PhD Thesis

Calculational Approach to Automatic Algorithm Construction

(運算手法による
アルゴリズムの自動構成に関する研究)

森畑 明昌
(Akimasa MORIHATA)

Department of Mathematical Informatics
University of Tokyo

Acknowledgement

I owe my thesis to kind support by a lot of people. I would like to acknowledge them.

First of all, I would like to express my deepest gratitude to Prof. Masato Takeichi, my supervisor. He consistently taught me an attitude towards science through his insightful advices, which guided me to the right way. This thesis would not have been possible without him.

I am much thankful to Prof. Zhenjiang Hu and Dr. Kiminori Matsuzaki, who constantly spared their time for discussing with me and proofreading my papers. Prof. Hu eagerly gave me lessons in functional programming and made a lot of constructive suggestions; especially, I considered optimal path problems on his advice. Dr. Matsuzaki kindly initiated me into a sense of parallel programming; moreover, implementations and experiments in this thesis partly relied on his help.

I am grateful to Mr. Kento Emoto and Mr. Kazutaka Matsuda, my brothers in our laboratory. We spent every day having discussions together, suffering from research together, encouraging each other, and enjoying ourselves. Although their contributions did not appear explicitly in this thesis, their insights are underlying throughout it.

Except for those I have mentioned, several people made technical contribution to this thesis. Especially, I am grateful to Mr. Kazutaka Morita for his interesting inversion-based automatic parallelization system, Dr. Shin-Cheng Mu for his lucid explanation of relationship between my derivation of dynamic programming algorithms and relational calculus by Bird et al., Miss Nanao Kita for her careful proofreading of my early draft about optimal path querying. I am also grateful to Dr. Makoto Hamana for discussions with him about functional treatment of graphs.

Last but not least, I am very thankful for all people who were with me, such as my friends, my teachers, my pupils, and my acquaintances. I would not continue researching without the happy lives with them.

My work was supported by the Grant-in-Aid for JSPS research fellows $20 \cdot 2411$.

Abstract

Today, computers are very popular and familiar to us. Everything is dealt with computers, and everyone uses his or her own computer every day. Popularization of computers calls for methodologies of systematic development of efficient algorithms. Although efficient algorithms are necessary for effective use of computers, it is difficult for nonspecialist to construct them.

Program calculation is a methodology to develop efficient algorithms systematically. In program calculation, we develop an efficient program, which represents an efficient algorithm, in two steps. We first construct a naive but apparently correct program without taking care of its efficiency, and then, we improve its efficiency by applying *calculational laws*, which are mathematically correct program transformation rules. One strength of program calculation is that the derived program is proved to be correct by its construction. Since each step of derivation is an application of a calculational law proved to be correct, the derivation constitutes a proof of the correctness of the derived program. Another strength is that program calculation is potentially suitable for automatic implementation. Calculational laws are program transformation rules, and we can automate calculations by implementing program transformation systems.

Although program calculation is a good methodology for algorithm construction, automatic algorithm development on calculations has not been sufficiently carried through. The most difficult part is the steps in which we should reveal and verify problem-dependent properties that are hardly seen from its specification.

In this thesis, we report our attempt to develop efficient algorithms automatically. Our idea is to provide "protocols" between us and program transformation systems. The role of the protocol is to supply the systems with the clue to efficient algorithms. For this purpose, we develop calculational laws to present the clue to efficient algorithms to the systems. Then, we design a programming language to utilize the calculational laws. The language is designed so that it clarifies the clue to efficient algorithms. Then, the system will recognize the clue and derive an efficient algorithm automatically.

We struggle for two kinds of problems in this thesis: combinatorial optimization problems and parallelization problems. For both problems, we develop calculational laws to derive efficient algorithms, design programming languages for automatic implementation of the derivation, and propose systems for obtaining efficient programs automatically.

Combinatorial optimization problems are problems to find the optimal one from a given set of candidates. Since combinatorial optimization problems have a great many applications, they are recognized to be one of the most important classes of problems in algorithm construction.

To develop efficient algorithms for combinatorial optimization problems systematically, we carefully examine structures of problems, such as structures of enumerating candidates, structures of orders to optimize, and structures of constraint that solutions should satisfy; then, we propose calculational laws to derive dynamic programming algorithms from the structures. We demonstrate the effectiveness of our calculational laws through derivations of algorithms for constrained shortest path problems.

We also propose a generic framework for optimal path querying. An optimal path query is a query to find the optimal path in a graph, where the criterion of optimality is specified by users. Our optimal path querying system builds on a domain-specific language to describe criteria of optimality, and from the description, it generates a program for efficient optimal path querying. By the virtue of a careful design of the language, the language is expressive enough to describe many practical problems; moreover, the generated programs are efficient in the sense that they are a generalization of known efficient algorithms. We also explain our implementation of the system and report some experiments.

As the latter part, we discuss derivation of efficient divide-and-conquer parallel algorithms. Developing parallel programs is much harder than developing sequential programs, and thus, automatic parallelization methods that generate parallel programs from sequential ones are called for.

We develop a calculational framework for deriving efficient divide-and-conquer parallel programs. We focus on the third list-homomorphism theorem, which states that if a list-iterating function can be defined in two certain forms, there exists a divide-and-conquer parallel algorithm to evaluate the function. We first confirm effectiveness of the theorem on lists, and after that, we generalize the theorem so as to deal with tree-iterating functions. The key to the generalization is to recognize tree-iterating functions as list-iterating functions so as to utilize theories on lists.

We also develop systems for automatic parallelization based on the third list-homomorphism theorem. We first prepare a programming language to describe sequential programs that are objects of automatic parallelization. The language is designed so as to utilize the third list-homomorphism theorem and automatic theorem proving techniques. Based on the language, we propose two automatic parallelization systems: one is based on automatic inversion, and the other is based on generation-and-testing. We report experiments with the systems and discuss further improvements.

Contents

1 Introduction **1**
- 1.1 Background . . . 2
- 1.2 Transformational Program Development . . . 2
- 1.3 Program Calculation . . . 5
- 1.4 Automatic Algorithm Construction based on Languages . . . 8
- 1.5 Contributions and Organization of the Thesis . . . 10
- 1.6 Related Works . . . 12

2 Basis of Program Calculation **15**
- 2.1 Basic Definitions . . . 15
- 2.2 Operators and Laws for Manipulating Relations and Functions . . . 17
- 2.3 Functors and Relators . . . 20
- 2.4 Recursion Schemes and Inductions . . . 23
  - 2.4.1 Automata . . . 23
  - 2.4.2 Catamorphisms . . . 24
  - 2.4.3 Reflexive Transitive Closures . . . 27
  - 2.4.4 Incrementality . . . 29

3 Calculational Laws for Combinatorial Optimization Problems **31**
- 3.1 Calculational Formalization of Combinatorial Optimization Problems 32
  - 3.1.1 Orders . . . 32
  - 3.1.2 Minimums and Minimals . . . 33
  - 3.1.3 Generic Specification of Combinatorial Optimization Problems 34
- 3.2 Greedy Theorems . . . 36
  - 3.2.1 Monotonicity and Greedy Theorems . . . 36
  - 3.2.2 Thinning Theorems . . . 39
  - 3.2.3 Drawbacks of Greedy Theorems and Thinning Theorems . . . 40
- 3.3 Solving Maximum Marking Problems . . . 41
  - 3.3.1 Deriving Linear-Time Algorithms by Specifying Requirements for Markings . . . 41
  - 3.3.2 Drawbacks and Relationship to Monotonicity . . . 44
- 3.4 Supplemental Lemmas . . . 44

4 Compositional Approach to Monotonicity **51**
4.1 Constructing Monotonicity Conditions 52
4.1.1 Specifying Candidate Generators as Unions of Functions . . . 52
4.1.2 Constructing Monotonic Orders 55
4.1.3 Deriving Dynamic Programming Algorithms 60
4.2 Deriving Algorithms for Shortest Path Problems and Their Variants . 62
4.2.1 Shortest Path Problems 62
4.2.2 Regular-Language-Constrained Shortest Path Problems 65
4.2.3 Resource-Constrained Shortest Path Problems 67
4.3 Summary and Discussions 69
4.4 Supplemental Lemmas 70

5 A Generic Framework for Optimal Path Querying **75**
5.1 Designing a Domain-Specific Language for Optimal Path Querying . . 76
5.2 Language for Optimal Path Querying 77
5.2.1 Language Description 77
5.2.2 Writing Optimal Path Queries by the Language 79
5.3 Deriving Optimal Path Querying Algorithm 81
5.3.1 Deriving Case-Analysing Function 82
5.3.2 Optimal Path Querying Algorithm 84
5.3.3 Relationship to Product Construction 86
5.3.4 Improving Efficiency of the Optimal Path Querying Algorithm 87
5.3.5 Correspondence to Existing Algorithms 89
5.3.6 How the Querying Algorithm Works for Examples 89
5.4 Optimal Path Querying System 90
5.4.1 Implementation of Optimal Path Querying System 91
5.4.2 Sample Codes 92
5.4.3 Experiments 92
5.5 Summary and Discussions 95

6 Calculational Laws for Parallel Programming **99**
6.1 Parallel Programming on Lists 100
6.1.1 List Homomorphisms 100
6.1.2 The Third List-Homomorphism Theorem 102
6.1.3 Developing Parallel Programs for Lists with the Third List-Homomorphism Theorem 102
6.1.4 Experiments 108
6.2 Parallel Programming on Binary Trees 109
6.2.1 Parallel Tree Contraction 110
6.2.2 m-Bridges 112
6.2.3 The Third List-Homomorphism Theorem on Binary Trees . . 115
6.2.4 Developing Parallel Programs for Binary Trees with the Third List-Homomorphism Theorem 120

6.2.5 Experiments . . . 125
6.3 Parallel Programming on Non-Binary Trees . . . 127
6.3.1 Parallel Tree Contraction Algorithm for Non-Binary Trees . . 128
6.3.2 The Third List-Homomorphism Theorem on Polynomial Data Structures . . . 131
6.4 Summary and Discussions . . . 134

7 Automatic Parallelization on the Third List-Homomorphism Theorem 137
7.1 Brief Introduction of Quantifier Elimination . . . 137
7.2 Designing a Language for Automatic Parallelization . . . 139
7.3 Parallelization via Inversion . . . 140
7.3.1 Automatic Derivation of Right Inverses . . . 140
7.3.2 Strong Points and Drawbacks . . . 143
7.4 Parallelization via Candidate Generation and Associativity Testing . 144
7.4.1 Generating Candidates and Testing Associativity . . . 145
7.4.2 Implementation . . . 147
7.4.3 Experiments . . . 150
7.5 Comparison of two Automatic Parallelization Methods . . . 151
7.5.1 Comparing Efficiency . . . 151
7.5.2 Comparing Feasibility . . . 151
7.5.3 Relationship between two Methods . . . 152
7.6 Summary and Discussions . . . 153

8 Conclusion 157
8.1 Summary of the Thesis . . . 157
8.2 Future Works . . . 158

Bibliography 161

Chapter 1

Introduction

『どういふわけでうれしい？』といふ質問に対して人は容易にその理由を説明することができる。けれども『どういふ工合にうれしい？』といふ問に対しては何人もたやすくその心理を説明することは出来ない。どんな場合にも、人が自己の感情を完全に表現しようと思つたら、それは容易のわざではない。この場合には言葉は何の役にもたたない。そこには音楽と詩があるばかりである。

私はときどき不幸な狂水病者のことを考へる。あの病気にかかつた人間は非常に水を恐れるといふことだ。若しその患者自身がこの苦しい実感を傍人に向つて説明しようと試みるならば患者自身はどんな手段をとるべきであらう。恐らくはどのやうな言葉の説明を以てしても、この奇異な感情を表現することは出来ないであらう。けれども、若し彼に詩人としての才能があつたら、もちろん彼は詩を作るにちがひない。詩は人間の言葉で説明することの出来ないものまでも説明する。詩は言葉以上の言葉である。

（萩原朔太郎「月に吠える」[1]序文より抜粋）

Sakutaro Hagiwara (1886–1942), who was a Japanese poet, wrote, “We can easily explain *why* we feel, while we can hardly explain *how* we feel. Words are helpless for this purpose, and only poems and music can explain our feeling.” I like this sentence. The world is filled with misunderstanding, prejudice, deception, and so on. We are always troubled by communication gaps. He pointed out that communication gaps were inevitable; however, he also pointed out that we human were on a common ground, which is revealed by artistic works such as poems and music.

By the way, how do we explain our feeling on programming? Are poems and music helpful? I believe they aren’t, because programming consists of logical thinking that can hardly be expressed by poems and music. Then, is mathematics sufficient? I also think it isn’t, because programming consists of intuitions and passions that can hardly be read from mathematical expressions by others except for few specialists. Do you have any idea?

Well, let us consider the way to explain our sense of programming not only to our friends but also to our computers, which is the theme of this thesis.

[1] 青空文庫: `www.aozora.gr.jp`.

1.1 Background

Only a few decades ago, we were not accustomed to computers. Computers were special tools that only specialists used for their specific purposes. In this a few decades, computers have become powerful and cheap, and have been used for tasks that had been considered to be unsuitable. Now, computers are very popular and familiar to us. Everything is dealt with computers, and everyone uses his or her own computer for her/his own purpose every day.

Although the power of computers has been improved a lot, the effectiveness of computers still completely depends on the procedures that the computers perform. Good procedures enable computers to manage huge and complicated works in a moment, while bad procedures make considerably powerful computers be hulks that reply no result or wrong results even for trivial works. Good procedures are called *algorithms*, and efficient algorithms have been studied for many years.

While many algorithms have been invented for many problems, popularization of computers calls for much more algorithms. We often encounter our own problems, which require us to develop our own algorithms. However, it is difficult to develop efficient algorithms, even though most of our problems might be variants of problems for which efficient algorithms are known. For developing efficient algorithms, we need to learn a lot of existing algorithms, understand how they work, choose an appropriate one, refine it so as to deal with our specific problem, implement it, and confirm its correctness. In short, constructing efficient algorithms is too difficult for nonspecialists. Methodologies for systematic development of efficient algorithms are called for.

1.2 Transformational Program Development

Transformational program development, which was devised in 1970s [CW72, Ger75, BGW76, Lov76, Weg76, BD77], is a methodology for developing efficient programs systematically. In transformational program development, we construct an efficient program in two steps. We first construct a naive but apparently correct program without taking care of its efficiency; then, we improve efficiency of the program by applying mathematically correct program transformation rules.

To taste transformational program development, let us develop an efficient program to compute variance in a transformational manner. The variance of a set X, denoted by $V[X]$, is defined as follows,

$$V[X] \stackrel{\text{def}}{=} \frac{1}{|X|} \sum_{a \in X} (a - E[X])^2$$

where $|X|$ and $E[X]$ respectively denote the size and the average of X.

Consider the equation above as a program to compute variance. The variance of X is computed by computing $(a - E[X])^2$ for each element a in X and calculating their average. Of course this program is correct, because it is the definition

of variance; however, the program is known to be inefficient a bit. In transformational program development, we try to discover a more efficient program by using calculations, as follows.

$$
\begin{aligned}
V[X] &= \{ \text{ definition } \} \\
&\quad \frac{1}{|X|}\sum_{a\in X}(a - E[X])^2 \\
&= \{ \text{ expand the square } \} \\
&\quad \frac{1}{|X|}\sum_{a\in X}(a^2 - 2aE[X] + E[X]^2) \\
&= \{ \text{ associativity of } + \} \\
&\quad \frac{1}{|X|}(\sum_{a\in X} a^2 - \sum_{a\in X} 2aE[X] + \sum_{a\in X} E[X]^2) \\
&= \{ \text{ factorization } \} \\
&\quad \frac{1}{|X|}(\sum_{a\in X} a^2 - 2E[X]\sum_{a\in X} a + E[X]^2\sum_{a\in X} 1) \\
&= \{ \textstyle\sum_{a\in X} 1 = |X| \text{ and } \sum_{a\in X} a = |X|E[X] \} \\
&\quad \frac{1}{|X|}(\sum_{a\in X} a^2 - 2|X|E[X]^2 + |X|E[X]^2) \\
&= \{ \text{ simplification } \} \\
&\quad (\frac{1}{|X|}\sum_{a\in X} a^2) - E[X]^2
\end{aligned}
$$

We have repeatedly applied mathematical laws, which can be considered as program transformation rules, and finally derived another program to compute variance, which is more efficient than the original one. Notice that the derived program requires minus operation only once, while the original one requires $|X|$ times of minus operations; in addition, we can evaluate the former in a one-path manner, namely scanning elements of X once, while the latter is a two-path algorithm, because it is necessary to compute $E[X]$ in advance.

Transformational program development has several strengths. One is that the derived program is proved to be correct by its construction. Since each step of derivation is an application of a program transformation rule that is proved to be correct, the derivation constitutes a proof of the correctness of the derived program. Recall the example above. Although the correctness of the equation $V[X] = (\sum_{a\in X} a)/|X| - E[X]^2$ is not apparent, the development process is the proof of its correctness. In usual, when we come across an efficient procedure to solve a problem, we will suffer for giving a proof of its correctness; besides, our thought is frequently wrong. Transformational program development enables us to avoid such troubles. Another strength is that the methodology is potentially suitable for

automatic implementation. We can implement the development by implementing program transformations.

One of the drawbacks of transformational program development is the difficulty in controlling application of program transformation rules. There are many program transformation rules we may think of, while we must choose an appropriate sequence of transformation rules for improving a program. Recall the previous derivation of the efficient program for computing variance. In the first step, we expanded $\sum(a-E[X])^2$ to $\sum(a^2-2aE[X]+E[X]^2)$, which declined efficiency; thus, the inverse, namely factorizing $\sum(a^2 - 2aE[X] + E[X]^2)$ to $\sum(a - E[X])^2$, seems appropriate. However, the expansion is necessary for deriving the efficient program by factorizing statically computable parts, namely $\sum 1$ and $\sum a$. This problem is serious for automatic implementation of transformational developments.

As another example, consider map, which is a higher-order macro [Wad90][2] defined as follows.

$$\begin{array}{lcl} \mathsf{map}_f([]) & = & [] \\ \mathsf{map}_f([a] +\!\!+ x) & = & [f(a)] +\!\!+ \mathsf{map}_f(x) \end{array}$$

In the definition, $[]$ denotes an empty sequence, $[a]$ denotes a singleton sequence that consists of an element a, and $+\!\!+$ denotes the concatenation of two sequences. The macro map_f is parametrized by a function f, and expresses the iteration to apply the function f to each element in the sequence.

Now, let us derive an efficient program that computes $\mathsf{map}_f \circ \mathsf{map}_g$, where the operator $\circ$ is the function composition operator and its definition is $(f \circ g)(x) = f(g(x))$. Here, we would like to use *unfolding-folding* [BD77] methodology, which is one of the best-known methods of transformational program development. In unfolding-folding, we develop an efficient program by, roughly speaking, a sequence of an unfolding rule that replaces a function call by its body expression, and a folding rule that replaces an expression by a call of a function having the expression as its body.

Let us start calculation. By induction, consider the case of computation for the empty sequence first.

$$\begin{array}{lcl} (\mathsf{map}_f \circ \mathsf{map}_g)([]) & = & \{ \text{ unfolding } \circ \} \\ & & \mathsf{map}_f(\mathsf{map}_g([])) \\ & = & \{ \text{ unfolding } \mathsf{map}_g \} \\ & & \mathsf{map}_f([]) \\ & = & \{ \text{ unfolding } \mathsf{map}_f \} \\ & & [] \\ & = & \{ \text{ folding } \mathsf{map}_{f \circ g} \} \\ & & \mathsf{map}_{f \circ g}([]) \end{array}$$

[2]We will not consider higher-order values as a first class citizen in this thesis. Therefore, we will consider map as a higher-order macro rather than a higher-order function.

Next, consider the case of nonempty sequences.

$$
\begin{aligned}
(\mathsf{map}_f \circ \mathsf{map}_g)([a] +\!\!+\, x) &= \{ \text{ unfolding } \circ \} \\
&\quad \mathsf{map}_f(\mathsf{map}_g([a] +\!\!+\, x)) \\
&= \{ \text{ unfolding } \mathsf{map}_g \} \\
&\quad \mathsf{map}_f([g(a)] +\!\!+\, \mathsf{map}_g(x)) \\
&= \{ \text{ unfolding } \mathsf{map}_f \} \\
&\quad [f(g(a))] +\!\!+\, \mathsf{map}_f(\mathsf{map}_g(x)) \\
&= \{ \text{ folding } \circ \} \\
&\quad [(f \circ g)(a)] +\!\!+\, (\mathsf{map}_f \circ \mathsf{map}_g)(x) \\
&= \{ \text{ induction hypothesis } \} \\
&\quad [(f \circ g)(a)] +\!\!+\, \mathsf{map}_{f \circ g}(x) \\
&= \{ \text{ folding } \mathsf{map}_{f \circ g} \} \\
&\quad \mathsf{map}_{f \circ g}([a] +\!\!+\, x)
\end{aligned}
$$

After all, we obtained a program $\mathsf{map}_{f \circ g}$, which is equivalent to and more efficient than the original program $\mathsf{map}_f \circ \mathsf{map}_g$.

As similar to the previous example, the most difficult part is the control of application of rules. Especially, the last step for the case of empty sequence is strange. We performed the folding of $[]$ to $\mathsf{map}_{f \circ g}([])$, even though $\mathsf{map}_{f \circ g}$ did not appear in the specification! Such steps are called *eureka steps*, which are in fact the keys to efficient programs. Control of unfolding steps is also troublesome. It was unnecessary to unfold f and g in the calculation above, though the reason was unclear. What is evident is that we need to control unfolding steps, because repeated unfolding will cause non-terminating transformations.

1.3 Program Calculation

Program calculation [BdM96] (also called *calculational programming*) is a style for achieving transformational program development. In program calculation, we derive efficient programs by a set of program transformation rules, called *calculational laws*, each of which is organized by more primitive program transformation rules such as unfolding and folding.

As an example, consider the following theorem about map.

Theorem 1.1 (map-map fusion [Bir89])**.** For any two functions f and g, the following equation holds.

$$\mathsf{map}_f \circ \mathsf{map}_g = \mathsf{map}_{f \circ g} \qquad \square$$

By reading the equation as a rewrite rule form the left hand side to the right hand side, we can recognize Theorem 1.1 as a program transformation rule to fuse two recursive functions into one. Notice that the theorem does not supply additional power to the usual unfolding-folding methodology. As demonstrated in the previous

section, Theorem 1.1 just represents a sequence of unfolding and folding. Even so, the theorem is effective. First, the theorem is frequently applicable, because map corresponds to many functions, such as squaring all elements in a sequence, converting all numbers in a sequence into strings, and so on. Second, it is unnecessary to worry about the way to control unfolding and folding, once functions in source programs are recognized as map. In other words, Theorem 1.1 identifies a useful idiom in transformational program development. From other point of view, the theorem is a high-level program transformation rule for a high-level programming language containing map as an additional primitive construct. Such high-level abstraction by using higher-order macros is the strength of program calculation. High-level program transformation rules, namely calculational laws, reduce the difficulty to control manipulation of programs. Furthermore, high-level abstraction is suitable for algorithm development, which is the objective of this thesis, if we can formalize algorithmic idioms by calculational laws.

