherwise
>       where
>         rmdup' b1 (b2:bs)
>             = rmdup' b1 bs,       if b1 = b2
>             = b2:rmdup' b2 bs,    if b_nonExact b2
>             = cycle [b2],         if b_isExact b2
```

Clearly, ii_rmdup x is equivalent to x and evaluating ii_rmdup x takes $O(n)$ time if x is a list with $n$ non-exact bounds.

The function ii_lb may be optimized by unfolding its definition:

```
ii_lb a x
  =  ii_max (ii_exact a)(b_bot:x)
  =  zipwithord ub_red b_max (ii_exact a)(b_bot:x)
  =  zipwithord ub_red b_max (cycle [(V a,V a)])(b_bot:x)
  =  b_max (V a,V a) b_bot:zipwithord ub_red b_max (cycle [(V a,V a)]) x
  =  (V a,Inf):zipwithord ub_red b_max (cycle [(V a,V a)]) x
```

Let

```
ii_lb' a x = zipwithord ub_red b_max (cycle [(V a,V a)])(x)
```

Since `zipwithord` is strict in its list argument, `x` must be evaluated and the pattern `b:x` can be used to define `ii_lb'`.

```
ii_lb' a (b:x)
  =  zipwithord ub_red b_max (cycle [(V a,V a)])(b:x)
  =  b_max (V a,V a) b:zipwithord ub_red b_max (cycle [(V a,V a)])(x)
```

There are three cases for `b_max (V a,V a) b`:

1. If `b_max (V a,V a) b` = `(V a,V a)` then the subsequent bounds in `x` can be ignored. However, this case does not require special handling because the function `ii_rmdup` ensures that the result is turned into an infinite list once an exact bound is found.

2. If `b_max (V a,V a) b` = `b` then `(lb b)` $\geq$ `a`. Therefore, if `b'` is a bound tighter than `b` then `b_max (V a,V a) b'` = `b'`. So,

   ```
   ii_lb' a (b:x) = b:x, if (b_max (V a,V a) b) = b
   ```

3. If `b_max (V a,V a) b` $\neq$ `b` then

   ```
   ii_lb' a (b:x) = b_max (V a,V a) b:ii_lb' a x
   ```

Finally, unfolding `b_max`, gives

```
b_max (V a,V a) b = (v_max (V a) (lb b),v_max (V a) (ub b))
```

Combining all of the above results in the following definition for `ii_lb`.