As the next example, let us consider derivation of divide-and-conquer algorithms, which is one of the main topics of this thesis. Consider another higher-order macro foldr defined as follows.

$$\begin{array}{lcl} \mathsf{foldr}_{\oplus,e}([\,]) & = & e \\ \mathsf{foldr}_{\oplus,e}([a] +\!\!\!+\, x) & = & a \oplus \mathsf{foldr}_{\oplus,e}(x) \end{array}$$

Given a sequence $[a_0, a_1, \ldots, a_n]$, $\mathsf{foldr}_{\oplus,e}$ collapses the sequence into a value by using a binary operator $\oplus$, namely $\mathsf{foldr}_{\oplus,e}([a_0, a_1, \ldots, a_n]) = a_0 \oplus (a_1 \oplus (\cdots (a_n \oplus e) \cdots))$.

The macro foldr iterates elements in the sequence one by one. If the operator $\oplus$ is associative, namely $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ holds, the computation of $\mathsf{foldr}_{\oplus,e}$ can be accomplished in another manner.

Theorem 1.2 ([Bir87])**.** For an associative operator $\oplus$, the following equation holds.

$$\mathsf{foldr}_{\oplus,e}(x +\!\!\!+\, y) = \mathsf{foldr}_{\oplus,e}(x) \oplus \mathsf{foldr}_{\oplus,e}(y) \qquad \square$$

The equation in Theorem 1.2 means that we can compute the value of $\mathsf{foldr}_{\oplus,e}$ for a long list $x +\!\!\!+\, y$ by computing the values of the left part ($\mathsf{foldr}_{\oplus,e}(x)$) and the right part ($\mathsf{foldr}_{\oplus,e}(y)$) independently and merging them. Therefore, Theorem 1.2 is a calculational law to derive divide-and-conquer algorithms. Divide-and-conquer is an important computation pattern, especially because it is suitable for parallel computation. Independent subproblems, $\mathsf{foldr}_{\oplus,e}(x)$ and $\mathsf{foldr}_{\oplus,e}(y)$ in this case, can be computed in parallel when plural processors are available.

Like this, calculational laws can formalize the derivation of useful algorithm patterns. Then, we can perform transformational development of efficient algorithms in ease. There have been many studies for formalizing effective calculational laws [Bir84, BdM93a, BdM93b, dM95, Cur96, Gib96, SHTO00, Cur03, MKHT06, Gib07], deriving nontrivial algorithms [Bir89, Rav99b, dMG99, dMG00, Bir01, Bir06], and automating calculations [dMS01, SdM01, Yok06].

Although program calculation is a good methodology for algorithm construction, it is yet not easy enough for nonspecialists. Derivation of efficient algorithms still requires eureka steps in general, in which we need to reveal secret problem-dependent properties that are hardly seen from the description of the program.

Consider deriving a divide-and-conquer algorithm for computing polynomial function. Given a value C and a sequence $[a_0, a_1, \ldots, a_n]$ representing coefficients, the function $poly_C([a_0, a_1, \ldots, a_n]) = a_0 + a_1 C^1 + \cdots + a_n C^n$ computes the value of polynomial. It is not difficult to specify the function $poly_C$ by foldr.

$$\begin{aligned} poly_C &\stackrel{\text{def}}{=} \mathsf{foldr}_{\oslash,0} \\ a \oslash r &= a + r \times C \end{aligned}$$

The definition above corresponds to Horner algorithm. Now we would like to confirm associativity of the operator $\oslash$ for utilizing Theorem 1.2. However, the operator $\oslash$ does not satisfy associativity, and we get stuck.

This standstill is a typical situation we face in developing efficient algorithms calculationally. In such cases, we need to work out a program satisfying certain properties, namely associativity in this case. In fact, we can derive a divide-and-conquer algorithm for computing polynomials by using some tricks.

$$\begin{aligned} poly_C &= \quad \{ \text{ definition } \} \\ &\quad\ \mathsf{foldr}_{\oslash,0} \\ &= \quad \{ \text{ introducing } \pi_1(a,b) \stackrel{\text{def}}{=} a, \text{ where let } a \ominus (r,c) \stackrel{\text{def}}{=} (a + r \times C, C \times c) \ \} \\ &\quad\ \pi_1 \circ \mathsf{foldr}_{\ominus,(0,1)} \\ &= \quad \left\{ \begin{array}{l} \text{extracting } \mathsf{map}, \text{ where let } f_C(a) \stackrel{\text{def}}{=} (a, C) \\ \text{and } (a',c') \otimes (r,c) \stackrel{\text{def}}{=} (a' + r \times c', c' \times c) \end{array} \right\} \\ &\quad\ \pi_1 \circ \mathsf{foldr}_{\otimes,(0,1)} \circ \mathsf{map}_{f_C} \end{aligned}$$

Each step is not very difficult to confirm, while both of the last two steps are eureka steps. After that, the main part, $\mathsf{foldr}_{\otimes,(0,1)}$, computes a pair of values. Then, strangely enough, the operator $\otimes$ is associative, though it is nontrivial to verify the associativity. Then, Theorem 1.2 enables us to derive a divide-and-conquer algorithm for computing polynomial function,

As seen, use of calculational laws often requires certain properties, which is generally latent. Therefore, it is necessary to verify the properties or work out programs that satisfy the properties. Besides, necessity of latent properties brings additional difficulty. The guideposts to show the way to improve efficiency is hidden behind the latent properties. For example, we happily derive divide-and-conquer algorithms if associativity is visible; however, if associativity is latent, it is out of reason to make an attempt to derive divide-and-conquer algorithms, even if it is effective in truth. Therefore, we should provide a good strategy to control applications of calculational laws, as similar to the case of the unfolding-folding methodology. In summary, it is still difficult for nonspecialists to obtain efficient algorithms based on program calculation.

1.4 Automatic Algorithm Construction based on Languages

Our goal is to develop effective and useful methods for algorithm construction. We hope that our calculations will derive a lot of efficient algorithms fully automatically. However, as seen, automatic derivation of efficient algorithms is generally truly difficult, because it is necessary to reveal latent properties. To resolve the difficulty, it should be essential for us to cooperate with program transformation systems. Then, the problem raised is the communication gap. How can we convey our insight to the systems and get the systems deriving efficient algorithms? How can the systems interpret our vague insight and carry out the derivation?

In my opinion, only *languages* help us to make our vague insight concrete and convey it to our systems. Of course, existing language would not be applicable for this purpose. It is necessary to develop languages and provide "protocols" between us and systems.

Now let us introduce our approach. Our approach consists of two steps. First, we prepare a "protocol" to program transformation systems so as to supply the systems with the clue to efficient algorithms. For this purpose, we develop calculational laws that enable us to present the clue to the systems. Next, we design a programming language to utilize the calculational laws. The language is designed so that it clarifies the clue to efficient algorithms. In the language, we may be required to write a bit bothersome program to make the clue explicit; however, necessity of such additional effort is natural, because efficient algorithms require more insight than naive algorithms. After that, the system will recognize the clue and derive an efficient algorithm automatically. Such a framework is useful even for hand-development of efficient algorithms, because finding and utilizing such clues is the wisdom for algorithm development.

To see our approach, recall the derivation of a divide-and-conquer algorithm for $poly_C$, the function computing polynomial. Although Theorem 1.2 certainly expresses the key to divide-and-conquer algorithms, namely associativity, it is not suitable for automatic implementation. The theorem provides no method for verifying or revealing associativity, and there is nothing we can do if associativity does not hold. Instead of it, we would like to make use of *the third list-homomorphism theorem*[3], which we will introduce in Chapter 6. The theorem states that a function on a sequence can be computed in a divide-and-conquer manner if and only if there exist two sequential programs that respectively scan the sequence leftward and rightward, and compute the value of the function.

Previously we have introduced the definition of $poly_C$ by using foldr, and here

[3]Note that this theorem have been called *the third homomorphism theorem* in calculational programming community. However, a theorem having the same name is known in the group theory. To avoid conflict between them, we will call it "the third *list*-homomorphism theorem" in this thesis.

we repeat it in a foldr-less style.

$$\begin{array}{lcl} poly_C([]) & = & 0 \\ poly_C([a] +\!\!+ x) & = & a + poly_C(x) \times C \end{array}$$

The program above computes the value of $poly_C([a] +\!\!+ x)$ from the value of right part of the sequence, $poly_C(x)$; thus, this is a leftward definition. The third list-homomorphism theorem indicates that a leftward definition is insufficient for deriving a divide-and-conquer algorithm. Therefore, we consider a rightward definition of $poly_C$, and think of the following one, where the function *length* computes the length of a sequence.

$$\begin{array}{lcl} poly_C([]) & = & 0 \\ poly_C(x +\!\!+ [a]) & = & poly_C(x) + a \times C^{length(x)} \end{array}$$

This program corresponds to the straightforward computation of polynomials, and computes the value of $poly_C(x +\!\!+ [a])$ in a rightward manner.What is important is that the rightward program uses an auxiliary function *length*. Then, roughly speaking, the theorem proves that the information of *length* is necessary for a divide-and-conquer algorithm of $poly_C$. Recall the following strange program that we have used for deriving a divide-and-conquer algorithm of $poly_C$.

$$\begin{array}{l} poly_C = \pi_1 \circ \mathsf{foldr}_{\otimes,(0,1)} \circ \mathsf{map}_{f_C} \\ \quad \pi_1(a,b) = a \\ \quad (a',c') \otimes (r,c) = (a' + r \times c', c' \times c) \\ \quad f_C(a) = (a, C) \end{array}$$

In fact, it is $C^{length(x)}$ that is retained in the second component of $(\mathsf{foldr}_{\otimes,(0,1)} \circ \mathsf{map}_{f_C})(x)$ and exactly corresponds to the information of *length*. In other words, the information of *length* is certainly the clue to derive a divide-and-conquer algorithm in this case. In summary, the third list-homomorphism theorem is useful for deriving divide-and-conquer algorithms, because it enables us to find the clue.

Next, we would like to automate such derivations by designing a language for automatic derivation of divide-and-conquer algorithms. The third list-homomorphism theorem indicates that it is effective to write two programs, namely leftward and rightward programs, because it enables us to reveal information necessary for deriving divide-and-conquer algorithms. In addition, it would be useful to design a language that ease the difficulty to verify associativity of the derived operator. Based on these observations, we will propose a language for automatic parallelization together with automatic parallelization algorithms in Chapter 7.

As explained, we will design languages so as to utilize calculational laws to derive efficient algorithms. Note that the languages will be *domain-specific languages* rather than general-purpose languages, on one hand, because it is difficult to derive efficient algorithms for general problems. Instead, our languages provide characterizations of classes in which deriving efficient algorithms is relatively easy. On the other hand,

our languages should have the ability to solve a large class of practical problems, otherwise the languages are useless. In summary, most important requirements for our languages are the following two.

- *Effectiveness*: the language enables us to generate efficient programs for solving problems described; in other words, the language raises sufficient information for utilizing powerful calculational laws.
- *Expressiveness*: we can describe a large class of practical problems by the language; in addition, it is desirable that writing programs by the language or compiling programs into the language is easy.

1.5 Contributions and Organization of the Thesis

We struggle for two kinds of problems in this thesis: combinatorial optimization problems and parallelization problems. Thus, this thesis consists of two main parts. After Chapter 2, where we prepare basic definitions, we discuss systematic derivation of efficient algorithms for combinatorial optimization problems. Next, as the second part, we consider systematic derivation of divide-and-conquer parallel algorithms, and lastly, we conclude this thesis and discuss directions of further research in Chapter 8. In both parts, we develop calculational laws to derive efficient algorithms, design programming languages for automatic implementation of the derivation, and construct systems for deriving efficient programs automatically.

Let us overview the two main parts.

Combinatorial optimization problems are the theme of the first part, which consists of Chapters 3, 4, and 5. Combinatorial optimization problems are problems to find the optimal solution among those satisfy certain requirements. Since combinatorial optimization problems have a great many applications, they are recognized to be one of the most important classes of problems in algorithm construction.

In Chapter 3, we review existing calculational studies about combinatorial optimization problems. On one hand, Bird, de Moor, and Curtis [BdM93b, BdM93a, dM95, BdM96, Cur96, Cur03] studied calculational characterization of efficient algorithms for combinatorial optimization problems, and their studies are summarized as some calculational laws, called *greedy theorems*. While greedy theorems provide a generic characterization for efficient algorithms, it is not suitable for automatic implementation. The theorems require orders satisfying certain properties, and such orders are difficult to find in general. On the other hand, it was proved that a class of combinatorial optimization problems, called *maximum marking problems*, is efficiently solvable once the problem is specified in certain forms [ALS91, BPT92, SHTO00]. However, the results about maximum marking problems are not sufficiently generic. They cannot deal with a lot of important problems such as problems concerning graphs.

In Chapter 4, we develop calculational laws that are both generic and implementable. We focus on structures of problems, such as structures of enumerating

candidates, structures of orders to optimize, and structures of constraint that solutions should satisfy; then, we can develop efficient algorithms in ease on the structures. As a summary, we propose calculational laws to derive dynamic programming algorithms. The laws correspond to a generalization of the known results about maximum marking problems. We demonstrate the effectiveness of our calculational laws through derivations of algorithms for several problems including shortest path problems and their variants.

In Chapter 5, we concentrate on the optimal path querying problem. An optimal path query is a query to find the optimal path in a graph, where the criterion of optimality is specified by users. We propose a system for optimal path querying based on the result shown in Chapter 4. The system builds on a domain-specific language to describe optimal path queries, and from the criterion of optimality written in the language, it generates a program for efficient optimal path querying. By the virtue of a careful design of the language, the language is expressive enough to describe many practical problems; moreover, the generated programs are efficient in the sense that they correspond to a generalization of known efficient algorithms. We also explain our implementation of the system and report some experiments.

In the second part, which consists of Chapters 6 and 7, we discuss derivation of efficient divide-and-conquer parallel algorithms. Developing parallel programs is much more difficult than developing sequential programs, and thus, we seek for automatic parallelization methods that generate parallel programs from sequential programs. While there have been a lot of studies for systematic development of parallel programs, few studies exist for automatic parallelization of complex reductions and scanning computations. Efficient divide-and conquer parallel algorithms for reductions or scanning computations require certain properties on operations that merge the results of subproblems, and the properties disturb automatic parallelization.

In Chapter 6, we develop a calculational framework for deriving efficient divide-and-conquer parallel programs. We consider that it is too difficult to obtain a parallel program from a sequential program; instead, we attempt to obtain a parallel program from two sequential programs. In fact, the third list-homomorphism theorem [Gib96], which is a fork theorem in the community of program calculation, states that if a list-iterating function can be defined in two certain forms, there exists a divide-and-conquer parallel algorithm to evaluate the function. We first confirm effectiveness of the theorem. After that, we generalize the theorem so as to deal with tree-iterating functions. The key to the generalization is to simulate computations on trees by list-iterating computation so that we can utilize theories on lists.

In Chapter 7, we develop systems for automatic parallelization based on the third list-homomorphism theorem. We first prepare a programming language to describe sequential programs that are objects of automatic parallelization. The language is designed so as to utilize the third list-homomorphism theorem and automatic theorem proving technique. Based on the language, we propose two automatic parallelization systems: one is based on automatic inversion, and the other is based

on generation-and-testing. We report experiments with the systems and discuss further improvements.

1.6 Related Works

We will study automatic algorithm construction based on program calculation. Here, we would like to discuss relationships to the existing works concerning systematic algorithm construction, and later we will discuss those concerning each detailed topic, including works based on program calculation.

There are very many studies about transformational program development, and Pettorossi and Proietti [PP96] carried out an extensive survey of this topic.

Unfolding-folding, first formalized by Burstall and Darlington [BD77], is one of the best-known methods for transformational program development. Unfolding-folding is a generic method, and the key issue is the control of application of unfolding steps and folding steps. We need to provide a strategy to identify which function we should perform unfolding and when we should perform folding by introducing appropriate definition of a new function. There are many studies to provide a good strategy, and Sørensen et al. [SGJ94] gave a comparison of some of then. *Partial evaluation* [JGS93, Jon96] is one of the most successful approaches to give a good strategy for unfolding-folding. In partial evaluation, we consider transforming a general-purpose program to a specialized one from given static inputs. The unfolding steps are controlled so that static computations for the static inputs will be fixed into static values. A strong point of partial evaluation is that it is very suitable for automatic implementations. Many studies showed that partial evaluation techniques automatically derived efficient programs for many problems, as reported in a survey by Jones [Jon96].

Another successful study is the sequence of works by Paige et al. [Pai83, CP89, PY97], which is sometimes referred as *finite differencing*, the program transformation rule centered around. In Paige's method, we first specify the problem by a high-level language that consists of set-based fixed-point operations, derive an iterative program by transforming fixed-point operations into loops, improve efficiency of the loops by replacing costly set-based operations by incremental operations, and compile it into efficient codes by selecting appropriate data structure for implementing sets. As a complete example, see Liu and Yu [LY02] that demonstrated effectiveness of the method by deriving efficient implementation of regular path querying. As the same as partial evaluation, a strong point of Paige's method is that it is implementable; besides, and different from unfolding-folding, the method consists of only three steps and it is unnecessary to worry about the control of the transformation steps.

As a price of implementable formalization, these methods tend to derive efficient low-level implementations of generic programs, rather than improving algorithms on high-level languages. Therefore, they are not suitable for the purpose of this thesis,

while they will be effective for providing efficient implementations for programs derived by our methods.

From the algorithmic viewpoint, our study can be seen as a variant of those providing generic algorithms for a set of problems. In our approach, we first pick up an algorithm pattern and formalize a calculational law that shows a way to solve a set of problems by the algorithm pattern. In other words, the calculational law forms a generic algorithm for a set of problems. There are quite many good studies for giving generic algorithms, and especially, the studies about matroids and their variants [Whi35, Edm71, Fra81, KL81, HMS93] provide generic ways to solve a large set of combinatorial optimization problems efficiently. One of the most distinctive parts of our study is that we intensively consider automatic reduction into the generic algorithms. Even if an algorithm is very generic and efficient, nonspecialists cannot utilize it unless they can easily reduce their problems into problems solved by the algorithm. We design languages for implementing automatic reduction into generic algorithms so as to nonspecialists can utilize our results. One of the our ideals is the linear programming [Chv83]. No knowledge about theories of linear programming is necessary for using linear programming solvers, and it is sufficient to specify problems by providing a set of inequalities and equations.

Chapter 2

Basis of Program Calculation

In this chapter, we introduce basic definitions and notions used in this thesis. We borrow many basic notions from relational calculus by Bird and de Moor [BdM96] and functional programming language Haskell [Pey03].

2.1 Basic Definitions

Sets

We use a pair of curly brackets to denote a set[1]. For example, $\{3\}$ denotes a set that consists of 3, and $\{1, 4, 6\}$ denotes a set that consists of 1, 4, and 6. The empty set is denoted by $\emptyset$. A set of all subset of A is denoted by 2^A. Basic operators for sets, such as $\cup$, $\cap$, and $\times$ are defined as usual: $A \cap B \stackrel{\text{def}}{=} \{a \mid a \in A \wedge a \in B\}$, $A \cup B \stackrel{\text{def}}{=} \{a \mid a \in A \vee a \in B\}$, and $A \times B \stackrel{\text{def}}{=} \{(a, b) \mid a \in A \wedge b \in B\}$. The size of a set S is denoted by $|S|$, or just by S if it is not ambiguous. The function $\{\cdot\}$ takes an element and returns the singleton set containing the element.

The set of Boolean values is denoted by *Bool*, namely $\mathit{Bool} \stackrel{\text{def}}{=} \{\mathit{True}, \mathit{False}\}$. The set of all natural numbers, the set of all integers, the set of all non-negative integers, and the set of all real numbers are respectively denoted by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Z}_+$, and $\mathbb{R}$. The set that contains only one element $()$ is denoted by $\mathbf{1}$.

Relations and Functions

Relation is a set of pairs. For sets A and B, R is said to be a relation between A and B if $R \subseteq A \times B$; if so, we write $R : B \leftrightarrow A$. We may write $x\,R\,y$ instead of $(x, y) \in R$. The identity relation on a set A is denoted by $id_A : A \leftrightarrow A$, i.e., $id_A \stackrel{\text{def}}{=} \{(a, a) \mid a \in A\}$. The subscript may be omitted if it is apparent from its

[1] We only consider small sets, namely sets in a specified universe, to avoid dealing with sets of sets. For the same reason, we do not consider higher-order functions or higher-order relations in this thesis.

context. The converse of a relation $R : A \leftrightarrow B$ is denoted by $R^\circ : B \leftrightarrow A$, which is defined as follows.

$$R^\circ \stackrel{\text{def}}{=} \{(b, a) \mid (a, b) \in R\}$$

A relation $f : B \leftrightarrow A$ is said to be a function if it is simple, that is, the following property holds.

$$((a, b_1) \in f \wedge (a, b_2) \in f) \Rightarrow b_1 = b_2$$

The property is equivalent to the following point-free style inequality.

$$f \circ f^\circ \subseteq id$$

In such case, we write $f : B \rightarrow A$ to explicitly denote that f is a function, and write $f(b) = a$ instead of $(a, b) \in f$ or $a\,f\,b$. Parentheses that stand for function application may be omitted. If a function $f : A \rightarrow B$ is total, namely $\forall a \in A : \exists b \in B : f(a) = b$ holds, then f is called a total function[2]. A function may be called a partial function if it is not total. A total function that yields a Boolean value is called a predicate. A function f is said to be injective if f° is a function. A total injective function is called bijective.

As usual in functional programming, we may define functions by a set of equations, each of which may consist of variables. The underscore _ abbreviates a variable having a distinguishable name, and means that we will not care values bound to the variable. For notational convenience, we may use equations whose left hand sides have overlapping domains, and in such cases, former equations have higher priority. For example, multiplication function on non-negative integers $mult{:}(\mathbb{Z}_+ \times \mathbb{Z}_+) \rightarrow \mathbb{Z}_+$ can be defined by the following two equations: $mult(0, _) \stackrel{\text{def}}{=} 0$ and $mult(n, m) \stackrel{\text{def}}{=} m + mult(n - 1, m)$.

Tuples

Given an natural number i, π_i denotes the projection of the ith element from a tuple, i.e., $\pi_i(a_1, \ldots, a_i, \ldots, a_k) \stackrel{\text{def}}{=} a_i$ where $k \geq i$. An operator $(- \vartriangle -)$ is used to construct pairs, and its definition is the following.

$$(R \vartriangle S) \stackrel{\text{def}}{=} \{((b, c), a) \mid (b, a) \in R \wedge (c, a) \in S\}$$

Another operator $(- \times -)$ is also related to tuples, and defined as follows.

$$(R \times S) \stackrel{\text{def}}{=} \{((c, d), (a, b)) \mid (c, a) \in R \wedge (d, b) \in S\}$$

We use binary operators such as $\uparrow$, $\oplus$, and $\oslash$ to denote functions that take a tuple as their arguments. In short, $a \oplus b = c$ is equivalent to $(\oplus)(a, b) = c$.

[2] Note that the definition of functions is different from that of Bird and de Moor [BdM96]. They define a function as a simple and total relation.

Tagged Sums

The tagged sum (which can be seen as "disjoint sum") of two sets A and B is denoted by $A + B$, whose definition is $A + B \stackrel{\text{def}}{=} (\{\mathit{True}\} \times A) \cup (\{\mathit{False}\} \times B)$. We will use L and R for shorthands of tagged elements, i.e., $\mathsf{L}(a)$ and $\mathsf{R}(b)$ respectively denote (True, a) and (False, b). As similar to the case of pairs, two operators $(- \triangledown -)$ and $(- + -)$ are related to tagged sums, and defined as follows.

$$
\begin{aligned}
(R \triangledown S) &\stackrel{\text{def}}{=} \{(a, \mathsf{L}(b)) \mid (a, b) \in R\} \cup \{(a, \mathsf{R}(b)) \mid (a, b) \in S\} \\
(R + S) &\stackrel{\text{def}}{=} \{(\mathsf{L}(a), \mathsf{L}(b)) \mid (a, b) \in R\} \cup \{(\mathsf{R}(a), \mathsf{R}(b)) \mid (a, b) \in S\}
\end{aligned}
$$

Sequences

We use a pair of brackets split by commas to denote a sequence, which is also called a list. Given a set A, A^* denotes a set of all sequences whose each element is an elements of A. The empty sequence is denoted by $[]$. The concatenation of two sequences is denoted by $+\!\!+$, i.e., $[x_0, \ldots, x_n] +\!\!+ [y_0, \ldots, y_m] \stackrel{\text{def}}{=} [x_0, \ldots, x_n, y_0, \ldots, y_m]$.

Graphs

A graph $G = (V, E)$ consists of a set of vertexes V and a set of edges E. Functions $hd : E \to V$ and $tl : E \to V$ respectively yield the startpoint and the endpoint of an edge. Each edge has weights specified by weight functions. Given an alphabet Σ, a graph $G = (V, E)$ is said to be labeled by Σ when a labeling function $l : E \to \Sigma$ is specified. For a weight function w, a labeling function l, and a sequence of edges x, $l(x)$ and $w(x)$ respectively stand for the label and the weight of x, as usual.

A sequence of edges $[a_0, a_1, \ldots, a_n] \in E^*$ is said to be a path if $tl(a_k) = hd(a_{k+1})$ holds for all $0 \leq k < n$. A function $dst : E^* \to (V \cup \mathsf{1})$ takes a path and returns its destination, and its definition is $dst([]) \stackrel{\text{def}}{=} ()$ and $dst(x +\!\!+ [a]) \stackrel{\text{def}}{=} tl(a)$.

Basic knowledge about standard graph algorithms are assumed, and refer to textbooks such as [CSRL01, KT05] if necessary.

Basic Functions

We will use several standard functions in Haskell, and their definitions are summarized in Figure 2.1. In addition to them, we use $\uparrow$ and $\downarrow$ to denote binary maximum and minimum operators respectively.

2.2 Operators and Laws for Manipulating Relations and Functions

In program calculation, functions and relations are respectively used to express deterministic and nondeterministic computations. Some operators are used to express

$$\begin{aligned}
wrap(a) &\stackrel{\text{def}}{=} [a] \\[6pt]
tails([]) &\stackrel{\text{def}}{=} [[]] \\
tails([a] +\!\!+ x) &\stackrel{\text{def}}{=} [[a] +\!\!+ x] +\!\!+ tails(x) \\[6pt]
inits([]) &\stackrel{\text{def}}{=} [[]] \\
inits(x +\!\!+ [a]) &\stackrel{\text{def}}{=} inits(x) +\!\!+ [x +\!\!+ [a]] \\[6pt]
\mathsf{const}_a(_) &\stackrel{\text{def}}{=} a \\[6pt]
\mathsf{map}_f([]) &\stackrel{\text{def}}{=} [] \\
\mathsf{map}_f([a] +\!\!+ x) &\stackrel{\text{def}}{=} [f(a)] +\!\!+ \mathsf{map}_f(x) \\[6pt]
\mathsf{foldr}_{\oplus,e}([]) &\stackrel{\text{def}}{=} e \\
\mathsf{foldr}_{\oplus,e}([a] +\!\!+ x) &\stackrel{\text{def}}{=} a \oplus \mathsf{foldr}_{\oplus,e}(x) \\[6pt]
\mathsf{foldl}_{\otimes,e}([]) &\stackrel{\text{def}}{=} e \\
\mathsf{foldl}_{\otimes,e}(x +\!\!+ [a]) &\stackrel{\text{def}}{=} \mathsf{foldl}_{\otimes,e}(x) \otimes a \\[6pt]
\mathsf{scanr}_{\oplus,e}([]) &\stackrel{\text{def}}{=} [e] \\
\mathsf{scanr}_{\oplus,e}([a] +\!\!+ x) &\stackrel{\text{def}}{=} [\mathsf{foldr}_{\oplus,e}([a] +\!\!+ x)] +\!\!+ \mathsf{scanr}_{\oplus,e}(x) \\[6pt]
\mathsf{scanl}_{\otimes,e}([]) &\stackrel{\text{def}}{=} [e] \\
\mathsf{scanl}_{\otimes,e}(x +\!\!+ [a]) &\stackrel{\text{def}}{=} \mathsf{scanl}_{\oplus,e}(x) +\!\!+ [\mathsf{foldl}_{\oplus,e}(x +\!\!+ [a])]
\end{aligned}$$

Figure 2.1. Definitions of standard functions

combinations or relationship of functions and relations.

Given a relation R, the *domain* of a relation R, denoted by $\mathrm{dom}(R)$, is defined by $\mathrm{dom}(R) \stackrel{\text{def}}{=} \{b \mid (a,b) \in R\}$. Similarly, the *range* of a relation R, denoted by $\mathrm{ran}(R)$, is defined by $\mathrm{ran}(R) \stackrel{\text{def}}{=} \{a \mid (a,b) \in R\}$.

The binary operator $\circ$ denotes the composition of relations, and it is defined as follows:

$$R \circ S \stackrel{\text{def}}{=} \{(c,a) \mid \exists b \in B :\ (c,b) \in R \land (b,a) \in S\}$$

It is worth noting that the operator $\circ$ is associative. To denote repeated compositions of the same relation, we borrow the power notation, i.e., given a relation $R : A \leftrightarrow A$, $R^0 \stackrel{\text{def}}{=} id_A$ and $R^n \stackrel{\text{def}}{=} R \circ R^{n-1}$ for $n \in \mathbb{N}$.

The operator $\overline{}$ denotes the negation of a relation, and it is defined as follows.

$$a\, \overline{R}\, b \stackrel{\text{def}}{\Longleftrightarrow} \neg(a\, R\, b)$$

The operator $\cap$ is used to express a conjunction of two relations. The following is the definition of $\cap$, which is nothing but the usual intersection operator when we

recognize relations as sets.

$$a\,(R \cap S)\,b \overset{\text{def}}{\Leftrightarrow} a\,R\,b \wedge a\,S\,b$$

The operator $\cap$ can be characterized by the following property.

$$(R \cap S) \supseteq X \Leftrightarrow (R \supseteq X) \wedge (S \supseteq X)$$

Similar to $\cap$, we define a disjunction operator $\cup$ as follows.

$$a\,(R \cup S)\,b \overset{\text{def}}{\Leftrightarrow} a\,R\,b \vee a\,S\,b$$

The operator $\cup$ is characterized by the following property.

$$(R \cup S) \subseteq X \Leftrightarrow (R \subseteq X) \wedge (S \subseteq X)$$

The operator $\Rightarrow$, corresponding to the logical implication, is defined as follows.

$$a\,(R \Rightarrow S)\,b \overset{\text{def}}{\Leftrightarrow} a\,\overline{R}\,b \vee a\,S\,b$$

Its characteristic property is the following.

$$(R \Rightarrow S) \supseteq X \Leftrightarrow S \supseteq (R \cap X)$$

Note that in our setting, the operators for relations satisfy laws that logical operators satisfy, such as the double negation elimination law and the De Morgan law.

We will use the right-division operator $/$, which introduces "for all" quantification.

$$a\,(R/S)\,b \overset{\text{def}}{\Leftrightarrow} \forall c:\ b\,S\,c \Rightarrow a\,R\,c$$

Its axiomatic definition is the following.

$$R/S \supseteq X \Leftrightarrow R \supseteq (X \circ S)$$

From the definition, $/$ operator is anti-monotonic to the right operand.

The operator Λ, called power transpose, is used to transform a nondeterministic computation into a deterministic computation that generates all possible solutions.

$$\Lambda R(a) \overset{\text{def}}{=} \{b \mid (b,a) \in R\}$$

The power transpose operator is characterized by the following equation.

$$S = \in \circ\, \Lambda S$$

Given a predicate $p : A \rightarrow \mathit{Bool}$, the filtering function raised by p is denoted by $p\triangleleft : 2^A \rightarrow 2^A$. Its definition is the following.

$$p\triangleleft(X) \overset{\text{def}}{=} \{a \mid a \in X \wedge p(a)\}$$

Similarly, the filtering relation $p? : A \leftrightarrow A$ is defined as follows.

$$a\,p?\,a \overset{\text{def}}{\Leftrightarrow} p(a)$$

The function $p\triangleleft$ and the relation $p?$ satisfies $p\triangleleft = \Lambda(p? \circ \in)$.

Later we will make intensive use of *right inverses*.

Definition 2.1 (right inverse)**.** A function $f : B \to A$ is said to be a *right inverse* of a relation $R : A \leftrightarrow B$ if $R \circ f \circ R = R$ holds. □

To show explicitly that a function is a right inverse of R, we will write the function as $R^\bullet$. Note that more than one right inverses may exist for a relation. The converse of R is not a right inverse of R and only an inequality $R^\bullet \subseteq R^\circ$ holds, because that R° may not be a function; in other words, $R^\bullet$ is a refinement of R° into a function.

For total functions, some useful properties are known. We will use the following properties [BdM96], where R and S are relations and f is a total function.

$$\Lambda(S \circ f) = \Lambda S \circ f \tag{2.2}$$
$$(f \circ R \subseteq S) \Leftrightarrow (R \subseteq f^\circ \circ S) \tag{2.3}$$
$$(R \circ f^\circ \subseteq S) \Leftrightarrow (R \subseteq S \circ f) \tag{2.4}$$

2.3 Functors and Relators

We borrow some notions from the category theory.

A category consists of objects and morphisms. In this thesis, we consider only two kinds of categories. One is the category of *Set*, where objects are sets and morphisms from an object A to an object B are total functions from a set A to a set B. The other is the category of *Rel*, where objects are sets and morphisms from an object A to an object B are relations between a set A and a set B.

A functor is a morphism of categories. For two categories $\mathcal{A}$ and $\mathcal{B}$, a functor $\mathsf{F} : \mathcal{A} \to \mathcal{B}$ maps each object $A \in \mathcal{A}$ to $\mathsf{F}A \in \mathcal{B}$ and each morphism $f \in \mathcal{A}$ to $\mathsf{F}f \in \mathcal{B}$, with satisfying the following properties.

$$\mathsf{F}(id_{\mathcal{A}}) = id_{\mathcal{B}}$$
$$\mathsf{F}(f \circ g) = \mathsf{F}f \circ \mathsf{F}g$$

An example of functors is the power-set functor P whose definition is $\mathsf{P}(A) \stackrel{\text{def}}{=} 2^A$ and $\mathsf{P}f(X) \stackrel{\text{def}}{=} \{f(a) \mid a \in X\}$.

A functor is said to be polynomial if it is constructed by the combinations of the identity functor I, the constant functor $!_B$ where B is a parameter, the product bifunctor $\times$, and the coproduct bifunctor $+$. The definition is the following, in which F and G denote functors, A and B denote objects, and f denotes a morphism of appropriate type.

$$\begin{array}{ll}
\mathsf{I}A \stackrel{\text{def}}{=} A & \mathsf{I}f \stackrel{\text{def}}{=} f \\
!_B A \stackrel{\text{def}}{=} B & !_B f \stackrel{\text{def}}{=} id_B \\
(\mathsf{F} \times \mathsf{G})A \stackrel{\text{def}}{=} (\mathsf{F}A \times \mathsf{G}A) & (\mathsf{F} \times \mathsf{G})f \stackrel{\text{def}}{=} (\mathsf{F}f \times Gf) \\
(\mathsf{F} + \mathsf{G})A \stackrel{\text{def}}{=} \mathsf{F}A + \mathsf{G}A & (\mathsf{F} + \mathsf{G})f \stackrel{\text{def}}{=} (\mathsf{F}f + \mathsf{G}f)
\end{array}$$

The least fixed point of a functor F, denoted by $\mu\mathsf{F}$, is the smallest set that satisfies an equation $\mathsf{F}(\mu\mathsf{F}) = \mu\mathsf{F}$. The least fixed point exists for each polynomial functor. It is well known that the least fixed points of polynomial functors provides a good characterization of a large class of algebraic data structures [MFP91, Fok92, Mei92, BJJM99], and such data structures are called *polynomial data structures*.

As an example, let us consider binary trees retaining natural numbers in their internal nodes.

$$\begin{array}{rcl} \textbf{data}\ \mathsf{Tree}_{\mathbb{N}} & = & \mathit{Leaf} \\ & | & \mathit{Node}(\mathbb{N}, \mathsf{Tree}_{\mathbb{N}}, \mathsf{Tree}_{\mathbb{N}}) \end{array}$$

To capture this structure by the least fixed point of a functor, consider the following polynomial functor T.

$$\mathsf{T} = !_1 + !_{\mathbb{N}} \times (\mathsf{I} \times \mathsf{I})$$

Interpret $\mathsf{L}()$ as a leaf, *Leaf*, and $\mathsf{R}(a, (t_1, t_2))$ where $a \in \mathbb{N}$ and $t_1, t_2 \in \mu\mathsf{T}$ as an internal node, $\mathit{Node}(a, t_1, t_2)$. Then, we can recognize the least fixed point of T as the set of all node-valued binary trees. Each tree is constructed in a bottom-up manner, namely from leaves to the root.

Next, consider the following polynomial functor S.

$$\mathsf{S} = !_1 + !_{\mathbb{Z}} \times \mathsf{I}$$

As similar to the case of node-valued binary trees, we can interpret the least fixed point of S as a set of sequences of integers, where $\mathsf{L}()$ and $\mathsf{R}(a, x)$ respectively correspond to $[]$ and $[a] +\!\!+ x$. In other words, $\mu\mathsf{S}$ corresponds to the set of sequences of integers, in which each sequence is constructed from an empty sequence by repeatedly adding an element to its left.

In the category *Rel*, not all functors are useful for calculations, and we only consider *relators*.

Definition 2.5 (relator)**.** A functor F is said to be a *relator* if it respects inclusions, that is, $R \subseteq S$ implies $\mathsf{F}R \subseteq \mathsf{F}S$ for any relations R and S.

Actually most of the useful functors in computer science are relators. For example, the powerset functor is a relator. Polynomial functor are also relators. It is known [BdM96] that relators respect converses, namely $\mathsf{F}(R^\circ) = (\mathsf{F}R)^\circ$.

Let us see relationships between relators and relation-manipulating operators.

First, consider the operator $\cap$. As the following lemmas show, relators do not satisfy distributivity over $\cap$ in general, while polynomial functors do.

Lemma 2.6. For any relator F and relations R and S, $\mathsf{F}(R \cap S) \subseteq \mathsf{F}R \cap \mathsf{F}S$ holds.

Proof.

$$\begin{array}{rl} \mathsf{F}(R \cap S) \subseteq (\mathsf{F}R \cap \mathsf{F}S) \quad \Leftrightarrow & \{\ \text{property of } \cap\ \} \\ & (\mathsf{F}(R \cap S) \subseteq \mathsf{F}R) \wedge (\mathsf{F}(R \cap S) \subseteq \mathsf{F}S) \\ \Leftarrow & \{\ \text{relator}\ \} \\ & ((R \cap S) \subseteq R) \wedge ((R \cap S) \subseteq S) \\ \Leftrightarrow & \{\ \text{trivial } (\cap)\ \} \\ & \mathit{True} \end{array}$$

□

Lemma 2.7. For any polynomial functor F and relations R and S, $\mathsf{F}(R \cap S) = \mathsf{F}R \cap \mathsf{F}S$ holds.

Proof. It is a direct consequence of Propositions 6.3.10 and 5.3.9 in [dM92]. □

Different from $\cap$, relators do not satisfy distributivity over $\cup$, even if they are polynomial.

Lemma 2.8. For any relator F and relations R and S, $\mathsf{F}R \cup \mathsf{F}S \subseteq \mathsf{F}(R \cup S)$ holds.

Proof.

$$
\begin{array}{rl}
(\mathsf{F}R \cup \mathsf{F}S) \subseteq \mathsf{F}(R \cup S) \;\Leftrightarrow & \{\text{ property of } \cup \text{ }\} \\
& (\mathsf{F}R \subseteq \mathsf{F}(R \cup S)) \wedge (\mathsf{F}S \subseteq \mathsf{F}(R \cup S)) \\
\Leftarrow & \{\text{ relator }\} \\
& (R \subseteq (R \cup S)) \wedge (S \subseteq (R \cup S)) \\
\Leftrightarrow & \{\text{ trivial } (\cup) \text{ }\} \\
& \mathit{True}
\end{array}
$$

□

Nothing interesting is known about relationship between relators and the $\Rightarrow$ operator in general. But, for polynomial functors, the following lemma holds.

Lemma 2.9. For any polynomial functor F and relations R and S, $\mathsf{F}(R \Rightarrow S) \subseteq \mathsf{F}R \Rightarrow \mathsf{F}S$ holds.

Proof.

$$
\begin{array}{rl}
\mathsf{F}(R \Rightarrow S) \subseteq (\mathsf{F}R \Rightarrow \mathsf{F}S) \;\Leftrightarrow & \{\text{ property of } \Rightarrow \text{ }\} \\
& (\mathsf{F}(R \Rightarrow S) \cap \mathsf{F}R) \subseteq \mathsf{F}S \\
\Leftrightarrow & \{\text{ Lemma 2.7 }\} \\
& \mathsf{F}((R \Rightarrow S) \cap R) \subseteq \mathsf{F}S \\
\Leftarrow & \{\text{ relator }\} \\
& ((R \Rightarrow S) \cap R) \subseteq S \\
\Leftrightarrow & \{\text{ property of } \Rightarrow \text{ }\} \\
& \mathit{True}
\end{array}
$$

□

Following lemmas show relationships between right-division and relators.

Lemma 2.10. For any relator F and relations R and S, $\mathsf{F}(R/S) \subseteq \mathsf{F}R/\mathsf{F}S$ holds.

Proof.

$$
\begin{array}{rl}
\mathsf{F}(R/S) \subseteq \mathsf{F}R/\mathsf{F}S \;\Leftrightarrow & \{\text{ property of } / \text{ }\} \\
& (\mathsf{F}(R/S) \circ \mathsf{F}S) \subseteq \mathsf{F}R \\
\Leftrightarrow & \{\text{ functor }\} \\
& \mathsf{F}((R/S) \circ S) \subseteq \mathsf{F}R \\
\Leftarrow & \{\text{ relator }\} \\
& ((R/S) \circ S) \subseteq R \\
\Leftrightarrow & \{\text{ property of } / \text{ }\} \\
& \mathit{True}
\end{array}
$$

□

Lemma 2.11 (Lemma 8.3.1.2 in [dM92])**.** For any polynomial functor F and relations R and S, $\mathsf{F}((R/S) \cap S^\circ) = (\mathsf{F}R/\mathsf{F}S) \cap \mathsf{F}S^\circ$ holds. □

The following lemma shows a relationship between functors and the power transpose.

Lemma 2.12. For any relator F and relation S, $\Lambda\mathsf{F}S = \Lambda\mathsf{F}{\in} \circ \mathsf{F}\Lambda S$ holds.

Proof.

$$\begin{array}{rcl} \Lambda\mathsf{F}S = \Lambda\mathsf{F}{\in} \circ \mathsf{F}\Lambda S & \Leftrightarrow & \{ \text{ property of } \Lambda \ \} \\ & & \mathsf{F}S = {\in} \circ \Lambda\mathsf{F}{\in} \circ \mathsf{F}\Lambda S \\ & \Leftrightarrow & \{ \ {\in} \text{ cancels } \Lambda \text{ out } \} \\ & & \mathsf{F}S = \mathsf{F}{\in} \circ \mathsf{F}\Lambda S \\ & \Leftrightarrow & \{ \text{ functor } \} \\ & & \mathsf{F}S = \mathsf{F}({\in} \circ \Lambda S) \\ & \Leftrightarrow & \{ \ {\in} \text{ cancels } \Lambda \text{ out } \} \\ & & \mathsf{F}S = \mathsf{F}S \end{array}$$

□

2.4 Recursion Schemes and Inductions

In program calculation, recursion schemes, namely the ways to accomplish computations through recursions, play important roles. Recursion schemes implicate the way to prove properties by inductions, which are captured by *fusion laws*.

2.4.1 Automata

A finite state automaton, which represents scanning over sequences, is defined as follows.

Definition 2.13 (automaton)**.** A *finite state automaton* (also called *NFA, nondeterministic finite state automaton*) $\mathcal{A} = (S, \Sigma, \tau, S_I, S_F)$ consists of a finite set of states S, a finite alphabet Σ, a transition relation $\tau : (S \times \Sigma) \leftrightarrow S$, a set of initial states $S_I \subseteq S$, and a set of finial states $S_F \subseteq S$. □

An NFA $\mathcal{A} = (S, \Sigma, \tau, S_I, S_F)$ is *deterministic* if τ is a function and S_I is a singleton set. In this case, $\mathcal{A}$ is called a *deterministic finite state automaton (in short, DFA)*.

Given an NFA $\mathcal{A} = (S, \Sigma, \tau, S_I, S_F)$, the *representative relation* of $\mathcal{A}$, denoted by $rp_{\mathcal{A}} : \Sigma^* \leftrightarrow S$, is defined as follows.

$$\begin{array}{l} (s_{n+1}, [\sigma_0, \sigma_1, \ldots, \sigma_n]) \in rp_{\mathcal{A}} \\ \quad \overset{\text{def}}{\Longleftrightarrow} \exists s_0, \ldots, s_n \in S : \ s_0 \in S_I \wedge (0 \leq \forall i \leq n : \ (s_{i+1}, (s_i, \sigma_i)) \in \tau) \end{array}$$

An NFA $\mathcal{A} = (S, \Sigma, \tau, S_I, S_F)$ *accepts* a sequence of symbols $x \in \Sigma^*$ if $\Lambda rp_{\mathcal{A}}(x) \cap S_F \neq \emptyset$ holds. $L_{\mathcal{A}} \subseteq \Sigma^*$ denotes the set of all sequences that $\mathcal{A}$ accepts.