```
> ii_lb a x = seq (force a) (ii_rmdup ((V a,Inf):ii_lb' a x))
> ii_lb' a (b:x)
>     = (b:x),              if max_b = b
>     = max_b:(ii_lb' a x), otherwise
>       where max_b = (v_max' a (lb b), v_max' a (ub b))
>             v_max' a Neginf = (V a)
>             v_max' a (V b) = (V (max2 a b))
>             v_max' a Inf = Inf
```

The other functions can be optimized in a similar manner.

# Appendix B

# Performance Testing

This appendix describes some very preliminary results on the performance of search programs that use improving intervals. These tests were not meant as serious experiments but as a guide in assessing the potential of the improving intervals approach. Several branch-and-bound programs (using Haskell, Eiffel, and C) were written to solve the 0/1 Knapsack problem (as defined in section 1.1).

Problem instances with $n = 100$ objects were generated with profits and weights uniformly random in $[1, 1000]$ and with the capacity set to half the sum of the weights. The branching function and the bounding functions follow the fixed-size tuple approach of Horowitz and Sahni [26]. In this approach, the objects are initially sorted in non-increasing order of their profit-to-weight ratios. A node at depth $k$ in the search tree represents the subproblem that packs the $k^{th}$ through $n^{th}$ objects. A node has, at most, two children that correspond to including or excluding the $k^{th}$ object in the knapsack. The lower bound is computed by including all objects, from the remaining objects, that will fit in the knapsack. The upper bound is computed by greedily packing the remaining objects and including a fraction of the first object that does not fit.

We ran tests using several different sequential implementations:

1. A Haskell program that corresponds to the Miranda implementation in Chapter 3 as well as a Haskell program that includes the optimizations in Appendix A (plus a few additional ones). We tested the Haskell program with both the depth-first strategy and the breadth-first strategy.

2. A generic best-first branch-and-bound program, written in Eiffel, and instantiated for

the 0/1 knapsack problem.

3. A specialized depth-first branch-and-bound program, written in C, for solving the 0/1 knapsack problem (converted from a program in [59]).

The times for the Haskell programs were approximately the same for the depth-first strategy and for the best-first strategy because of the nature of the problem. The original Haskell program was a factor of 30 times slower than the C program while the Eiffel program was a factor of 15 times slower than the C program. However, the more optimized Haskell program was a factor of 7 times slower than the C program or approximately twice as fast as the Eiffel version.

These results are very preliminary: none of the programs has been optimized using profiling information and only very limited datasets were used. While the performance of the Haskell program is still poor with respect to the C program, the results are promising. The Haskell program uses a generic branch-and-bound program while the C program is specialized for the 0/1 knapsack problem. There are also likely to be other optimizations that could be applied to the Haskell version.

# Bibliography

[1] S. Akl, D. Barnard, and R. Doran. Design, analysis and implementation of a parallel tree search algorithm. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-4(2):192–203, 1982.

[2] K.R. Apt and E.R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1992.

[3] Henry G. Baker and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Notices*, 12(8):55–59, 1977.

[4] G. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. PhD thesis, Carnegie-Mellon University, Computer Science Dept., 1978.

[5] G. M. Baudet. On the branching factor of alpha-beta pruning. *Artificial Intelligence*, 10(2):173–199, 1978.

[6] R. S. Bird and John Hughes. The alpha-beta algorithm: An exercise in program transformation. *Information Processing Letters*, 24:53–57, 1987.

[7] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[8] Manfred Broy. A theory for nondeterminism, parallelism, communication, and concurrency. *Theoretical Computer Science*, 45(1):1–61, 1986.

[9] F. Warren Burton. Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Transactions on Programming Languages and Systems*, 6(2):159–174, 1984.

[10] F. Warren Burton. Controlling speculative computation in a parallel functional programming language. In *Proceedings of The Fifth International Conference on Distributed Computing Systems*, pages 453–458, Denver, Colorado, 1985.

[11] F. Warren Burton. Speculative computation, parallelism, and functional programming. *IEEE Transactions on Computers*, C-34(12):1190–1193, 1985.

[12] F. Warren Burton. Functional programming for concurrent and distributed computing. *The Computer Journal*, 30(5):437–450, 1987.

[13] F. Warren Burton. Nondeterminism with referential transparency in functional programming languages. *The Computer Journal*, 31(3):243–247, 1988.

[14] F. Warren Burton. Encapsulating nondeterminacy in an abstract data type with deterministic semantics. *Journal of Functional Programming*, 1(1):3–20, 1991.

[15] Magnus Carlsson and Thomas Hallgren. Fudgets: A graphical user interface in a lazy functional language. In *Conference on Functional Programming and Computer Architecture*, 1993.

[16] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Pub. Co., 1988.

[17] E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[18] Andrew Dwelly. Functions and dynamic user interfaces. In *Functional Programming Languages and Computer Architecture*, pages 371–381, 1989.