Given an NFA $\mathcal{A} = (S, \Sigma, \tau, S_I, S_F)$ and a set of state $S' \subseteq S$, we will write another NFA $(S, \Sigma, \tau, S', S_F)$ by $\mathcal{A}[S']$ for notational convenience. By definition, the NFA $\mathcal{A}[S']$ the same as $\mathcal{A}$ except for their initial state.

Many operations are known for automata. One is *product construction*. Given two NFAs, product construction derives an NFA that is the composition of two NFAs in the sense that the derived NFA accepts a sequence if and only if both of the original two accept it.

Definition 2.14 (product construction)**.** Given two NFAs $\mathcal{A} = (A, \Sigma, \alpha, A_I, A_F)$ and $\mathcal{B} = (B, \Sigma, \beta, B_I, B_F)$, the product of $\mathcal{A}$ and $\mathcal{B}$ is an NFA $\mathcal{C} = (A \times B, \Sigma, \gamma, A_I \times B_I, A_F \times B_F)$, where γ is defined as follows.

$$((a', b'), ((a, b), \sigma)) \in \gamma \overset{\text{def}}{\Leftrightarrow} (a', (a, \sigma)) \in \alpha \wedge (b', (b, \sigma)) \in \beta \qquad \square$$

Lemma 2.15. Given two NFAs $\mathcal{A}$ and $\mathcal{B}$, the product of $\mathcal{A}$ and $\mathcal{B}$ accepts a sequence x if and only if both $\mathcal{A}$ and $\mathcal{B}$ accept x.

Proof. Let $\mathcal{A} = (A, \Sigma, \alpha, A_I, A_F)$, $\mathcal{B} = (B, \Sigma, \beta, B_I, B_F)$, the product of $\mathcal{A}$ and $\mathcal{B}$ be $(A \times B, \Sigma, \gamma, A_I \times B_I, A_F \times B_F)$, and $x = [\sigma_0, \sigma_1, \ldots, \sigma_n]$.

$$\begin{array}{l}
x \in L_{(A \times B, \Sigma, \gamma, A_I \times B_I, A_F \times B_F)} \\
\Leftrightarrow \ \{ \text{ definition of accept } \} \\
\quad \exists (a_0, b_0), (a_1, b_1), \ldots (a_{n+1}, b_{n+1}) \in (A \times B) : \\
\quad (a_0, b_0) \in (A_I \times B_I) \wedge (a_{n+1}, b_{n+1}) \in (A_F \times B_F) \wedge \\
\quad (0 \leq \forall i \leq n : ((a_{i+1}, b_{i+1}), ((a_i, b_i), \sigma_i)) \in \gamma) \\
\Leftrightarrow \ \{ \text{ definition of } \gamma \ \} \\
\quad \exists (a_0, b_0), (a_1, b_1), \ldots (a_{n+1}, b_{n+1}) \in (A \times B) : \\
\quad a_0 \in A_I \wedge b_0 \in B_I \wedge a_{n+1} \in A_F \wedge b_{n+1} \in B_F \wedge \\
\quad (0 \leq \forall i \leq n : (a_{i+1}, (a_i, \sigma_i)) \in \alpha \wedge (b_{i+1}, (b_i, \sigma_i)) \in \beta) \\
\Leftrightarrow \ \{ \text{ distributivity of } \forall \text{ to } \wedge \ \} \\
\quad (\exists a_0, \ldots a_{n+1} \in A : a_0 \in A_I \wedge a_{n+1} \in A_F \wedge (0 \leq \forall i \leq n : (a_{i+1}, (a_i, \sigma_i)) \in \alpha)) \wedge \\
\quad (\exists b_0, \ldots b_{n+1} \in B : b_0 \in B_I \wedge b_{n+1} \in B_F \wedge (0 \leq \forall i \leq n : (b_{i+1}, (b_i, \sigma_i)) \in \beta)) \\
\Leftrightarrow \ \{ \text{ definition of accept } \} \\
\quad x \in L_{\mathcal{A}} \wedge x \in L_{\mathcal{B}}
\end{array}$$

$\square$

2.4.2 Catamorphisms

Although automata are useful to formalize predicates on sequences, they cannot cope with generic computations. Catamorphisms [MFP91,Fok92,Mei92,BdM96] are a more general computation pattern that can express a larger class of computations.

Definition 2.16 (algebra)**.** For a functor F, an F-*algebra* is a pair (A, ψ), where A is an object and $\psi : \mathsf{F}A \to A$ is a morphism. $\square$

Definition 2.17 (algebra morphism)**.** For two F-algebras $\mathcal{A} = (A, \psi)$ and $\mathcal{B} = (B, \phi)$, an algebra morphism from $\mathcal{A}$ to $\mathcal{B}$ is a morphism $h : A \to B$ that makes the following diagram commute.

$$\begin{array}{ccc} \mathsf{F}A & \xrightarrow{\psi} & A \\ {\scriptstyle \mathsf{F}h}\downarrow & & \downarrow{\scriptstyle h} \\ \mathsf{F}B & \xrightarrow[\phi]{} & B \end{array}$$

□

Definition 2.18 (initial algebra and catamorphism)**.** An F-algebra $(A, \mathsf{in}_\mathsf{F})$ is said to be *initial* if for each F-algebra (B, ϕ) there exists a unique algebra morphism from $(A, \mathsf{in}_\mathsf{F})$ to (B, ϕ). The unique morphism is called *catamorphism* and denoted by $([\phi])_\mathsf{F}$. □

The catamorphism $([\phi])_\mathsf{F}$ is well-defined because the initial F-algebra is unique up to isomorphism. We may omit subscripts for catamorphisms if they are clear from their context.

It is known that for each polynomial functor F and each bijective function $f : \mathsf{F}\mu\mathsf{F} \to \mu\mathsf{F}$, $(\mu\mathsf{F}, f)$ forms an initial algebra. Recall that polynomial functors correspond to tree-like data structures. Since bottom-up stepwise construction of tree-like structure is a bijective computation, we can recognize initial algebras as such constructions.

As an example, let us consider catamorphisms on sequences. Recall that sequences of integers can be captured by the least fixed point of the polynomial functor $\mathsf{S} = !_1 + !_\mathbb{Z} \times \mathsf{I}$. Define a function $\mathsf{in}_\mathsf{S} : \mathsf{S}\mu\mathsf{S} \to \mu\mathsf{S}$ by $\mathsf{in}_\mathsf{S} \stackrel{\text{def}}{=} (\mathsf{const}_{[]} \triangledown \mathit{cons})$, where $\mathit{cons} : (\mathbb{Z}, \mathbb{Z}^*) \to \mathbb{Z}^*$ is defined by $\mathit{cons}(a, x) \stackrel{\text{def}}{=} [a] +\!\!+ x$. Then, since in_S is bijective, $(\mu\mathsf{S}, \mathsf{in}_\mathsf{S})$ forms an initial S-algebra. Next, consider a catamorphism $([\phi])_\mathsf{S}$, where $\phi = (b \triangledown f)$, where $b : \mathsf{1} \to A$ and $f : (\mathbb{Z}, A) \to A$ are given functions. Since in_S is bijective, $([\phi])_\mathsf{S}$ satisfies an equation $([\phi])_\mathsf{S} = \phi \circ \mathsf{S}([\phi])_\mathsf{S} \circ {\mathsf{in}_\mathsf{S}}^\circ$; thus, $([\phi])_\mathsf{S}$ can be seen as the following recursive function.

$$\begin{array}{lcl} ([\phi])_\mathsf{S}([]) & = & b() \\ ([\phi])_\mathsf{S}([a] +\!\!+ x) & = & f(a, ([\phi])(x)) \end{array}$$

For example, $([(\mathsf{const}_1 \triangledown \mathit{succ})])_\mathsf{S}$ where $\mathit{succ}(a, n) \stackrel{\text{def}}{=} 1 + n$ corresponds to the function to compute the length of a list, and $([(\mathsf{const}_{[]} \triangledown \mathit{cons} \circ (f \times \mathit{id}))])_\mathsf{S}$ corresponds to the function map_f. It is worth noting that catamorphisms on lists is equivalent to functions written by foldr, namely $\mathsf{foldr}_{\oplus,e}$ corresponds to $([(\mathsf{const}_e \triangledown (\oplus))])_\mathsf{S}$.

As another example, let us consider catamorphisms on node-valued binary trees, which are characterized by the least fixed point of the polynomial functor $\mathsf{T} = !_1 + \mathbb{N} \times (\mathsf{I} \times \mathsf{I})$. Let $\mathsf{in}_\mathsf{T} : \mathsf{T}\mu\mathsf{T} \to \mu\mathsf{T}$ be a function such that $\mathsf{in}_\mathsf{T} \stackrel{\text{def}}{=} (\mathsf{const}_{\mathit{Leaf}} \triangledown \mathit{node})$ where $\mathit{node}(a, (t_1, t_2)) \stackrel{\text{def}}{=} \mathit{Node}(a, t_1, t_2)$. Then, $(\mu\mathsf{T}, \mathsf{in}_\mathsf{T})$ forms an initial algebra, and a catamorphism $([(l \triangledown f)])_\mathsf{T}$ corresponds to the following bottom-up recursive function.

$$\begin{array}{lcl} ([(l \triangledown f)])_\mathsf{T}(\mathit{Leaf}) & = & l() \\ ([(l \triangledown f)])_\mathsf{T}(\mathit{Node}(a, (t_1, t_2))) & = & f(a, (([(l \triangledown f)])_\mathsf{T}(t_1), ([(l \triangledown f)])_\mathsf{T}(t_2))) \end{array}$$

For example, $(\![(\mathsf{const}_0 \triangledown add)]\!)$ where $add(a,(x,y)) = a + x + y$ corresponds to a function that sums up the values of all elements in a tree.

As seen, catamorphisms express bottom-up computations on tree-like structures. Since the structure of recursive calls is inductive, induction is effective for proving properties of catamorphisms. Induction on tree-like structures is formalized as the following fusion law.

Theorem 2.19 (cata fusion [MFP91, BdM96]).

$$(f \circ \phi = \psi \circ \mathsf{F}f) \Leftrightarrow (f \circ (\!|\phi|\!)_\mathsf{F} = (\!|\psi|\!)_\mathsf{F}) \qquad \square$$

Proof.

$$\begin{array}{ccc}
\mathsf{F}\mu\mathsf{F} & \xrightarrow{\mathsf{in}_\mathsf{F}} & \mu\mathsf{F} \\
\downarrow \mathsf{F}(\!|\phi|\!)_\mathsf{F} & & (\!|\phi|\!)_\mathsf{F} \downarrow \\
\mathsf{F}A & \xrightarrow[\phi]{} & A \\
\downarrow \mathsf{F}f & & f \downarrow \\
\mathsf{F}B & \xrightarrow[\psi]{} & B
\end{array}
\qquad \mathsf{F}(\!|\psi|\!)_\mathsf{F} : \mathsf{F}\mu\mathsf{F} \to \mathsf{F}B, \quad (\!|\psi|\!)_\mathsf{F} : \mu\mathsf{F} \to B$$

$\square$

Notice that $(\!|\phi|\!)_\mathsf{F}$ is an inductive computation on $\mu\mathsf{F}$. Therefore, the premise of Theorem 2.19, namely $f \circ \phi = \psi \circ \mathsf{F}f$, means that we can inductively (incrementally) compute the value of $f \circ \phi$ on $\mu\mathsf{F}$. In other words, Theorem 2.19 is a formalization of successful inductive proofs. It is worth noting that similar theorems to Theorem 2.19 hold on the category *Rel*: Both $(f \circ \phi \subseteq \psi \circ \mathsf{F}f) \Leftrightarrow (f \circ (\!|\phi|\!)_\mathsf{F} \subseteq (\!|\psi|\!)_\mathsf{F})$ and $(f \circ \phi \supseteq \psi \circ \mathsf{F}f) \Leftrightarrow (f \circ (\!|\phi|\!)_\mathsf{F} \supseteq (\!|\psi|\!)_\mathsf{F})$ hold if F is a relator.

As an example, let us consider a fusion transformation of two map macros, namely Theorem 1.1. Recall that $(\![(\mathsf{const}_{[]} \triangledown cons \circ (g \times id))]\!)_\mathsf{S}$ corresponds to map_g.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{map}_f \circ \mathsf{map}_g = \mathsf{map}_{f \circ g}} \\
\Leftrightarrow & \{ \text{ catamorphism } \} \\
& \mathsf{map}_f \circ (\![(\mathsf{const}_{[]} \triangledown cons \circ (g \times id))]\!)_\mathsf{S} = (\![(\mathsf{const}_{[]} \triangledown cons \circ ((f \circ g) \times id))]\!)_\mathsf{S} \\
\Leftrightarrow & \{ \text{ Theorem 2.19 } \} \\
& \mathsf{map}_f \circ (\mathsf{const}_{[]} \triangledown cons \circ (g \times id)) = (\mathsf{const}_{[]} \triangledown cons \circ ((f \circ g) \times id)) \circ \mathsf{S}\mathsf{map}_f \\
\Leftarrow & \{ \text{ distributivity } \} \\
& (\mathsf{map}_f \circ \mathsf{const}_{[]} \triangledown \mathsf{map}_f \circ cons \circ (g \times id)) \\
& \quad = (\mathsf{const}_{[]} \triangledown cons \circ ((f \circ g) \times id)) \circ \mathsf{S}\mathsf{map}_f \\
\Leftrightarrow & \{ \ \mathsf{map}_f \circ \mathsf{const}_{[]} = \mathsf{const}_{[]} \text{ and } \mathsf{map}_f \circ cons = cons \circ (f \times \mathsf{map}_f) \ \} \\
& (\mathsf{const}_{[]} \triangledown cons \circ (f \times \mathsf{map}_f) \circ (g \times id)) \\
& \quad = (\mathsf{const}_{[]} \triangledown cons \circ ((f \circ g) \times id)) \circ \mathsf{S}\mathsf{map}_f \\
\Leftrightarrow & \{ \text{ composition, and } \mathsf{map}_f \circ id = \mathsf{map}_f \ \} \\
& (\mathsf{const}_{[]} \triangledown cons \circ ((f \circ g) \times \mathsf{map}_f)) = (\mathsf{const}_{[]} \triangledown cons \circ ((f \circ g) \times id)) \circ \mathsf{S}\mathsf{map}_f \\
\Leftrightarrow & \{ \text{ definition of } \mathsf{S} \ \} \\
& \mathit{True}
\end{array}$$

Then, this calculation provides a proof of Theorem 1.1 by Theorem 2.19. In other words, Theorem 2.19 is a generalization of Theorem 1.1. Compare this calculation to the unfolding-folding calculation we have done in Section 1.2. The premise of Theorem 2.19 corresponds to a sufficient condition for successful folding. From this viewpoint, we can recognize Theorem 2.19 as an effective strategy for unfolding-folding.

As the final remark about catamorphisms, we introduce the following lemma that shows relationship between catamorphisms and the power transpose.

Lemma 2.20 (Eilenberg-Wright [BdM93a, BdM96])**.**

$$\Lambda(\![S]\!)_{\mathsf{F}} = (\![\Lambda(S \circ \mathsf{F}{\in})]\!)_{\mathsf{F}}$$

Proof.

$$\begin{array}{rll}
\Lambda(\![S]\!)_{\mathsf{F}} = (\![\Lambda(S \circ \mathsf{F}{\in})]\!)_{\mathsf{F}} & \Leftrightarrow & \{\text{ power transpose }\} \\
& & (\![S]\!)_{\mathsf{F}} = {\in} \circ (\![\Lambda(S \circ \mathsf{F}{\in})]\!)_{\mathsf{F}} \\
& \Leftarrow & \{\text{ Theorem 2.19 }\} \\
& & S \circ \mathsf{F}{\in} = {\in} \circ \Lambda(S \circ \mathsf{F}{\in}) \\
& \Leftrightarrow & \{\text{ canceling } \Lambda \text{ and } {\in} \text{ }\} \\
& & \mathit{True}
\end{array}$$

□

2.4.3 Reflexive Transitive Closures

Although catamorphisms are an expressive recursion schema, there are computations that are hardly captured by catamorphisms, such as a traverse on graphs. There have been a lot of studies for expressing graph-traversing computations by catamorphisms or catamorphism-like recursion schemes [Gib95,KL95,Erw97,SHT00,Erw00,Erw01], yet it is unclear whether catamorphism-like schemes can express practical graph traversing functions and be useful for practical calculations. Instead of them, we will use the notion of *reflexive transitive closures*. A comparison between catamorphisms and reflexive transitive closure was made by Curtis [Cur96].

Definition 2.21 (reflexive transitive closure)**.** For a relation $R{:}A \to A$, the reflexive transitive closure of R, denoted by $R{*}$, is defined as follows.

$$R{*} \stackrel{\text{def}}{=} \{(a,b) \mid \exists n \in \mathbb{Z}_{+} :\ (a,b) \in R^{n}\}$$

□

The reflexive transitive closure of R considers all cases that are obtained by repeated application of R. Therefore, the while-loop idiom "apply a computation R while it satisfies a predicate p" can be easily expressed by a reflexive transitive closure $(\mathit{not} \circ p)? \circ (R \circ p?){*}$, where $\mathit{not}(a) \stackrel{\text{def}}{=} \neg a$. In other words, reflexive transitive closures correspond to while-loop like iterations.

Let us consider Euclidean algorithm as an example. The following relation GCD yields the greatest common divisor of given two natural numbers.

$$\begin{aligned} GCD &\stackrel{\text{def}}{=} \pi_1 \circ (isZero \circ \pi_2)? \circ (step \circ (isNotZero \circ \pi_2)?)* \\ step(n,m) &\stackrel{\text{def}}{=} (m, mod(n,m)) \\ mod(n,m) = k &\stackrel{\text{def}}{\Longleftrightarrow} 0 \leq k < m \wedge \exists c \in \mathbb{Z}_+ : n = m \times c + k \\ isZero(a) &\stackrel{\text{def}}{=} a = 0 \\ isNotZero(a) &\stackrel{\text{def}}{=} a \neq 0 \end{aligned}$$

The computation of GCD can be interpreted as follows. At each step of recursion, we divide the first integer by the second one, which should be the reminder calculated at the previous recursion, and go to the next recursion with the smaller one and the reminder given by the division. The recursion continues until the second integer becomes zero. When the second is zero, the first is the greatest common divisor of initial values.

Reflexive transitive closures satisfy the following promotion law, which characterizes induction on their computation.

Theorem 2.22 (promotion law for reflexive transitive closures)**.**

$$(f \circ \phi = \psi \circ f) \Rightarrow (f \circ \phi* = \psi* \circ f)$$

Proof. From the definition of reflexive transitive closures, it is sufficient to show $\forall n \in \mathbb{Z}_+ : f \circ \phi^n = \psi^n \circ f$ holds. The claim is proved by induction. The claim obviously holds for the base case, namely the case of $n = 0$. The step case is proved by the following calculation.

$$\begin{aligned} f \circ \phi^{k+1} &= \{ \text{ definition of the power notation } \} \\ &\quad f \circ \phi \circ \phi^k \\ &= \{ \text{ assumption } \} \\ &\quad \psi \circ f \circ \phi^k \\ &= \{ \text{ induction hypothesis } \} \\ &\quad \psi \circ \psi^k \circ f \\ &= \{ \text{ definition of the power notation } \} \\ &\quad \psi^{k+1} \circ f \end{aligned}$$

□

Similar to the case of catamorphisms, the inequality-versions of Theorem 2.22 also hold, namely $(f \circ \phi \subseteq \psi \circ f) \Rightarrow (f \circ \phi* \subseteq \psi* \circ f)$ and $(f \circ \phi \supseteq \psi \circ f) \Rightarrow (f \circ \phi* \supseteq \psi* \circ f)$.

The following lemma shows relationship between reflexive transitive closures and the power transpose.

Lemma 2.23 (Eilenberg-Wright [Cur96])**.**

$$\Lambda(S*) = (\Lambda(S \circ \in))* \circ \{\cdot\}$$

Proof.

$$
\begin{array}{rll}
\Lambda(S*) = (\Lambda(S \circ \in))* \circ \{\cdot\} & \Leftrightarrow & \{ \text{ power transpose } \} \\
 & & S* = \in \circ (\Lambda(S \circ \in))* \circ \{\cdot\} \\
 & \Leftarrow & \{ \text{ Theorem 2.22 and } \in \circ \{\cdot\} = id \ \} \\
 & & S \circ \in = \in \circ \Lambda(S \circ \in) \\
 & \Leftrightarrow & \{ \text{ canceling } \Lambda \text{ and } \in \} \\
 & & \mathit{True}
\end{array}
$$

□

2.4.4 Incrementality

Theorems 2.19 and 2.22 are formalization of successful induction, and the premise of the theorems are useful to formalize properties that can be verified by induction. Therefore, we would like to name the premise.

Definition 2.24 (incremental)**.** For relations $R{:}A \leftrightarrow B$, $S{:}\mathsf{F}A \leftrightarrow A$, and $S'{:}B \to B$, R is said to be incremental on S by S' if $R \circ S = S' \circ \mathsf{F}R$ holds. □

The following lemma clarifies a relationship between the notion of incrementality and induction.

Lemma 2.25. For a function $f : A \to B$ and a relation $S : \mathsf{F}A \leftrightarrow A$, $f(a) = f(b)$ implies $\Lambda(f \circ S)(a) = \Lambda(f \circ S)(b)$ if f is incremental on S.

Proof. Let S' be the relation such that $f \circ S = S' \circ \mathsf{F}f$ holds. Then, equations $\Lambda(f \circ S)(a) = \Lambda(S' \circ \mathsf{F}f)(a) = \Lambda(S' \circ \mathsf{F}f)(b) = \Lambda(f \circ S)(b)$ hold. □

As seen in Lemma 2.25, when a function f is incremental on a relation S, S does not disorder information drawn by f. In other words, the incrementality condition indicates that if elements are generated by a repeated application of S, induction on computations of S will enable us to prove properties of information drawn by f. For example, consider the case where a predicate f is incremental on S by S' such that $\Lambda S'(\mathit{True}) = \{\mathit{True}\}$ holds; then, $f(a)$ implies $f(a')$ for any $a' \in \Lambda S(a)$, and thus, the property f on repetition of S will be successfully proved by induction. In summary, incrementality condition is important not only for fusion transformations but also other calculations.

It is worth noting there are many studies for automatic implementation of the theorems [SdM01, YHT05, Yok06]. Therefore, we can automatically verify incrementality conditions by applying such studies.

Chapter 3

Calculational Laws for Combinatorial Optimization Problems

Combinatorial optimization problems are problems to find the optimal solution in a set of feasible solutions, where solutions are discrete. For example, all of finding the shortest path, finding the optimal packing, and finding the optimal scheduling are combinatorial optimization problems. Since combinatorial optimization problems have a great many applications, they are recognized as one of the most important classes of problems in algorithm construction.

In this chapter, we will review existing calculational laws to develop efficient algorithms for combinatorial optimization problems. First in Section 3.1, we formalize generic specification of combinatorial optimization problems in terms of program calculation. Then, in the next two sections, we introduce two kinds of calculational laws: greedy theorems and unified solutions for maximum marking problems. Supplemental lemmas are shown in the last section, Section 3.4.

We introduce greedy theorems in Section 3.2. Greedy theorems, which are formalized by Bird, de Moor, and Curtis [BdM93a, BdM93b, dM95, BdM96, Cur96, Cur03], are calculational laws that show a generic way for calculating efficient algorithms for combinatorial optimization problems. Although the theorems capture really a large class of problems, they are not easy to use. They require a certain property, which is generally difficult to confirm or obtain.

Next, in Section 3.3, we review a unified solution of maximum marking problems [ALS91, BPT92, SHTO00]. Arnborg et al., Borie et al., and Sasano et al., independently showed that a class of combinatorial optimization problems, called maximum marking problems, can be solved automatically once it is specified in certain forms. Their results are easy to use even for nonspecialists, because they are fully automatic; however, their domain is not sufficiently large, and many interesting problems cannot be coped with them. Bird [Bir01] showed a relationship between the results about maximum marking problems and greedy theorems. The study

of Bird seems to indicate that there are both generic and useful calculational laws between them, though no concrete result has been shown yet.

3.1 Calculational Formalization of Combinatorial Optimization Problems

3.1.1 Orders

Before introducing the notion of combinatorial optimization problems, we would like to prepare some notions concerning orders, which are used for describing optimality. We use *quasi-order* (also called *preorder*) to formalize optimization problems.

Definition 3.1 (quasi-order)**.** A relation $R : A \leftrightarrow A$ is called *quasi-order* if the following two properties are satisfied.

$$\begin{array}{ll} \forall a \in A : \; a \, R \, a & \text{(reflectivity)} \\ (a \, R \, b \wedge b \, R \, c) \Rightarrow a \, R \, c & \text{(transitivity)} \end{array} \qquad \Box$$

Definition 3.2 (totality)**.** A relation $R : A \leftrightarrow A$ is said to be *total* if $a \, R \, b \vee b \, R \, a$ holds for all $a \in A$ and $b \in A$. $\Box$

In this thesis, we read $a \, R \, b$ as "a is smaller than b" or "a is preferred to b" when R is a quasi-order.

Equivalence relations and linear orders[1] are important classes of quasi-orders.

Definition 3.3 (equivalence relation)**.** A quasi-order $R : A \leftrightarrow A$ is called a *equivalence relation* if the following property is satisfied.

$$a \, R \, b \Leftrightarrow b \, R \, a \quad \text{(symmetry)} \qquad \Box$$

Definition 3.4 (linear order)**.** A total quasi-order $R : A \leftrightarrow A$ is called a *linear order* if the following property is satisfied.

$$(a \, R \, b \wedge b \, R \, a) \Leftrightarrow a = b \quad \text{(antisymmetry)} \qquad \Box$$

It is known that the sequential composition (also called lexicographic composition) of two quasi-orders is a quasi-order.

Definition 3.5 (sequential composition of two orders)**.** For two relations $R : A \leftrightarrow A$ and $S : A \leftrightarrow A$, the *sequential composition* of R and S, denoted by $R \mathbin{;} S$, is defined as follows[2].

$$a \, (R \mathbin{;} S) \, b \overset{\text{def}}{\Longleftrightarrow} a \, S \, b \wedge (b \, \overline{S} \, a \vee a \, R \, b) \qquad \Box$$

[1] Linear orders are also called *total orders*. We do not use the name to avoid the confusion between total orders and total quasi-orders.

[2] $R \mathbin{;} S$ is denoted by $S \, ; R$ in [BdM96].

The sequential composition of two quasi-orders R and S, namely $R\,;S$, is a quasi-order, where the ordering is the same as S expect for equivalent elements in S, and equivalent elements in S are compared by R. The operator $;$ is associative. Note that the following definition is equivalent to the definition above.

$$R\,;S \stackrel{\text{def}}{=} S \cap (S^\circ \Rightarrow R)$$

We make a special use of $=$ and $<$. For a quasi-order R, $\stackrel{R}{=}$ denotes the equivalent part of R, i.e., $a \stackrel{R}{=} b \stackrel{\text{def}}{\iff} a\,R\,b \wedge b\,R\,a$. Similarly, for a quasi-order R, $\stackrel{R}{<}$ denotes the strict part of R, i.e., $a \stackrel{R}{<} b \stackrel{\text{def}}{\iff} a\,R\,b \wedge b\,\overline{R}\,a$. We use a function to produce an order from an order. For a function $g : B \to A$ and a quasi-order $R : A \leftrightarrow A$, a quasi-order $R_g : B \leftrightarrow B$ is defined by $a\;R_g\;b \stackrel{\text{def}}{\iff} g(a)\;R\;g(b)$ where both $g(a)$ and $g(b)$ must be defined. The following equations show alternative definitions of them.

$$\begin{aligned} \stackrel{R}{=} &\stackrel{\text{def}}{=} R \cap R^\circ \\ \stackrel{R}{<} &\stackrel{\text{def}}{=} R \cap \overline{R^\circ} \\ R_g &\stackrel{\text{def}}{=} g^\circ \circ R \circ g \end{aligned}$$

3.1.2 Minimums and Minimals

To extract minimum elements, we use an operator *min*. For a relation $R : A \leftrightarrow A$, the relation $min_R : \mathsf{P}A \leftrightarrow A$ is defined as follows.

$$(a, X) \in min_R \stackrel{\text{def}}{\iff} \forall b \in X :\; a\,R\,b$$

We can also give an equivalent definition in point-free style as follows.

$$min_R \stackrel{\text{def}}{=} \in \cap\, R/\ni$$

Given a total quasi-order R, $min_R(X)$ yields the minimum elements in X based on the order R. It is worth noting that min_R is not very useful if R is not total. For example, assume neither $a\,R\,b$ nor $b\,R\,a$ holds; then, $\Lambda min_R(X \cup \{a, b\})$ is empty unless X contains an element c such both $c\,R\,a$ and $c\,R\,b$ hold. In such cases, it is appropriate to use another operator *mnl* which extracts minimal elements, because mnl_R works well even when R is not total.

$$(a, X) \in mnl_R \stackrel{\text{def}}{\iff} \forall b \in X :\; b\,R\,a \Rightarrow a\,R\,b$$

The following equation characterized the operator *mnl*.

$$mnl_R = min_{R^\circ \Rightarrow R}$$

Even if a quasi-order R is total, $\Lambda min_R(X)$ may yield an empty set for non-empty X. For example, $\Lambda min_{\leq}(\mathbb{Z})$ is empty, because there exists no least element in integers. It is sometimes useful to exclude such peculiar cases for formalizing calculational laws, and for this purpose, we introduce two notions: *well-bounded* and *well-supported*.

Definition 3.6 (well-bounded [BdM92, BdM96])**.** A relation $R : A \leftrightarrow A$ is well-bounded if $\Lambda min_R(X)$ is nonempty for any nonempty set $X \subseteq A$. □

The following inequality provides alternative characterization of well-bounded relations; in other words, a relation R is well-bounded if and only if the following inequality holds.

$$\in \subseteq R^\circ \circ min_R$$

Definition 3.7 (well-supported [BdM92, BdM96])**.** A relation $R : A \leftrightarrow A$ is well-supported if $\Lambda mnl_R(X)$ is nonempty for any nonempty set $X \subseteq A$. □

Similar to well-bounded relations, the following inequality provides alternative characterization of well-supported relations.

$$\in \subseteq R^\circ \circ mnl_R$$

3.1.3 Generic Specification of Combinatorial Optimization Problems

Now let us provide a generic specification of combinatorial optimization problems based on program calculations. A combinatorial optimization problem is a problem whose objective is to find the best solution among those being feasible. We can express specifications of combinatorial optimization problems by a relation, a predicate, and a quasi-order: A relation *candidates* specifies candidates of solutions, a predicate *feasible* tests whether a solution is feasible or not, and an order R determines which solution is better.

$$min_R \circ feasible\lhd \circ \Lambda candidates$$

If the relation *candidates* has no specific structure, then there is little hope to obtain the best solution efficiently. In other words, efficient algorithms have been studied for cases where *candidates* has a certain structure.

In usual, each candidate of solution is specified by a sequence of nondeterministic choices in a combinatorial optimization problem. Let S be a relation that corresponds to a nondeterministic choice. Then, *candidates* may be expressed by S with a catamorphism.

$$candidates = (\![S]\!)$$

Or, in some cases, a transitive closure may be appropriate.

$$candidates = S*$$

Therefore, we would like to provide effective calculational laws for problems described in the following forms.

$$min_R \circ feasible\lhd \circ \Lambda(\![S]\!)$$
$$min_R \circ feasible\lhd \circ \Lambda(S*)$$

It is worth noting that *min* can perform the computation of $\mathit{feasible}\lhd$. Let Q be the relation whose definition is $a\,Q\,b \overset{\text{def}}{\Longleftrightarrow} \mathit{feasible}(a) \vee \neg\mathit{feasible}(b)$; then Q is a total quasi-order, on which elements that satisfy *feasible* are smaller than those that do not. Now the following expression is equivalent to the specification above.

$$\mathit{min}_{R;Q} \circ \Lambda \mathit{candidates}$$

In summary, our goal is to provide calculational laws for composition of *min* and enumeration of candidates.

As an example, let us consider shortest path problems. Given a graph (V, E), a weight function $w : E \to \mathbb{R}$, and two ends $s, t \in V$, a shortest path problem is the problem to find the minimum-weighted path from the source s to the destination t. Since it is natural to enumerate paths by repetitions of extensions of a path, we prepare a function $\mathit{extend}_e : E^* \to E^*$ defined as follows.

$$\mathit{extend}_e(p) \overset{\text{def}}{=} p +\!\!+ [e] \quad \text{if } p = [\,] \vee \mathit{dst}(p) = \mathit{hd}(e)$$

The function extend_e extends a path by an edge e whenever it makes a path. Then, the enumeration of all paths is specified as follows.

$$\mathit{paths} \overset{\text{def}}{=} (\textstyle\bigcup_{e \in E} \mathit{extend}_e)*$$

For determining optimality, we need to prepare two predicates together with a quasi-order $\leq_w$ that compares paths by their weight. The predicate from_s checks whether a path starts from the vertex s, and the predicate $\mathit{endWith}_t$ checks whether a path bounds for the vertex t.

$$\begin{aligned} \mathit{from}_s([\,]) &\overset{\text{def}}{=} \mathit{False} \\ \mathit{from}_s([e] +\!\!+ p) &\overset{\text{def}}{=} \mathit{hd}(e) = s \\ \mathit{endWith}_t(p) &\overset{\text{def}}{=} \mathit{dst}(p) = t \end{aligned}$$

In all, shortest path problems are formalized as follows.

$$\mathit{SP} \overset{\text{def}}{=} (\mathit{min}_{\leq_w} \circ \mathit{endWith}_t\lhd \circ \mathit{from}_s\lhd \circ \Lambda \mathit{paths})\ [\,]$$

As explained, we can merge the filters, namely from_s and $\mathit{endWith}_t$, into the *min* operator by defining the following total quasi-order $R_{s,t}$.

$$\begin{aligned} & p_1\, R_{s,t}\, p_2 \\ \overset{\text{def}}{\Longleftrightarrow}\ & (\mathit{endWith}_t(p_1) \wedge \mathit{from}_s(p_1) \wedge \mathit{endWith}_t(p_2) \wedge \mathit{from}_s(p_2) \wedge p_1 \leq_w p_2) \vee \\ & (\mathit{endWith}_t(p_1) \wedge \mathit{from}_s(p_1) \wedge \neg(\mathit{endWith}_t(p_2) \wedge \mathit{from}_s(p_2))) \vee \\ & (\neg(\mathit{endWith}_t(p_1) \wedge \mathit{from}_s(p_1)) \wedge \neg(\mathit{endWith}_t(p_2) \wedge \mathit{from}_s(p_2)) \wedge p_1 \leq_w p_2) \end{aligned}$$

Then, the shortest path problem is specified as follows if there is a path from the source s to the destination t.

$$\mathit{SP} = (\mathit{min}_{R_{s,t}} \circ \Lambda \mathit{paths})\ [\,]$$

3.2 Greedy Theorems

In this subsection, we introduce *greedy theorems* and *thinning theorems*, which were proposed by by Bird, de Moor, and Curtis [BdM93b, BdM93a, dM95, BdM96, Cur96, Cur03]. The theorems provide a formalization of derivation of efficient algorithms for combinatorial optimization problems. They mainly considered obtaining one best solution of each combinatorial optimization problem; here, in addition to them, we propose theorems for enumerating all the best solutions.

3.2.1 Monotonicity and Greedy Theorems

As shown in many studies [Mor82, BdM93b, BdM96, Cur96, Cur03], *monotonicity* is the key to efficient algorithms.

Definition 3.8 (monotone)**.** A relation $S : \mathsf{F}A \leftrightarrow A$ is *monotonic* on a relation $R : A \leftrightarrow A$ if the following property holds.

$$\forall a_1, a_2 \in \mathsf{F}A, a_1' \in A : (a_1 \, \mathsf{F}R \, a_2 \wedge a_1' \, S \, a_1) \Rightarrow (\exists a_2' \in A : a_2' \, S \, a_2 \wedge a_1' \, R \, a_2') \quad \Box$$

Alternative characterization of monotonicity is the following inequality.

$$S \circ \mathsf{F}R \subseteq R \circ S$$

Note that monotonicity intuitively means that *larger* elements yield *larger* ones. We will also use variants of monotonicity.

Definition 3.9 (strictly monotone)**.** A relation $S : \mathsf{F}A \leftrightarrow A$ is *strictly monotonic* on a relation $R : A \leftrightarrow A$ if the following inequality holds.

$$\forall a_1, a_2 \in \mathsf{F}A, a_1' \in A : (a_1 \overset{\mathsf{F}R}{<} a_2 \wedge a_1' \, S \, a_1) \Rightarrow (\exists a_2' \in A : a_2' \, S \, a_2 \wedge a_1' \overset{R}{<} a_2') \quad \Box$$

Definition 3.10 (completely monotone)**.** A relation S is *completely monotonic* on R if S is both monotonic and strictly monotonic on R. □

As similar to the case of monotonicity, strictly monotonicity is also characterized by the following inequality.

$$S \circ \overset{\mathsf{F}R}{<} \subseteq \overset{R}{<} \circ S$$

Now let us introduce the greedy theorems, which provide formalizations of derivation of efficient algorithms for combinatorial optimization problems.

Theorem 3.11 (greedy theorem for catamorphqisms [BdM93b, BdM96])**.** If a relation $S : \mathsf{F}A \leftrightarrow A$ is monotonic on a quasi-order R°, then the following inequality holds.

$$min_R \circ \Lambda(\!|S|\!) \supseteq (\!|min_R \circ \Lambda S|\!) \quad \Box$$

Theorem 3.12 (greedy theorem for repetitions [Cur96, Cur03])**.** If a relation $S : A \leftrightarrow A$ is monotonic on a quasi-order R°, then the following inequality holds.

$$min_R \circ \Lambda(S*) \supseteq (min_R \circ \Lambda S)* \qquad \square$$

The statements of greedy theorems are natural. Recall that monotonicity means that *larger* elements yield *larger* ones. Thus, the monotone property on R° implies that smaller elements are produced from smaller elements. Therefore, we can discard non-minimum elements at each step to obtain the minimum solution.

The greedy theorem is formalized by inequality, which means that the efficient procedure may not generate all of minimum solutions. Someone may worry that the efficient procedure would yield an empty result. Actually the efficient procedure yields at least one solution if there is a feasible solution and the quasi-order R is well-bounded, which is easily proved by induction. It is worth noting that well-boundedness implies totality; thus, total quasi-orders are useful for the theorems.

The greedy theorems for *min* are useful to obtain one best solution on total quasi-order. If we would like to enumerate all best solutions, or we would like to consider best solutions on non-total quasi-orders, it is better to shift from *min* to *mnl*. Let us introduce greedy theorems for *mnl*.

Theorem 3.13 (minimal-based greedy theorem for catamorphisms)**.** Given a polynomial functor F, a relation $S : \mathsf{F}A \leftrightarrow A$, and well-supported quasi-order R, the following equation holds if S is strictly monotonic on R°.

$$\Lambda mnl_R \circ \Lambda(\!|S|\!) = (\!|\Lambda mnl_R \circ \Lambda(S \circ \mathsf{F}\!\in)|\!)$$

Proof.

$$\begin{array}{ll}
\multicolumn{2}{l}{\Lambda mnl_R \circ \Lambda(\!|S|\!) = (\!|\Lambda mnl_R \circ \Lambda(S \circ \mathsf{F}\!\in)|\!)} \\
\Leftrightarrow & \{ \text{ Eilenberg-Wright (Lemma 2.20) } \} \\
& \Lambda mnl_R \circ (\!|\Lambda(S \circ \mathsf{F}\!\in)|\!) = (\!|\Lambda mnl_R \circ \Lambda(S \circ \mathsf{F}\!\in)|\!) \\
\Leftarrow & \{ \text{ fusion (Theorem 2.19) } \} \\
& \Lambda mnl_R \circ \Lambda(S \circ \mathsf{F}\!\in) = \Lambda mnl_R \circ \Lambda(S \circ \mathsf{F}\!\in) \circ \mathsf{F}\Lambda mnl_R \\
\Leftrightarrow & \{ \mathsf{F}\Lambda mnl_R \text{ is a total function, and the property of total functions (2.2) } \} \\
& \Lambda mnl_R \circ \Lambda(S \circ \mathsf{F}\!\in) = \Lambda mnl_R \circ \Lambda(S \circ \mathsf{F}\!\in \circ\, \mathsf{F}\Lambda mnl_R) \\
\Leftrightarrow & \{ \mathsf{F} \text{ is a functor, and the cancellation of } \Lambda \text{ and } \in \} \\
& \Lambda mnl_R \circ \Lambda(S \circ \mathsf{F}\!\in) = \Lambda mnl_R \circ \Lambda(S \circ \mathsf{F}mnl_R) \\
\Leftarrow & \{ \text{ Lemma 3.28, because } S \circ \mathsf{F}\!\in\, \supseteq S \circ \mathsf{F}mnl_R \text{ and } R \text{ is well-supported } \} \\
& mnl_R \circ \Lambda(S \circ \mathsf{F}\!\in) \subseteq S \circ \mathsf{F}mnl_R \\
\Leftarrow & \{ \text{ Lemma 3.31 } \} \\
& \mathit{True}
\end{array} \qquad \square$$

Theorem 3.14 (minimal-based greedy theorem for repetitions)**.** Given a relation $S : A \leftrightarrow A$ and a well-supported quasi-order $R : A \leftrightarrow A$, the following equation holds if S is strictly monotonic on R°.

$$\Lambda mnl_R \circ \Lambda(S*) = (\Lambda mnl_R \circ \Lambda(S \circ \in))* \circ \{\cdot\}$$

Proof.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\Lambda mnl_R \circ \Lambda(S*) = (\Lambda mnl_R \circ \Lambda(S \circ \in))* \circ \Lambda mnl_R} \\
\Leftrightarrow & \{ \text{ Eilenberg-Wright (Lemma 2.23) } \} \\
 & \Lambda mnl_R \circ (\Lambda(S \circ \in))* \circ \{\cdot\} = (\Lambda mnl_R \circ \Lambda(S \circ \in))* \circ \{\cdot\} \\
\Leftarrow & \{ \ \Lambda mnl_R \circ \{\cdot\} = \{\cdot\} \text{ and fusion (Theorem 2.22) } \} \\
 & \Lambda mnl_R \circ \Lambda(S \circ \in) = \Lambda mnl_R \circ \Lambda(S \circ \in) \circ \Lambda mnl_R \\
\Leftrightarrow & \{ \ \Lambda mnl_R \text{ is a total function, and the property of total functions (2.2) } \} \\
 & \Lambda mnl_R \circ \Lambda(S \circ \in) = \Lambda mnl_R \circ \Lambda(S \circ \in \circ \Lambda mnl_R) \\
\Leftrightarrow & \{ \text{ canceling } \Lambda \text{ and } \in \} \\
 & \Lambda mnl_R \circ \Lambda(S \circ \in) = \Lambda mnl_R \circ \Lambda(S \circ mnl_R) \\
\Leftarrow & \{ \text{ Lemma 3.28, because } S \circ \in \supseteq S \circ mnl_R \text{ and } R \text{ is well-supported } \} \\
 & mnl_R \circ \Lambda(S \circ \in) \subseteq S \circ mnl_R \\
\Leftarrow & \{ \text{ Lemma 3.31 } \} \\
 & \mathit{True}
\end{array}
$$

□

Theorems 3.13 and 3.14 clarify sufficient conditions to enumerate all optimal solutions efficiently. The theorems state that minimal solutions of each step is sufficient to obtain all minimal solutions if strictly monotonicity holds. It is worth noting that the theorems deal with non-total quasi-orders. As the price of enumeration of all solutions, the resulted programs of the greedy theorems for *mnl* are less efficient than those obtained from the greedy theorems for *min*; In each step, the former ones retain all minimal solutions, while the latter ones retain only one minimum solution.

We would like to remark that the theorems for *mnl* are applicable for obtaining not minimal but minimum elements, as the following lemmas show.

Lemma 3.15. For any well-bounded quasi-order R, min_R is equivalent to mnl_R.

Proof. First, notice that well-boundedness implies totality. It is because, if R is not total, there exist elements a and b such that neither $a\,R\,b$ nor $b\,R\,a$ holds, and then $\{a, b\}$ has no minimum element. Therefore, it is sufficient to prove that $R = (R^\circ \Rightarrow R)$ holds if R is total.

$$
\begin{array}{lll}
a\,(R^\circ \Rightarrow R)\,b & \Leftrightarrow & \{ \text{ definition of } \Rightarrow \} \\
 & & b\,\overline{R}\,a \vee a\,R\,b \\
 & \Leftrightarrow & \{ \text{ trivial } (\vee) \ \} \\
 & & (b\,\overline{R}\,a \wedge a\,\overline{R}\,b) \vee a\,R\,b \\
 & \Leftrightarrow & \{ \text{ negation } \} \\
 & & \neg(b\,R\,a \vee a\,R\,b) \vee a\,R\,b \\
 & \Leftrightarrow & \{ \ R \text{ is total } \} \\
 & & a\,R\,b
\end{array}
$$

□

Lemma 3.16. Well-bounded quasi-order is well-supported.

Proof. It is a direct consequence of 3.15. □

3.2.2 Thinning Theorems

It is frequent that the relation to generate candidates does not satisfy monotonicity on the order that we would like to optimize. In such cases, the greedy theorems do not work at all. Even so, there may exist efficient procedure to obtain the best solution, because we may be able to discard apparently useless candidates. Bird and de Moor formalized this intuition as *thinning theorems* [BdM96, Bir01].