[19] Carl Ebeling. *All the Right Moves.* MIT Press, 1987.

[20] P. Emrath and D. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, 1988.

[21] J. Feo, D. Cann, and R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10:349–365, December 1990.

[22] Ananth Y. Grama and Vipin Kumar. Parallel processing of discrete optimization problems. FTP from ftp.cs.umn.edu in directory /users/kumar/survey_discrete_opt.ps, October 1992.

[23] D. H. Grit and R. L. Page. Deleting irrelevant tasks in an expression-oriented multiprocessor system. *ACM Transactions of Programming Languages and Systems*, 3(1):49–59, 1981.

[24] Peter Henderson. Purely functional operating systems. In John Darlington Peter Henderson David A. Turner, editor, *Functional Programming and its Applications*, pages 177–192. Cambridge University Press, 1982.

[25] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[26] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.

[27] Paul Hudak. Para-functional programming. *Computer*, 18(8):60–71, 1986.

[28] Paul Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21(3):359–411, September 1989.

[29] Paul Hudak, Simon Peyton Jones, Philip Wadler, and et. al. Report on the programming language Haskell: A non-strict purely functional language. *ACM SIGPLAN Notices*, 27(5), 1992.

[30] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[31] John Hughes and John O'Donnell. Expressing and reasoning about non-deterministic functional programs. In *IV Higher Order Workshop*, Banff, 1990. Springer-Verlag.

[32] W. Ken Jackson and F. Warren Burton. Improving intervals. *Journal of Functional Programming*, 3(2), 1993.

[33] S.B. Jones and A.F. Sinclair. Functional programming and operating systems. *The Computer Journal*, 32(2):162–174, 1989.

[34] L.V. Kalé and Vikram A. Saletore. Parallel state-space search for a first solution with consistent linear speedups. *International Journal of Parallel Programming*, 19(4):251–293, 1990.

[35] V. Kumar and L.N. Kanal. A general branch and bound formulation for understanding and synthesizing and/or tree search procedures. *Aritificial Intelligence*, 21(1):179–198, 1983.

[36] Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 1988 National Conference on Artificial Intelligence*. Morgan Kaufmann, 1988.

[37] Vipin Kumar and V. Nageshwara Rao. Scalable parallel formulations of depth-first search. In Vipin Kumar, P.S. Gopalakrishnan, and Laveen N. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 42–65. Springer-Verlag, New York, 1989.

[38] Ten-Hwang Lai and Sartaj Sahni. Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27(6):594–602, 1984.

[39] E.L. Lawler and D. Woods. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[40] Guo-Jie Li and Benjamin W. Wah. Computational efficiency of parallel combinatorial or-tree searches. *IEEE Transactions on Software Engineering*, 16(1):13–31, 1990.

[41] T.A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14(4):533–551, 1982.

[42] Betrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[43] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[44] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.

[45] Randy B. Osborne. *Speculative Computation in Multilisp*. PhD thesis, MIT, 1989.

[46] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[47] S. L. Peyton Jones. Parallel implementation of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[48] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[49] G.D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5:452–487, 1976.

[50] Curt Powley, Chris Ferguson, and Richard E. Korf. Parallel heuristic search: Two approaches. In Vipin Kumar, P.S. Gopalakrishnan, and Laveen N. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 42–65. Springer-Verlag, New York, 1989.

[51] Catherine Roucairol. Parallel branch and bound algorithms - an overview. In Michel Cosnard, Yves Robert, Patrice Quinton, and Michel Raynal, editors, *Parallel and Distributed Algorithms*. North-Holland, 1989. Proceedings of the International Workshop on Parallel and Distributed Algorithms.

[52] Jonathan Schaeffer. Distributed game-tree searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.

[53] David A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn and Bacon, Inc., Boston, 1986.

[54] Ehud Shapiro. The family of concurrent logic programming languages. *Computing Surveys*, 21(3):412–510, September 1989.

[55] Wei Shu and L.V. Kalé. A dynamic scheduling strategy for the chare-kernel system. In *Proceedings of Supercomputing 89*, pages 389–398, 1989.

[56] D.R. Smith. Random trees and the analysis of branch-and-bound procedures. *Journal of the ACM*, 31(1):163–188, 1984.

[57] Richard Mark Soley. *On the Efficient Exploitation of Speculation under Dataflow Paradigms of Control*. PhD thesis, MIT, 1989.

[58] H. Søndergaard and P.Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

[59] Igor Steinberg and Marvin Solomon. Searching game trees in parallel. In *1990 International Conference on Parallel Processing*, pages III-9—17, 1990.

[60] S.M. Syslo, N. Deo, and J.S. Kowalik. *Discrete Optimization Algorithms : with Pascal Programs*. Prentice-Hall, 1983.

[61] A.P. Tolmach and A.W. Appel. Debuggable concurrency extensions for standard ML. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 120–131, 1991.

[62] David A. Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, Dec. 1986.

[63] Ellen E. Witte, Roger D. Chamberline, and Mark A. Franklin. Parallel simulated annealing using speculative computation. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):483–493, 1991.