A technical difficulty for formalizing the thinning theorem is treatment of non-total orders. The order to discard apparently useless candidates should be weaker than the original order that we would like to optimize, and thus the order may not be total. However, as explained, the greedy theorems for *min* implicitly requires total quasi-orders. The greedy theorems for *mnl* does not require totality, while the resulted programs are a bit inefficient for obtaining one best solution. Bird and de Moor proposed another operator *thin* to deal with non-total quasi-orders. For a quasi-order $R : A \leftrightarrow A$, $\mathit{thin}_R : \mathsf{P}A \leftrightarrow \mathsf{P}A$ is a relation satisfying the following axiomatic property.

$$(Y, X) \in \mathit{thin}_R \overset{\text{def}}{\iff} (Y \subseteq X \wedge \forall b \in X, \exists a \in Y \,:\, a\,R\,b)$$

Intuitively, *thin* discards a part of elements that are larger than another element. Now let us introduce the thinning theorem.

Theorem 3.17 (thinning theorem for catamorphisms [BdM96, Bir01])**.** For any relations R and S, and an quasi-order Q, the following inequality holds provided that $Q \subseteq R$ holds and S is monotonic on Q°.

$$\mathit{min}_R \circ \Lambda(\![S]\!) \supseteq \mathit{min}_R \circ (\![\mathit{thin}_Q \circ \Lambda(S \circ \mathsf{F}{\in})]\!) \qquad \square$$

Theorem 3.18 (thinning theorem for repetitions)**.** For any relations R and S, and an quasi-order Q, the following inequality holds provided that $Q \subseteq R$ holds and S is monotonic on Q°.

$$\mathit{min}_R \circ \Lambda(S*) \supseteq \mathit{min}_R \circ (\mathit{thin}_Q \circ \Lambda(S \circ \in))* \circ \{\cdot\} \qquad \square$$

As similar to the greedy theorems, the statement of thinning theorems is intuitive. We can discard elements that never yield minimums, even when min_R may discard elements that may yield minimums. The disposal of useless elements are performed by thin_Q, and the monotonicity condition of Q guarantees that the disposal is safe.

One important issue is the way to specify the order Q. The order Q does not appear in the specification, and thus, it is necessary to find out Q; besides, Q° must satisfy the monotone property, which makes it difficult to derive an appropriate Q.

Another important issue is the implementation of thin_Q. *thin* is a general operator and its axiom does not specify a concrete implementation. In other words, the efficiency of derived algorithms depends on the implementation of thin_Q. However, it is difficult to provide appropriate implementation for thin_Q, because there are many

functions that satisfy the characterization. For example, all of the identity function, Λmin_R where R is well-bounded quasi-order, and Λmnl_R where R is well-supported quasi-order satisfy the axiom of $thin_R$.

As similar to the case of greedy theorems, we can consider thinning theorems to enumerate all of best solutions as follows. It is worth noting that in the case of enumerating all of best solutions, thinning theorems can be seen as generalizations of greedy theorems for mnl.

Theorem 3.19 (minimal-based thinning theorem for catamorphisms)**.** Given a polynomial functor F, a relation $S : \mathsf{F}A \leftrightarrow A$, and well-supported quasi-order R, the following equation holds if $\overset{Q}{<} \supseteq \overset{R}{<}$ holds and S is strictly monotonic on Q°.

$$\Lambda mnl_R \circ \Lambda(\!|S|\!) = \Lambda mnl_R \circ (\!|\Lambda mnl_Q \circ \Lambda(S \circ \mathsf{F}{\in})|\!)$$

Proof.

$$\begin{aligned}
\Lambda mnl_R \circ \Lambda(\!|S|\!) &= \quad \{ \text{ Lemma 3.29 } \} \\
&\quad \Lambda mnl_R \circ \Lambda mnl_Q \circ \Lambda(\!|S|\!) \\
&= \quad \{ \text{ Theorem 3.13 } \} \\
&\quad \Lambda mnl_R \circ \Lambda mnl_Q \circ (\!|\Lambda mnl_Q \circ \Lambda(S \circ \mathsf{F}{\in})|\!) \\
&= \quad \{ \text{ Lemma 3.29 } \} \\
&\quad \Lambda mnl_R \circ (\!|\Lambda mnl_Q \circ \Lambda(S \circ \mathsf{F}{\in})|\!)
\end{aligned}$$
□

Theorem 3.20 (minimal-based thinning theorem for repetitions)**.** Given a relation $S : A \leftrightarrow A$ and well-supported quasi-order R, the following equation holds if $\overset{Q}{<} \supseteq \overset{R}{<}$ holds and S is strictly monotonic on Q°.

$$\Lambda mnl_R \circ \Lambda(S*) = \Lambda mnl_R \circ (\Lambda mnl_Q \circ \Lambda(S \circ \in))* \circ \{\cdot\}$$

Proof.

$$\begin{aligned}
\Lambda mnl_R \circ \Lambda(S*) &= \quad \{ \text{ Lemma 3.29 } \} \\
&\quad \Lambda mnl_R \circ \Lambda mnl_Q \circ \Lambda(S*) \\
&= \quad \{ \text{ Theorem 3.14 } \} \\
&\quad \Lambda mnl_R \circ \Lambda mnl_Q \circ (\Lambda mnl_Q \circ \Lambda(S \circ \in))* \circ \{\cdot\} \\
&= \quad \{ \text{ Lemma 3.29 } \} \\
&\quad \Lambda mnl_R \circ (\Lambda mnl_Q \circ \Lambda(S \circ \in))* \circ \{\cdot\}
\end{aligned}$$
□

3.2.3 Drawbacks of Greedy Theorems and Thinning Theorems

So far, we have reviewed greedy theorems and thinning theorems. Although these theorems provide a clear formalization of greedy algorithms, they are difficult to use for nonspecialist. The most significant hardship is the step to confirm or derive monotonicity condition. The theorems require monotonicity (or strictly monotonicity) condition, and even checking the condition is not easy.

For example, recall the specification shortest path problems.

$$SP \stackrel{\text{def}}{=} (min_{\leq_w} \circ endWith_t \triangleleft \circ from_s \triangleleft \circ \Lambda(\bigcup_{e \in E} extend_e)*) \; []$$

For applying greedy theorems or thinning theorems, we need to crush the predicates into the order.

$$SP = (min_{R_{s,t}} \circ \Lambda(\textstyle\bigcup_{e \in E} extend_e)*) \; []$$

$$\begin{aligned} & p_1 \; R_{s,t} \; p_2 \\ & \stackrel{\text{def}}{=} \; (endWith_t(p_1) \wedge from_s(p_1) \wedge endWith_t(p_2) \wedge from_s(p_2) \wedge p_1 \leq_w p_2) \vee \\ & \quad\;\; (endWith_t(p_1) \wedge from_s(p_1) \wedge \neg(endWith_t(p_2) \wedge from_s(p_2))) \vee \\ & \quad\;\; (\neg(endWith_t(p_1) \wedge from_s(p_1)) \wedge \neg(endWith_t(p_2) \wedge from_s(p_2)) \wedge p_1 \leq_w p_2) \end{aligned}$$

Then, we would like to check that the relation $\bigcup_{e \in E} extend_e$ is monotonic (or strictly monotonic) on the order $R^{\circ}_{s,t}$. This is quite hard task because of the complicated definition of $R_{s,t}$; moreover, and disappointingly, neither monotonicity condition nor strictly monotonicity condition holds.

Next we try to apply the thinning theorem. For the thinning theorem, we need to prepare a quasi-order on which the relation $\bigcup_{e \in E} extend_e$ is monotonic. Such a quasi-order is difficult to find. In fact, the following order Q_s is appropriate.

$$p_1 \; Q_s \; p_2 \stackrel{\text{def}}{=} from_s(p_1) \wedge (\neg from_s(p_2) \vee (dst(p_1) = dst(p_2) \wedge p_1 \leq_w p_2))$$

The order Q_s satisfies $Q_s \subseteq R_{s,t}$; furthermore, $\bigcup_{e \in E} extend_e$ is monotonic on Q°_s. Thus, we can use the thinning theorem and derive efficient algorithm for shortest path problems. However, even confirming such properties is not easy, much less deriving the order Q_s.

In summary, for systematic development of efficient algorithms, it is indispensable to develop methods for confirming or obtaining monotonicity conditions.

3.3 Solving Maximum Marking Problems

Some studies [ALS91, BPT92, SHTO00] show a unified algorithm for a class of combinatorial optimization problems, called *maximum marking problems*. Maximum marking problems are problems to find a marking of an underlying structure that has the maximum sum of marked elements. Marking should satisfy some requirements, which make the problem hard. Bird [Bir01] clarified the relationship between the result about maximum marking problems and thinning theorems. We review these studies in this section.

3.3.1 Deriving Linear-Time Algorithms by Specifying Requirements for Markings

First, we would like to formalize maximum marking problems. Let S_A be the type of the underlying structure that retains elements of type A. Let $M_A = A + A$ be

the type of markings for an element of type A: for an element $a \in A$, $\mathsf{L}(a)$ and $\mathsf{R}(a)$ respectively denote marked a and not marked a. Let $\mathit{marking}_S : S_{\mathbb{R}} \leftrightarrow S_{M_{\mathbb{R}}}$ be the relation that give a marking on the structure S, and $\mathit{wsum} : S_{M_{\mathbb{R}}} \to \mathbb{R}$ be the function that sums up all marked elements in the structure. Now, given a constraint $\mathit{constraint} : S_{M_{\mathbb{R}}} \to \mathit{Bool}$, a maximum marking problem is formalized as follows.

$$\mathit{min}_{\geq_{\mathit{wsum}}} \circ \mathit{constraint}\triangleleft \circ \Lambda \mathit{marking}_S$$

Sasano et al. [SHTO00] showed a unified algorithm for a class of maximum marking problems. Here, we consider a simple case, maximum marking problems on sequences. The followings are the definitions of functions $\mathit{marking} : \mathbb{R}^* \leftrightarrow M_{\mathbb{R}}^*$ and $\mathit{wsum} : M_{\mathbb{R}}^* \to \mathbb{R}$ for this case.

$$\begin{array}{lcl}
([a'_0, \ldots, a'_n], [a_0, \ldots, a_n]) \in \mathit{marking} & \stackrel{\text{def}}{\Leftrightarrow} & \forall 0 \leq i \leq n : \ a'_i \in \{\mathsf{L}(a_i), \mathsf{R}(a_i)\} \\
\mathit{wsum}([\,]) & \stackrel{\text{def}}{=} & 0 \\
\mathit{wsum}([\mathsf{L}(a)] \mathbin{+\!\!+} x) & \stackrel{\text{def}}{=} & a + \mathit{wsum}(x) \\
\mathit{wsum}([\mathsf{R}(a)] \mathbin{+\!\!+} x) & \stackrel{\text{def}}{=} & \mathit{wsum}(x)
\end{array}$$

Then, each predicate $\mathit{constraint} : \mathbb{R} \to \mathit{Bool}$ specifies an instance of maximum marking problems.

The idea of Sasano et al. is to consider the predicate *constraint* of a specific form. As seen in the previous section, existence of the requirement $\mathit{constraint}\triangleleft$ makes problems difficult, and thus, the problem will be solved in ease when *constraint* satisfies a good property. Sasano et al. considered the case in which *constraint* is written by foldr.

Theorem 3.21 (optimization theorem on sequences [SHTO00])**.** The following equation holds.

$$\begin{array}{l}
\mathit{min}_{\geq_{\mathit{wsum}}} \circ (\mathit{accept} \circ \mathsf{foldr}_{\oplus,e})\triangleleft \circ \Lambda \mathit{marking} \\
\quad = \ \mathit{min}_{\geq_{\mathit{wsum}}} \circ (\mathit{accept} \circ \mathsf{foldr}_{\oplus,e})\triangleleft \circ \mathit{maximals}
\end{array}$$

$$\begin{array}{lcl}
\mathit{maximals}([\,]) & \stackrel{\text{def}}{=} & \{[\,]\} \\
\mathit{maximals}([a] \mathbin{+\!\!+} x) & \stackrel{\text{def}}{=} & \Lambda \mathit{mnl}_R(\{[a'] \mathbin{+\!\!+} y \mid a' \in \{\mathsf{L}(a), \mathsf{R}(a)\} \wedge y \in \mathit{maximals}(x)\}) \\
a \, R \, b \stackrel{\text{def}}{=} a \geq_{\mathit{wsum}} b \wedge \mathsf{foldr}_{\oplus,e}(a) = \mathsf{foldr}_{\oplus,e}(b) & &
\end{array}$$

□

The function *maximals* generates only a small set of markings, because most of the candidates will be discarded by mnl_R in each recursion step; thus, the resulted program is efficient. In addition, if the range of $\mathsf{foldr}_{\oplus,e}$ is finite, the derived algorithm yields the optimal result in time linear to the length of the underlying sequence, because only constant number of candidates are considered in each step[3]. The

[3]Correctly, we need to assume that no two markings have the same weight for guaranteeing that the resulted algorithm is a linear-time algorithm. It is worth remarking that even if this assumption does not holds, we can obtain one optimal solution in time linear to the length of the sequence. Let a quasi-order R' be an order that compares two markings being equivalent on R by a lexicographic ordering. Then, as similar to R, R' satisfies monotonicity condition, and we can obtain a part of the optimal solutions by using R'.

strong point of Theorem 3.21 is that it is applicable without confirming any premise once the problem is specified in the form.

As a concrete example, let us solve the *maximum segment sum problem* [Ben86, Bir89] by Theorem 3.21. The problem is problem to find the segment that has the maximum sum, and here a segment is a consecutive subsequence including an empty sequence. For example, $mss([-3,2,4,-5,1,6,-8,3]) = 2+4+(-5)+1+6 = 8$, where mss is the function that solves the maximum segment sum problem. The specification of mss in the style of maximum marking problems is the following.

$$mss \stackrel{\text{def}}{=} wsum \circ min_{\geq_{wsum}} \circ isSeg\triangleleft \circ \Lambda marking$$

$$\begin{array}{lcl}
isSeg([\,]) & \stackrel{\text{def}}{=} & True \\
isSeg([\mathsf{L}(a)] \mathbin{+\!\!+} x) & \stackrel{\text{def}}{=} & isInitSeg(x) \\
isSeg([\mathsf{R}(a)] \mathbin{+\!\!+} x) & \stackrel{\text{def}}{=} & isSeg(x) \\
isInitSeg([\,]) & \stackrel{\text{def}}{=} & True \\
isInitSeg([\mathsf{L}(a)] \mathbin{+\!\!+} x) & \stackrel{\text{def}}{=} & isInitSeg(x) \\
isInitSeg([\mathsf{R}(a)] \mathbin{+\!\!+} x) & \stackrel{\text{def}}{=} & noMark(x) \\
noMark([\,]) & \stackrel{\text{def}}{=} & True \\
noMark([\mathsf{L}(a)] \mathbin{+\!\!+} x) & \stackrel{\text{def}}{=} & False \\
noMark([\mathsf{R}(a)] \mathbin{+\!\!+} x) & \stackrel{\text{def}}{=} & noMark(x)
\end{array}$$

The specification certainly requires that the marked elements should form a segment.

For use of Theorem 3.21, we should specify the predicate $isSeg$ in terms of foldr. For this purpose, *tupling transformations* [Fok89, Chi93, HITT97] is effective. Let us consider a function $g(x) \stackrel{\text{def}}{=} (isSeg(x), isInitSeg(x), noMark(x))$ that computes all of $isSeg$, $isInitSeg$, and $noMark$ in the same time. Then, it is straightforward to specify the function g by a foldr.

$$\begin{array}{l}
g = \mathsf{foldr}_{\oplus,e} \\
\quad e \stackrel{\text{def}}{=} (True, True, True) \\
\quad \mathsf{L}(a) \oplus (p,q,r) \stackrel{\text{def}}{=} (q,q,False) \\
\quad \mathsf{R}(a) \oplus (p,q,r) \stackrel{\text{def}}{=} (p,r,r)
\end{array}$$

Now the maximum segment sum problem is specified in the form that Theorem 3.21 is applicable.

$$mss = wsum \circ min_{\geq_{wsum}} \circ (\pi_1 \circ \mathsf{foldr}_{\oplus,e})\triangleleft \circ \Lambda marking$$

Then, Theorem 3.21 immediately derives a linear-time algorithm. The derived program exactly corresponds to the efficient program introduced by Bentley [Ben86], when we apply an efficiency improvement method proposed by Matsuzaki [Mat07b].

So far, we have considered maximum marking problems on a sequence of numbers. Sasano et al. [SHTO00] proved that we could generalize the result to polynomial data structures. Maximum marking problems on polynomial data structures

can be solved immediately once the constraint is specified in terms of a catamorphism.

Arnborg et al. [ALS91] and Borie et al. [BPT92] independently showed similar results. Any maximum marking problem is solvable in time linear to the size of the underlying structure, whenever the underlying structure is a tree-decomposable graph and the constraint is expressed by a monadic second order logic formula. The point is that we can translate these problems into equivalent maximum marking problems, where the underlying structure is a tree and the constraint can be checked by a tree automaton. Since representative relations of tree automata are catamorphisms of finite ranges, their results are quite similar to that of Sasano et al.

3.3.2 Drawbacks and Relationship to Monotonicity

These results about maximum marking problems are interesting and useful. We do not need to struggle for obtaining monotonicity conditions once the problem is specified in the form. However, their drawback is the lack of generality. They can deal with, essentially, only the maximum marking problems on a tree. For example, they cannot deal with problems on graphs, such as the shortest path problems.

Bird [Bir01] showed a relationship between these results and the thinning theorem. Rewrite the relation *marking* by a catamorphism on the category *Rel*.

$$\begin{aligned} \mathit{marking} &= (\![(\mathit{const}_{[]} \triangledown \mathit{mark})]\!) \\ \mathit{mark} &\stackrel{\text{def}}{=} \{([a'] +\!\!+ x, (a, x)) \mid a' \in \{\mathsf{L}(a), \mathsf{R}(a)\}\} \end{aligned}$$

The key observation is that the relation $(\mathit{const}_{[]} \triangledown \mathit{mark})$ is monotonic on the order $\geq_{wsum} \cap =_{\mathsf{foldr}_{\oplus,e}}$ for any $\oplus$ and e. Therefore, the thinning theorem enables us to discard unnecessary candidates, which yields Theorem 3.21.

This result seems helpful to extend Theorem 3.21 so as to cope with a more general class of problems. However, Bird showed nothing about the extension of Theorem 3.21.

3.4 Supplemental Lemmas

Lemma 3.22 (Inequality (4.6) of [BdM96])**.**

$$(R \circ S) \cap T \subseteq R \circ (S \cap (R^\circ \circ T)) \qquad \square$$

Lemma 3.23 (Equation (4.22) in [BdM96])**.**

$$\overline{R}/Y = \overline{R \circ Y^\circ} \qquad \square$$

Lemma 3.24 (Equation (7.5) of [BdM96])**.**

$$\mathit{min}_R \circ \Lambda S = (S \cap R/S^\circ) \qquad \square$$

Lemma 3.25 (Exercise 7.10 of [BdM96])**.** For reflexive relations R and S, $min_R \subseteq min_S$ holds if and only if $R \subseteq S$ holds.

Proof. The "if" part is trivial. We prove the "only if" part.

$$
\begin{array}{lll}
min_R \subseteq min_S & \Leftrightarrow & \{ \text{ power transpose } \} \\
 & & \forall X : \Lambda min_R(X) \subseteq \Lambda min_S(X) \\
 & \Rightarrow & \{ \text{ let } X = \{a, b\} \text{ such that } a\,R\,b \text{ holds } \} \\
 & & \forall a, b : a\,R\,b \Rightarrow (\Lambda min_R(\{a, b\}) \subseteq \Lambda min_S(\{a, b\})) \\
 & \Rightarrow & \{ \text{ definition of } min \} \\
 & & \forall a, b : a\,R\,b \Rightarrow (a \in \Lambda min_S(\{a, b\})) \\
 & \Rightarrow & \{ \text{ definition of } min \} \\
 & & \forall a, b : a\,R\,b \Rightarrow a\,S\,b
\end{array}
$$

□

Lemma 3.26 (Exercise 7.32 in [BdM96])**.** If R is a well-supported quasi-order, then the following equation holds.

$$mnl_R \circ \Lambda(X \cup Y) = mnl_R \circ \Lambda(mnl_R \circ \Lambda X \cup mnl_R \circ \Lambda Y)$$

Proof. $mnl_R \circ \Lambda(X \cup Y) \subseteq mnl_R \circ \Lambda(mnl_R \circ \Lambda X \cup mnl_R \circ \Lambda Y)$ (inequality (7.11) in [BdM96]) is easy to prove. We would like to prove the latter half. Here let $P \stackrel{\text{def}}{=} R^\circ \Rightarrow R$ and $S \stackrel{\text{def}}{=} (mnl_R \circ \Lambda X \cup mnl_R \circ \Lambda Y)$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{mnl_R \circ \Lambda S \subseteq mnl_R \circ \Lambda(X \cup Y)} \\
\Leftrightarrow & \{ \text{ Lemma 3.24 } \} \\
 & (S \cap P/S^\circ) \subseteq ((X \cup Y) \cap P/(X \cup Y)^\circ) \\
\Leftarrow & \{ \text{ trivial (intersection) } \} \\
 & (S \subseteq (X \cup Y) \wedge (P/S^\circ \subseteq P/(X \cup Y)^\circ) \\
\Leftrightarrow & \{ (mnl_R \circ \Lambda X \cup mnl_R \circ \Lambda Y) \subseteq (X \cup Y) \} \\
 & P/S^\circ \subseteq P/(X \cup Y)^\circ \\
\Leftrightarrow & \{ \text{ property of } / \} \\
 & P/S^\circ \circ (X \cup Y)^\circ \subseteq P \\
\Leftarrow & \{ \text{ well-supportedness } \} \\
 & P/S^\circ \circ (R^\circ \circ mnl_R \circ \Lambda(X \cup Y))^\circ \subseteq P \\
\Leftarrow & \{ mnl_R \circ \Lambda(X \cup Y) \subseteq (mnl_R \circ \Lambda X \cup mnl_R \circ \Lambda Y) = S \} \\
 & P/S^\circ \circ (R^\circ \circ S)^\circ \subseteq P \\
\Leftarrow & \{ \text{ right-division } \} \\
 & P \circ R \subseteq P \\
\Leftrightarrow & \{ \text{ claim (proved in the following) } \} \\
 & \mathit{True}
\end{array}
$$

The claim is proved as follows.

$$\begin{array}{rcl}
c\,(R^{\circ} \Rightarrow R)\,b \wedge b\,R\,a & \Leftrightarrow & \{\text{ definition of } \Rightarrow \}\\
& & (\neg(b\,R\,c) \vee c\,R\,b) \wedge b\,R\,a\\
& \Leftrightarrow & \{\text{ distributing } \wedge \text{ over } \vee \}\\
& & (\neg(b\,R\,c) \wedge b\,R\,a) \vee (c\,R\,b \wedge b\,R\,a)\\
& \Rightarrow & \{\text{ transitivity }\}\\
& & \neg(a\,R\,c) \vee (c\,R\,a)\\
& \Leftrightarrow & \{\text{ definition of } \Rightarrow \}\\
& & c\,(R^{\circ} \Rightarrow R)\,a
\end{array}$$

□

Lemma 3.27.

$$(R^{\circ} \circ X/Y \subseteq S) \Leftrightarrow (R \circ \overline{S} \subseteq \overline{X} \circ Y^{\circ})$$

Proof.

$$\begin{array}{rcl}
R^{\circ} \circ X/Y \subseteq S & \Leftrightarrow & \{\text{ converse }\}\\
& & (X/Y)^{\circ} \circ R \subseteq S^{\circ}\\
& \Leftrightarrow & \{\text{ right division }\}\\
& & (X/Y)^{\circ} \subseteq S^{\circ}/R\\
& \Leftrightarrow & \{\text{ double negation }\}\\
& & (\overline{\overline{X}}/Y)^{\circ} \subseteq \overline{\overline{S^{\circ}}}/R\\
& \Leftrightarrow & \{\text{ Lemma 3.23 }\}\\
& & (\overline{\overline{X} \circ Y^{\circ}})^{\circ} \subseteq \overline{(\overline{S^{\circ}} \circ R^{\circ})}\\
& \Leftrightarrow & \{\text{ negation and converse }\}\\
& & \overline{X} \circ Y^{\circ} \supseteq R \circ \overline{S}
\end{array}$$

□

Lemma 3.28. Given a well-supported quasi-order R, $X \supseteq Y \supseteq \Lambda mnl_R(X)$ implies $\Lambda mnl_R(X) = \Lambda mnl_R(Y)$.

Proof.

$$\begin{array}{rcl}
\Lambda mnl_R(Y) & = & \{\ Y \supseteq \Lambda mnl_R(X)\ \}\\
& & \Lambda mnl_R(\Lambda mnl_R(X) \cup Y)\\
& = & \{\text{ Lemma 3.26 }\}\\
& & \Lambda mnl_R(\Lambda mnl_R(\Lambda mnl_R(X)) \cup \Lambda mnl_R(Y))\\
& = & \{\ \Lambda mnl_R \text{ is idempotent }\}\\
& & \Lambda mnl_R(\Lambda mnl_R(X) \cup \Lambda mnl_R(Y))\\
& = & \{\text{ Lemma 3.26 }\}\\
& & \Lambda mnl_R(X \cup Y)\\
& = & \{\ X \supseteq Y\ \}\\
& & \Lambda mnl_R(X)
\end{array}$$

□

Lemma 3.29. For any well-supported quasi-order R and S, $\stackrel{R}{<} \supseteq \stackrel{S}{<}$ implies $\Lambda mnl_R \circ \Lambda mnl_S = \Lambda mnl_R$.

Proof. From Lemma 3.28 and the fact $\Lambda mnl_S(X) \subseteq X$, it is sufficient to prove $\Lambda mnl_R \subseteq \Lambda mnl_S$. From Lemma 3.25, it is sufficient to show that $R^{\circ} \Rightarrow R$ is

reflexive for any relation R and $\overset{R}{<} \supseteq \overset{S}{<}$ is equivalent to $(R^\circ \Rightarrow R) \subseteq (S^\circ \Rightarrow S)$. First, let us prove the former claim, namely the reflexivity of $R^\circ \Rightarrow R$.

$$
\begin{array}{rcl}
(R^\circ \Rightarrow R) \supseteq id & \Leftrightarrow & \{ \text{ axiom of } \Rightarrow \} \\
& & R \supseteq (R^\circ \cap id) \\
& \Leftrightarrow & \{ \text{ identity } \} \\
& & R \supseteq (R^\circ \cap id^\circ) \\
& \Leftrightarrow & \{ \text{ distributivity of converses over intersections } \} \\
& & R \supseteq (R \cap id)^\circ \\
& \Leftrightarrow & \{ R \cap id \subseteq id \} \\
& & R \supseteq (R \cap id) \\
& \Leftrightarrow & \{ \text{ intersection } \} \\
& & \mathit{True}
\end{array}
$$

The latter one is proved as follows.

$$
\begin{array}{rcl}
\overset{R}{<} \supseteq \overset{S}{<} & \Leftrightarrow & \{ \text{ definition of } \overset{R}{<} \text{ and } \overset{S}{<} \} \\
& & (R \cap \overline{R^\circ}) \supseteq (S \cap \overline{S^\circ}) \\
& \Leftrightarrow & \{ \text{ negation } \} \\
& & \overline{(R \cap \overline{R^\circ})} \subseteq \overline{(S \cap \overline{S^\circ})} \\
& \Leftrightarrow & \{ \text{ distributivity of negations over conjunctions } \} \\
& & (\overline{R} \cup \overline{\overline{R^\circ}}) \subseteq (\overline{S} \cup \overline{\overline{S^\circ}}) \\
& \Leftrightarrow & \{ \text{ double-negation elimination } \} \\
& & (\overline{R} \cup R^\circ) \subseteq (\overline{S} \cup S^\circ) \\
& \Leftrightarrow & \{ \text{ converse } \} \\
& & (\overline{R} \cup R^\circ)^\circ \subseteq (\overline{S} \cup S^\circ)^\circ \\
& \Leftrightarrow & \{ \text{ distributivity of converses over disjunctions and negations } \} \\
& & (\overline{R^\circ} \cup R) \subseteq (\overline{S^\circ} \cup S) \\
& \Leftrightarrow & \{ \text{ definition of implications } \} \\
& & (R^\circ \Rightarrow R) \subseteq (S^\circ \Rightarrow S)
\end{array}
$$

□

Lemma 3.30. For any polynomial functor F and a reflective relation R, $\mathsf{F}mnl_R = mnl_{\mathsf{F}R} \circ \Lambda\mathsf{F}\in$ holds.

Proof. First we reason as follows.

$$
\begin{array}{rcl}
\mathsf{F}mnl_R = mnl_{\mathsf{F}R} \circ \Lambda\mathsf{F}\in & \Leftrightarrow & \{ \text{ definition of } mnl \text{ and Lemma 3.24 } \} \\
& & \mathsf{F}(\in \cap (R^\circ \Rightarrow R)/\ni) = (\mathsf{F}\in \cap (\mathsf{F}R^\circ \Rightarrow \mathsf{F}R)/\mathsf{F}\ni) \\
& \Leftrightarrow & \{ \text{ Lemma 2.11 } \} \\
& & (\mathsf{F}\in \cap \mathsf{F}(R^\circ \Rightarrow R)/\mathsf{F}\ni) = (\mathsf{F}\in \cap (\mathsf{F}R^\circ \Rightarrow \mathsf{F}R)/\mathsf{F}\ni)
\end{array}
$$

Now, from Lemma 2.9, it is sufficient to prove $\mathsf{F}(R^\circ \Rightarrow R)/\mathsf{F}\ni \supseteq (\mathsf{F}R^\circ \Rightarrow \mathsf{F}R)/\mathsf{F}\ni$, which is equivalent to $\mathsf{F}(R^\circ \Rightarrow R) \supseteq (\mathsf{F}R^\circ \Rightarrow \mathsf{F}R)/\mathsf{F}\ni \circ \mathsf{F}\ni$. We prove it by induction on the functor. When the functor F is the identity functor or a constant functor,

the claim apparently holds. The case of sum, namely when $\mathsf{F} = \mathsf{G} + \mathsf{H}$, is also easy to prove. We reason as follows.

$$
\begin{array}{ll}
& \mathsf{L}(a)\ (((\mathsf{G}+\mathsf{H})R^\circ \Rightarrow (\mathsf{G}+\mathsf{H})R)/(\mathsf{G}+\mathsf{H})\ni \circ (\mathsf{G}+\mathsf{H})\ni)\ \mathsf{L}(b) \\
\Leftrightarrow & \{ \text{ definition of tagged sum } \} \\
& \mathsf{L}(a)\ ((\mathsf{G}R^\circ \Rightarrow \mathsf{G}R)/\mathsf{G}\ni \circ \mathsf{G}\ni)\ \mathsf{L}(b) \\
\Rightarrow & \{ \text{ induction hypothesis } \} \\
& \mathsf{L}(a)\ \mathsf{G}(R^\circ \Rightarrow R)\ \mathsf{L}(b) \\
\Leftrightarrow & \{ \text{ definition of tagged sum } \} \\
& \mathsf{L}(a)\ ((\mathsf{G}+\mathsf{H})(R^\circ \Rightarrow R))\ \mathsf{L}(b)
\end{array}
$$

And by reasoning the other case in the same way, we can prove the case of the sum. The only nontrivial case is the case of product, namely when $\mathsf{F} = \mathsf{G} \times \mathsf{H}$, which is proved as follows.

$$
\begin{array}{ll}
& (a_1, a_2)\ (((\mathsf{G}\times\mathsf{H})R^\circ \Rightarrow (\mathsf{G}\times\mathsf{H})R)/(\mathsf{G}\times\mathsf{H})\ni \circ (\mathsf{G}\times\mathsf{H})\ni)\ (b_1, b_2) \\
\Leftrightarrow & \{ \text{ definition of product } \} \\
& \exists X_1, X_2 :\ a_1\ \mathsf{G}{\in}\ X_1 \wedge b_1\ \mathsf{G}{\in}\ X_1 \wedge a_2\ \mathsf{H}{\in}\ X_2 \wedge b_2\ \mathsf{H}{\in}\ X_2 \wedge \\
& (\forall c_1\ \mathsf{G}{\in}\ X_1, c_2\ \mathsf{H}{\in}\ X_2 :\ (c_1\ \mathsf{G}R\ a_1 \wedge c_2\ \mathsf{H}R\ a_2) \Rightarrow (a_1\ \mathsf{G}R\ c_1 \wedge a_2\ \mathsf{H}R\ c_2)) \\
\Rightarrow & \{ \text{ instantiating } \forall \text{ quantified variables } \} \\
& \exists X_1, X_2 :\ a_1\ \mathsf{G}{\in}\ X_1 \wedge b_1\ \mathsf{G}{\in}\ X_1 \wedge a_2\ \mathsf{H}{\in}\ X_2 \wedge b_2\ \mathsf{H}{\in}\ X_2 \wedge \\
& (\forall c_1\ \mathsf{G}{\in}\ X_1 :\ (c_1\ \mathsf{G}R\ a_1 \wedge a_2\ \mathsf{H}R\ a_2) \Rightarrow (a_1\ \mathsf{G}R\ c_1 \wedge a_2\ \mathsf{H}R\ a_2)) \wedge \\
& (\forall c_2\ \mathsf{H}{\in}\ X_2 :\ (a_1\ \mathsf{G}R\ a_1 \wedge c_2\ \mathsf{H}R\ a_2) \Rightarrow (a_1\ \mathsf{G}R\ a_1 \wedge a_2\ \mathsf{H}R\ c_2)) \\
\Leftrightarrow & \{ \text{ simplify (reflectivity) } \} \\
& \exists X_1, X_2 :\ a_1\ \mathsf{G}{\in}\ X_1 \wedge b_1\ \mathsf{G}{\in}\ X_1 \wedge a_2\ \mathsf{H}{\in}\ X_2 \wedge b_2\ \mathsf{H}{\in}\ X_2 \wedge \\
& (\forall c_1\ \mathsf{G}{\in}\ X_1 :\ c_1\ \mathsf{G}R\ a_1 \Rightarrow a_1\ \mathsf{G}R\ c_1) \wedge (\forall c_2\ \mathsf{H}{\in}\ X_2 :\ c_2\ \mathsf{H}R\ a_2 \Rightarrow a_2\ \mathsf{H}R\ c_2) \\
\Leftrightarrow & \{ \text{ simplify (quantifier) } \} \\
& (\exists X_1 :\ a_1\ \mathsf{G}{\in}\ X_1 \wedge b_1\ \mathsf{G}{\in}\ X_1 \wedge (\forall c_1\ \mathsf{G}{\in}\ X_1 :\ c_1\ \mathsf{G}R\ a_1 \Rightarrow a_1\ \mathsf{G}R\ c_1)) \wedge \\
& (\exists X_2 :\ a_2\ \mathsf{H}{\in}\ X_2 \wedge b_2\ \mathsf{H}{\in}\ X_2 \wedge (\forall c_2\ \mathsf{H}{\in}\ X_2 :\ c_2\ \mathsf{H}R\ a_2 \Rightarrow a_2\ \mathsf{H}R\ c_2)) \\
\Leftrightarrow & \{ \text{ definition of product } \} \\
& (a_1, a_2)\ (((\mathsf{G}R^\circ \Rightarrow \mathsf{G}R)/\mathsf{G}\ni \circ \mathsf{G}\ni) \times ((\mathsf{H}R^\circ \Rightarrow \mathsf{H}R)/\mathsf{H}\ni \circ \mathsf{H}\ni))\ (b_1, b_2) \\
\Rightarrow & \{ \text{ induction hypothesis } \} \\
& (a_1, a_2)\ (\mathsf{G}(R^\circ \Rightarrow R) \times \mathsf{H}(R^\circ \Rightarrow R))\ (b_1, b_2) \\
\Leftrightarrow & \{ \text{ definition of product } \} \\
& (a_1, a_2)\ ((\mathsf{G}\times\mathsf{H})(R^\circ \Rightarrow R))\ (b_1, b_2)
\end{array}
$$

□

Lemma 3.31. Given a reflective relation $R : A \leftrightarrow A$ and a relation $S : \mathsf{F}A \leftrightarrow A$, where F is a polynomial functor, the following inequality holds, provided that S is strictly monotonic on R°.

$$mnl_R \circ \Lambda(S \circ \mathsf{F}{\in}) \subseteq S \circ \mathsf{F}mnl_R$$

Proof.

$$
\begin{array}{ll}
& mnl_R \circ \Lambda(S \circ \mathsf{F}{\in}) \subseteq S \circ \mathsf{F}mnl_R \\
\Leftrightarrow & \{ \text{ Lemma 3.30 } \} \\
& mnl_R \circ \Lambda(S \circ \mathsf{F}{\in}) \subseteq S \circ mnl_{\mathsf{F}R} \circ \Lambda\mathsf{F}{\in} \\
\Leftrightarrow & \{ \text{ definition of } mnl \text{ and Lemma 3.24 } \} \\
& (S \circ \mathsf{F}{\in}) \cap (R^\circ \Rightarrow R)/(\mathsf{F}{\ni} \circ S^\circ) \subseteq S \circ (\mathsf{F}{\in} \cap (\mathsf{F}R^\circ \Rightarrow \mathsf{F}R)/\mathsf{F}{\ni}) \\
\Leftarrow & \{ \text{ Lemma 3.22 } \} \\
& S \circ (\mathsf{F}{\in} \cap S^\circ \circ (R^\circ \Rightarrow R)/(\mathsf{F}{\ni} \circ S^\circ)) \subseteq S \circ (\mathsf{F}{\in} \cap (\mathsf{F}R^\circ \Rightarrow \mathsf{F}R)/\mathsf{F}{\ni}) \\
\Leftarrow & \{ \text{ trivial } \} \\
& S^\circ \circ (R^\circ \Rightarrow R)/(\mathsf{F}{\ni} \circ S^\circ) \subseteq (\mathsf{F}R^\circ \Rightarrow \mathsf{F}R)/\mathsf{F}{\ni} \\
\Leftarrow & \{ \text{ right division } \} \\
& S^\circ \circ (R^\circ \Rightarrow R)/S^\circ \subseteq (\mathsf{F}R^\circ \Rightarrow \mathsf{F}R) \\
\Leftrightarrow & \{ \text{ Lemma 3.27 } \} \\
& S \circ \overline{(\mathsf{F}R^\circ \Rightarrow \mathsf{F}R)} \subseteq \overline{(R^\circ \Rightarrow R)} \circ S \\
\Leftrightarrow & \{ \text{ definition of } \overset{R^\circ}{<} \} \\
& S \circ \overset{\mathsf{F}R^\circ}{<} \subseteq \overset{R^\circ}{<} \circ S \\
\Leftrightarrow & \{ \text{ premise: strictly monotonicity } \} \\
& \mathit{True}
\end{array}
$$

□

Chapter 4

Compositional Approach to Monotonicity

As seen in Chapter 3, monotonicity condition is a key to efficient algorithms of combinatorial optimization problems, though confirming or obtaining the condition is hard in practice. Our goal is to provide effective methods for such situations.

In this chapter, we develop calculational laws for obtaining and confirming monotonicity conditions. Our idea is to examine structures of problems more carefully. We examine how candidates are generated, on what order we would like to optimize, and what condition solutions should satisfy; then, based on the structures, we construct orders that satisfy monotonicity conditions. As a summary of our laws, we propose calculational laws that are useful to derive dynamic programming algorithms. For demonstrating effectiveness of our method, we show derivations of efficient algorithms for several problems, including shortest path problems and their variants. Our calculational laws enable us to derive efficient algorithms in ease, even for graph-iterating problems. Supplemental lemmas are shown in the last section, Section 4.4, with their proofs.

Before introducing our calculational laws, we would like to introduce a notion that is useful to formalize our laws.

Definition 4.1 (collapsing)**.** For functions $f : A \to B$ and $g : A \to C$, f is said to be *more collapsing* than g if $g(a_1) = g(a_2)$ implies $f(a_1) = f(a_2)$ for any $a_1, a_2 \in A$. $\square$

In usual, a function $f : A \to B$ maps each element of A to an element of B, with losing some information on A. We have introduced the notion of collapsing so as to compare how much information is lost by applying a function. When function f is more collapsing than another function g, then f loses more information than g; in other words, information remained after applying f can be retrieved from information remained after applying g.

There are some equivalent definitions.

This chapter corresponds to a revised version of [MMHT07].

Lemma 4.2. For total functions f and g, the following statements are equivalent.

1. f is more collapsing than g.
2. $=_f \supseteq =_g$.
3. There exists a (possibly partial) function f' such that $f = f' \circ g$.

Proof. The second statement is just a rephrase of the first one. It is easy to see the third statement implies the second one: $g(a_1) = g(a_2)$ implies $f(a_1) = f'(g(a_1)) = f'(g(a_2)) = f(a_2)$. In the following, we prove the second statement implies the third one. Let $f' = f \circ g^\circ$. Then f' is simple, namely $(f \circ g^\circ) \circ (f \circ g^\circ)^\circ \subseteq id$ holds, as the following calculation proves.

$$\begin{array}{ll} & (f \circ g^\circ) \circ (f \circ g^\circ)^\circ \subseteq id \\ \Leftrightarrow & \{\text{ distributing a converse over the composition }\} \\ & f \circ g^\circ \circ g \circ f^\circ \subseteq id \\ \Leftrightarrow & \{\text{ properties of total functions (2.3) and (2.4) }\} \\ & g^\circ \circ g \subseteq f^\circ \circ f \\ \Leftrightarrow & \{\ (f^\circ \circ f) = =_f\ \} \\ & =_g \subseteq =_f \end{array}$$

Now it is sufficient to prove $f = f' \circ g$. $f \subseteq f' \circ g$ evidently holds from the property of total functions (2.4). The following calculation proves $f \supseteq f' \circ g$.

$$\begin{array}{lll} f & = & \{\ h = h \circ h^\circ \circ h \text{ holds for any function } h\ \} \\ & & f \circ f^\circ \circ f \\ & \supseteq & \{\ =_f \supseteq =_g \text{ and } =_f = f^\circ \circ f\ \} \\ & & f \circ g^\circ \circ g \\ & = & \{\text{ definition of } f'\ \} \\ & & f' \circ g \end{array}$$

□

4.1 Constructing Monotonicity Conditions

In this section, we would like to discuss the way to confirm or obtain monotonicity conditions. We adopt a constructive approach. We confirm or obtain monotonicity conditions for complicated ones by combining simple ones.

4.1.1 Specifying Candidate Generators as Unions of Functions

First, we would like to make ease to confirm monotonicity condition. Monotonicity condition of relations forms a statement like "for all something, exists something, such that", which is hard to reason. In contrast to it, monotonicity condition of total functions is easy to reason.

Lemma 4.3. For a total function $f : \mathsf{F}A \to A$ and a relation $R : A \leftrightarrow A$, f is monotonic on R if and only if $R_f \supseteq \mathsf{F}R$ holds.

Proof.

$$
\begin{array}{rcl}
R \circ f \supseteq f \circ \mathsf{F}R & \Leftrightarrow & \{ \text{ property of total functions (2.3) } \} \\
& & f^\circ \circ R \circ f \supseteq \mathsf{F}R \\
& \Leftrightarrow & \{ \text{ definition of } R_f \ \} \\
& & R_f \supseteq \mathsf{F}R
\end{array}
$$

□

Lemma 4.4. For a total function $f : \mathsf{F}A \to A$ and a relation $R : A \leftrightarrow A$, f is strictly monotonic on R if and only if $(\overset{R}{<})_f \supseteq \overset{\mathsf{F}R}{<}$ holds.

Proof. The proof is almost the same as that of Lemma 4.3. □

Lemma 4.5. For a total function $f : \mathsf{F}A \to A$ and a relation $R : A \leftrightarrow A$, where F is a relator, f is monotonic (strictly monotonic, completely monotonic) on R if and only if f is monotonic (strictly monotonic, completely monotonic) on R°.

Proof. We prove the case of monotonicity. Others are similar.

$$
\begin{array}{rcl}
R \circ f \supseteq f \circ \mathsf{F}R & \Leftrightarrow & \{ \text{ converse } \} \\
& & f^\circ \circ R^\circ \supseteq (\mathsf{F}R)^\circ \circ f^\circ \\
& \Leftrightarrow & \{ \ \mathsf{F} \text{ is a relator } \} \\
& & f^\circ \circ R^\circ \supseteq \mathsf{F}R^\circ \circ f^\circ \\
& \Leftrightarrow & \{ \text{ properties of total functions (2.3) and (2.4) } \} \\
& & R^\circ \circ f \supseteq f \circ \mathsf{F}R^\circ
\end{array}
$$

□

As Lemmas 4.3, 4.4, and 4.5 show, use of total functions eases the difficulty to confirm monotonicity. Thus, we would like to use total functions instead of relations as much as possible. For this purpose, we decompose a relation in two steps. We first decompose a relation into a union of partial functions, and then, we decompose each partial functions into a total function with a filter. Such decompositions make it easy to confirm monotonicity conditions.

Lemma 4.6. Given two relations $S : \mathsf{F}A \to B$ and $T : \mathsf{F}A \to B$, $S \cup T$ is monotonic (strictly monotonic, completely monotonic) on a relation R if both S and T are monotonic (strictly monotonic, completely monotonic) on R.

Proof. We prove the case of monotonicity. Others are similar.

$$
\begin{array}{rcl}
R \circ (S \cup T) \supseteq (S \cup T) \circ \mathsf{F}R & \Leftrightarrow & \{ \text{ right distributivity of } \circ \text{ over } \cup \ \} \\
& & R \circ (S \cup T) \supseteq (S \circ \mathsf{F}R \cup T \circ \mathsf{F}R) \\
& \Leftrightarrow & \{ \text{ property of } \cup \ \} \\
& & (R \circ (S \cup T) \supseteq S \circ \mathsf{F}R) \land (R \circ (S \cup T) \supseteq T \circ \mathsf{F}R) \\
& \Leftarrow & \{ \text{ trivial } (\cup) \ \} \\
& & (R \circ S \supseteq S \circ \mathsf{F}R) \land (R \circ T \supseteq T \circ \mathsf{F}R) \\
& \Leftrightarrow & \{ \text{ premise } \} \\
& & \mathit{True}
\end{array}
$$

□

Lemma 4.7. Given relations $S{:}\mathsf{F}A \leftrightarrow A$ and $P{:}\mathsf{G}A \leftrightarrow A$, $P{\circ}S$ is monotonic (strictly monotonic, completely monotonic) on a relation R if both P is monotonic (strictly monotonic, completely monotonic) on R and S is monotonic (strictly monotonic, completely monotonic) on $\mathsf{G}R$.

Proof. We prove the case of monotonicity. Others are similar.

$$\begin{array}{rcl} R \circ P \circ S & \supseteq & \{ \text{ monotonicity } \} \\ & & P \circ \mathsf{G}R \circ S \\ & \supseteq & \{ \text{ monotonicity } \} \\ & & P \circ S \circ \mathsf{GF}R \end{array}$$

□

Partiality, or constraint on solutions, is difficult to deal with. One solvent is to retain a set of minimals such that at least one will satisfy the constraint.

Lemma 4.8. Given a predicate $p{:}A \to \mathit{Bool}$, let g be a function such that p is more collapsing than g. Then, a relation $p?$ is completely monotonic on a relation $R \cap =_g$ for any relation R.

Proof. First we prove that $a =_g b$ implies $p(a) = p(b)$.

$$\begin{array}{rcl} a =_g b & \Rightarrow & \{ \text{ definition of } =_g \} \\ & & g(a) = g(b) \\ & \Rightarrow & \{ \text{ } p \text{ is more collapsing than } g \text{ } \} \\ & & p(a) = p(b) \end{array}$$

Then, the claim apparently holds for the case of monotonicity, and since $\overset{R \cap =_g}{<} = \overset{R}{<} \cap =_g$ holds, so does for the case of strictly monotonicity. □

By combining Lemma 4.8 with Lemma 4.7, we can obtain monotonicity condition for partial functions in ease: decompose a partial function into a total function and a predicate that stands for the domain of the function; then, these two lemmas show a way to derive monotonicity condition. The point is to explicitly specify the partiality by a predicate.

Let us demonstrate the effectiveness of our lemmas. As an example, consider a work planning problem. We would like to plan our schedule of working. For each day, we can choose one of three choices: hard work, moderate work, and rest. Hard work will result in much profit, while we should take a rest in the next day of the hard work. Let h_i and m_i be respectively the profits of hard work and moderate work of the ith day. Then, given values of $h_1, \ldots, h_n, m_1, \ldots, m_n$, we would like to obtain the best schedule that yields the maximum profit for the next n days.

In this problem, a candidate is a schedule, which is a sequence of the way of work. Deciding a daily schedule for each day is formalized as the following relation *schedule*.

$$\begin{array}{lcl} ([a] +\!\!\!+ x, x) \in \mathit{schedule} & \overset{\text{def}}{\Longleftrightarrow} & a \in \{\mathit{Hard}, \mathit{Moderate}, \mathit{Rest}\} \wedge (\mathit{notHard}(x) \vee a = \mathit{Rest}) \\ \mathit{notHard}([\mathit{Hard}] +\!\!\!+ x) & \overset{\text{def}}{=} & \mathit{False} \\ \mathit{notHard}(_) & \overset{\text{def}}{=} & \mathit{True} \end{array}$$

Then, we can generate a schedule for the next n days by $\Lambda(\mathit{schedule}^n)([])$. Now we would like to find a quasi-order on which *schedule* is monotonic for utilizing the thinning theorem. First, decompose *schedule* into total functions and filters as follows.

$$\begin{aligned}
&\mathit{schedule} = (\mathit{addHard} \circ \mathit{notHard}?) \cup (\mathit{addMod} \circ \mathit{notHard}?) \cup \mathit{addRest} \\
&\mathit{addHard}(x) \stackrel{\text{def}}{=} [\mathit{Hard}] +\!\!+\, x \\
&\mathit{addMod}(x) \stackrel{\text{def}}{=} [\mathit{Moderate}] +\!\!+\, x \\
&\mathit{addRest}(x) \stackrel{\text{def}}{=} [\mathit{Rest}] +\!\!+\, x
\end{aligned}$$

Note that all of *addHard*, *addMod*, and *addRest* are total functions. Let P be the order to compare profit; then, from Lemma 4.8, *notHard*? is monotonic on $P \cap =_{\mathit{notHard}}$. It is easy to confirm that *addHard*, *addMod*, and *addRest* are monotonic on $P \cap =_{\mathit{notHard}}$, which is equivalent to that they are monotonic on $(P \cap =_{\mathit{notHard}})^{\circ}$ because of Lemma 4.5. Therefore, from Lemmas 4.6 and 4.7, *schedule* is monotonic on $(P \cap =_{\mathit{notHard}})^{\circ}$. In other words, we can obtain optimal scheduling by considering a case analyses in which we distinguish whether we worked hard in the previous day.

4.1.2 Constructing Monotonic Orders

We have examined structures to generate candidates. Next, let us consider structures of orders that we would like to optimize.

First, equivalence relations satisfy good properties.

Lemma 4.9. Any relation is strictly monotonic on any equivalence relation.

Proof. It is evident from that any equivalence relation has no strict part. □

Lemma 4.10. Any relation is completely monotonic on the equivalence relation =.

Proof. For any relation R, $a = b$ implies $\Lambda R(a) = \Lambda R(b)$, and thus, R is monotonic on =. From Lemma 4.9, any relation is strictly monotonic on =. In summary, any relation is completely monotonic on =. □

Monotonicity conditions are closed under intersections of orders, which is useful to construct a weaker order.

Lemma 4.11. For a relator F and relations R and S, a function $f : \mathsf{F}A \to A$ is monotonic on $R \cap S$, if f is monotonic on S and $(S \Rightarrow R) \circ f \supseteq f \circ \mathsf{F}R$ holds.

Proof.

$$
\begin{array}{ll}
& (R \cap S) \circ f \supseteq f \circ \mathsf{F}(R \cap S) \\
\Leftarrow & \{ \text{Lemma 2.6} \} \\
& (R \cap S) \circ f \supseteq f \circ (\mathsf{F}R \cap \mathsf{F}S) \\
\Leftrightarrow & \{ R \cap S = (S \Rightarrow R) \cap S \} \\
& ((S \Rightarrow R) \cap S) \circ f \supseteq f \circ (\mathsf{F}R \cap \mathsf{F}S) \\
\Leftrightarrow & \{ f \text{ is simple} \} \\
& ((S \Rightarrow R) \circ f) \cap (S \circ f) \supseteq f \circ (\mathsf{F}R \cap \mathsf{F}S) \\
\Leftarrow & \{ \text{premise} \} \\
& (f \circ \mathsf{F}R) \cap (f \circ \mathsf{F}S) \supseteq f \circ (\mathsf{F}R \cap \mathsf{F}S) \\
\Leftrightarrow & \{ \text{property of intersections} \} \\
& (f \circ \mathsf{F}R \supseteq f \circ (\mathsf{F}R \cap \mathsf{F}S)) \wedge (f \circ \mathsf{F}S \supseteq f \circ (\mathsf{F}R \cap \mathsf{F}S)) \\
\Leftrightarrow & \{ \text{trivial } (\cap) \} \\
& \mathit{True}
\end{array}
$$

□

Lemma 4.12. For a relator F and relations R and S, a function $f : \mathsf{F}A \to A$ is strictly monotonic on $R \cap S$, if f is strictly monotonic on S and $(\overset{S}{<} \Rightarrow \overset{R}{<}) \circ f \supseteq f \circ \overset{\mathsf{F}R}{<}$ holds.

Proof. The proof is almost the same as that of Lemma 4.11. □

Lemma 4.13. For a relator F and relations R and S, a function $f : \mathsf{F}A \to A$ is monotonic (strictly monotonic, completely monotonic) on $R \cap S$, if f is monotonic (strictly monotonic, completely monotonic) on both R and S.

Proof. It is evident from Lemmas 4.11 and 4.12 because $(S \Rightarrow R) \supseteq R$ holds. □

It is worth noting that Lemmas 4.11, 4.12, and 4.13 show effectiveness of use of functions instead of relations.

Next is about sequential compositions of orders.

Lemma 4.14. Given a polynomial functor F and relations R and S, a function $f : \mathsf{F}A \to A$ is monotonic (strictly/completely monotonic) on $R \,;\, S$, if f is completely monotonic on S and monotonic (strictly/completely monotonic) on R.

Proof. First we prove the monotonic case.

$$
\begin{array}{ll}
& (R \,;\, S) \circ f \supseteq f \circ \mathsf{F}(R \,;\, S) \\
\Leftrightarrow & \{ \text{definition of } R \,;\, S \} \\
& (S \cap (S^\circ \Rightarrow R)) \circ f \supseteq f \circ \mathsf{F}(S \cap (S^\circ \Rightarrow R)) \\
\Leftarrow & \{ \mathsf{F} \text{ is polynomial, and Lemmas 2.6 and 2.9} \} \\
& (S \cap (S^\circ \Rightarrow R)) \circ f \supseteq f \circ (\mathsf{F}S \cap (\mathsf{F}S^\circ \Rightarrow \mathsf{F}R)) \\
\Leftrightarrow & \{ \text{definition of } \Rightarrow \text{ and distributivity} \} \\
& ((S \cap \overline{S^\circ}) \cup (S \cap R)) \circ f \supseteq f \circ ((\mathsf{F}S \cap \overline{\mathsf{F}S^\circ}) \cup (\mathsf{F}S \cap \mathsf{F}R)) \\
\Leftrightarrow & \{ \text{distributively of compositions over unions} \} \\
& ((S \cap \overline{S^\circ}) \circ f) \cup ((S \cap R) \circ f) \supseteq (f \circ (\mathsf{F}S \cap \overline{\mathsf{F}S^\circ})) \cup (f \circ (\mathsf{F}S \cap \mathsf{F}R)) \\
\Leftarrow & \{ \text{trivial } (\cup) \} \\
& ((S \cap \overline{S^\circ}) \circ f \supseteq f \circ (\mathsf{F}S \cap \overline{\mathsf{F}S^\circ})) \wedge ((S \cap R) \circ f \supseteq f \circ (\mathsf{F}S \cap \mathsf{F}R)) \\
\Leftarrow & \{ \text{Lemma 4.13 and the premise} \} \\
& \mathit{True}
\end{array}
$$

Next we prove the strictly monotonic case.

$$
\begin{array}{ll}
& \overset{R\,;\,S}{<} \circ f \supseteq f \circ \overset{\mathsf{F}(R\,;\,S)}{<} \\
\Leftarrow & \{ \text{ Lemmas 4.26 and 4.27 } \} \\
& (\overset{R}{<}\,;\,S) \circ f \supseteq f \circ (\overset{\mathsf{F}S}{<} \cup (\mathsf{F}S \cap \overset{\mathsf{F}R}{<})) \\
\Leftrightarrow & \{ \text{ definition of } \overset{R}{<}\,;\,S \ \} \\
& (\overset{S}{<} \cup (S \cap \overset{R}{<})) \circ f \supseteq f \circ (\overset{\mathsf{F}S}{<} \cup (\mathsf{F}S \cap \overset{\mathsf{F}R}{<})) \\
\Leftarrow & \{ \text{ the premises, and do similar calculation to the previous case } \} \\
& \mathit{True}
\end{array}
$$

The completely monotonic case follows from the two results. □

Lemma 4.14 is useful in practice. An order $R\,;\,S$ is used to solve multi-objective optimization problems, in which one would like to find R-minimum elements in the S-minimum elements.

Let us consider examples borrowed from [MPRS99]. Consider a graph in which each edge has two weights, cost and capacity. The cost of a path is the sum of the costs of edges in the path, and the capacity of a path is the minimum capacity of edges. Now consider finding the minimum-cost path or the maximum-capacity path between given two vertexes. We may construct candidates of solutions by extending paths one by one, and thus, we would like to know whether an extension of a path satisfies monotonicity conditions. For the minimum-cost path problem, an extension of a path is completely monotonic; for two paths p_1 and p_2 of the same destination, where the cost of p_1 is strictly less than (or equal to) that of p_2, the cost of $p_1 +\!\!+ [e]$ is certainly strictly less than (equal to) $p_2 +\!\!+ [e]$. For the maximum-capacity path problem, an extension of a path is monotonic but not strictly monotonic; for two paths p_1 and p_2 of the same destination, where the capacity of p_1 is strictly less than that of p_2, the capacity of $p_1 +\!\!+ [e]$ is less than or equal to $p_2 +\!\!+ [e]$. The observation above and Lemma 4.14 indicate an efficient algorithm to find the maximum-capacity path among the minimum-cost paths; however, it might be difficult to find the minimum-cost path among the maximum-capacity paths, because Lemma 4.14 is not applicable.

Well, let us consider objective functions that map candidates onto an ordered set. Objective functions are useful to describe criterion of optimality. In fact, the notion of objective functions is also useful to obtain monotonicity condition. Later in Chapter 5, we will demonstrate effectiveness of description of objective functions for constructing efficient algorithms.

The following lemmas provide a way to check whether R_f satisfies monotonicity properties, in which f corresponds to an objective function.

Lemma 4.15 (variant of Proposition 9.2 of [BdM96])**.** For a relator F, a total function $f : B \to A$, and relations $R : A \leftrightarrow A$, $S : \mathsf{F}B \leftrightarrow B$, and $S' : \mathsf{F}A \leftrightarrow A$, assume that f is incremental on S by S'. Then, S is monotonic (strictly monotonic, completely

monotonic) on R_f if and only if S' is monotonic (strictly monotonic, completely monotonic, respectively) on R on the range of $\mathsf{F}f$.

Proof. We prove the case of monotonicity. Others are similar.

$$
\begin{array}{lll}
R_f \circ S \supseteq S \circ \mathsf{F}R_f & \Leftrightarrow & \{ \text{ definition of } R_f \ \} \\
& & f^\circ \circ R \circ f \circ S \supseteq S \circ \mathsf{F}(f^\circ \circ R \circ f) \\
& \Leftrightarrow & \{ \text{ property of total functions (2.3) } \} \\
& & R \circ f \circ S \supseteq f \circ S \circ \mathsf{F}(f^\circ \circ R \circ f) \\
& \Leftrightarrow & \{ \text{ premise } \} \\
& & R \circ S' \circ \mathsf{F}f \supseteq S' \circ \mathsf{F}f \circ \mathsf{F}f^\circ \circ \mathsf{F}R \circ \mathsf{F}f \\
& \Leftrightarrow & \{ \text{ property of total functions (2.4) } \} \\
& & R \circ S' \supseteq S' \circ \mathsf{F}f \circ \mathsf{F}f^\circ \circ \mathsf{F}R \circ \mathsf{F}f \circ \mathsf{F}f^\circ \\
& \Leftarrow & \{ \ \mathsf{F}f \text{ is a function; thus } \mathsf{F}f \circ \mathsf{F}f^\circ \subseteq id. \ \} \\
& & R \circ S' \supseteq S' \circ \mathsf{F}R
\end{array}
$$

□

Lemma 4.16. For a relation S and a total function f, S is completely monotonic on $=_f$ if and only if f is incremental on S.

Proof. It is a direct consequence of Lemmas 4.10 and 4.15. □

Lemmas 4.15 and 4.16 demonstrate that incrementality is a key to confirming monotonicity conditions. In fact, we will make intensive use of Lemma 4.16 to derive efficient algorithms.

The following lemma shows a similar result, which enables us to deal with objective functions whose value depends on contextual values.

Lemma 4.17. Given a relator F, a total function $f : B \to A$, function $h : \mathsf{F}B \to B$, and a relation $R : A \leftrightarrow A$, assume that there exist functions $g : B \to C$ and $h' : \mathsf{F}(A \times C) \to A$ such that g is incremental on h and both of $f \circ h = h' \circ \mathsf{F}(f \vartriangle g)$ and $R \circ h' \supseteq h' \circ \mathsf{F}(R \times id)$ hold. Then, h is monotonic on $R_f \cap =_g$.

Proof.

$$
\begin{array}{ll}
\multicolumn{2}{l}{(R_f \cap =_g) \circ h \supseteq h \circ \mathsf{F}(R_f \cap =_g)} \\
\Leftrightarrow & \{ \ h \text{ is simple } \} \\
& (R_f \circ h) \cap (=_g \circ h) \supseteq h \circ \mathsf{F}(R_f \cap =_g) \\
\Leftrightarrow & \{ \text{ property of intersections } \} \\
& (R_f \circ h \supseteq h \circ \mathsf{F}(R_f \cap =_g)) \wedge (=_g \circ h \supseteq h \circ \mathsf{F}(R_f \cap =_g)) \\
\Leftarrow & \{ \text{ Lemma 4.16 } \} \\
& (R_f \circ h \supseteq h \circ \mathsf{F}(R_f \cap =_g)) \wedge (h \circ \mathsf{F}{=_g} \supseteq h \circ \mathsf{F}(R_f \cap =_g)) \\
\Leftarrow & \{ \text{ Lemma 2.6 } \} \\
& R_f \circ h \supseteq h \circ \mathsf{F}(R_f \cap =_g) \\
\Leftrightarrow & \{ \text{ Lemma 4.24 } \} \\
& \mathit{True}
\end{array}
$$

□

The equation $f \circ h = h' \circ \mathsf{F}(f \vartriangle g)$ means that the result of g is necessary together with the result of f for incremental execution of the objective function f on the generation of candidates h. In other words, the objective function f uses contextual information computed by g. Lemma 4.17 shows a way to derive efficient algorithms for such cases.

Now, let us demonstrate a use of Lemma 4.17 by another planning problem. Consider that we are managing a supercomputer, which can process a certain amount of data par day. However, because of bugs of software, the processing speed becomes slower day by day, and thus, we must repair the supercomputer periodically. Given an expectation of amounts of available data for the next month, we would like to obtain the best scheduling for repairing the computer so that we can maximize the amount of data processed.

We express a schedule as a sequence of $\{\mathit{Repair}(d), \mathit{Work}(d)\}$, where d is the amount of available data, and generate a schedule by a relation to decide daily schedules, which will be denoted by $\mathit{repair} \cup \mathit{work}$.

$$\begin{aligned} \mathit{schedule} &= \mathit{repair} \cup \mathit{work} \\ \mathit{work}(a, x) &= [\mathit{Work}(a)] +\!\!+\, x \\ \mathit{repair}(a, x) &= [\mathit{Repair}(a)] +\!\!+\, x \end{aligned}$$

The difficulty of this problem is that the value of objective function depends not only the amount of available data but also the processing speed. Let $\mathit{process}(k)$ be the processing speed of the computer that has been working consecutively for k days. Then, given a schedule, the objective function amount is specified as follows.

$$\begin{array}{lcl} \mathit{amount}([]) & \stackrel{\text{def}}{=} & 0 \\ \mathit{amount}([\mathit{Work}(d)] +\!\!+\, x) & \stackrel{\text{def}}{=} & (d \downarrow \mathit{process}(\mathit{working}(x))) + \mathit{amount}(x) \\ \mathit{amount}([\mathit{Repair}(d)] +\!\!+\, x) & \stackrel{\text{def}}{=} & \mathit{amount}(x) \\ \mathit{working}([]) & \stackrel{\text{def}}{=} & 0 \\ \mathit{working}([\mathit{Work}(d)] +\!\!+\, x) & \stackrel{\text{def}}{=} & 1 + \mathit{working}(x) \\ \mathit{working}([\mathit{Repair}(d)] +\!\!+\, x) & \stackrel{\text{def}}{=} & 0 \end{array}$$

It is natural to introduce an auxiliary function $\mathit{working}$ for defining the objective function amount. This indicates that there exists a function work' such that $\mathit{amount}(\mathit{work}([a] +\!\!+\, x)) = \mathit{work}'(a, (\mathit{amount} \vartriangle \mathit{working})(x))$ holds, and in fact, the following function work' satisfies the requirement.

$$\begin{array}{lcl} \mathit{work}'(\mathit{Work}(d), (r, k)) & = & (d \downarrow \mathit{process}(k)) + r \\ \mathit{work}'(\mathit{Repair}(d), (r, k)) & = & r \end{array}$$

Next, observe that the schedule generating functions are monotonic on $\leq_{\mathit{amount}}$ if we could regard the value of $\mathit{working}$ to be constant, i.e., $r \leq r'$ implies $\mathit{work}'(a, (r, k)) \leq \mathit{work}'(a, (r', k))$; in addition, $\mathit{working}$ is incremental on schedule generating functions. In summary, Lemma 4.17 is applicable for this problem and derives an efficient algorithm that distinguishes each solution by the number of days that the computer has been consecutively working for.

We have explained how to obtain appropriate orders constructively. As a remark, we would like to mention well-supportedness. The proposed constructions preserve well-supportedness, as the following lemmas show; thus, we can use constructed orders without caring well-supportedness.

Lemma 4.18. Any equivalence relation is well-supported.

Proof. It is evident because $\Lambda mnl_E = id$ holds when E is an equivalence relation. □

Lemma 4.19 ([BdM92])**.** For any polynomial functor F, function f, and well-supported relations R and S, R_f, $R \cap S$, $R \mathbin{;} S$, and $\mathsf{F}R$ are well-supported. □

4.1.3 Deriving Dynamic Programming Algorithms

So far, we have introduced several laws for confirming and obtaining monotonicity conditions. As a summary of our laws, here we introduce calculational laws that are useful to solve a class of combinatorial optimization problems.

The first one is the following theorem.

Theorem 4.20 (derivation of dynamic programming algorithms)**.** The following are given: an index set I; predicates $p : A \to Bool$ and $q_i : \mathsf{F}A \to Bool$ $(i \in I)$; functions $f_i : \mathsf{F}A \to A$ $(i \in I)$, $g_p : \mathsf{F}A \to C$, and $g_q : \mathsf{F}A \to D$; a quasi-order $R : A \leftrightarrow A$. Assume the following three: (1) each function f_i is monotonic on R; (2) p and each q_i are respectively more collapsing than g_p, and g_q; (3) both g_p and g_q are incremental on each f_i. Then, the following equations hold, where R' is defined as $R' \stackrel{\text{def}}{=} R {\cap} {=_{g_p}} {\cap} {=_{g_q}}$.

$$min_R \circ p\lhd \circ \Lambda(\!|\bigcup_{i\in I}(f_i \circ q_i?)|\!) \supseteq min_R \circ p\lhd \circ (\!|thin_{R'} \circ \Lambda(\bigcup_{i\in I}(f_i \circ q_i?) \circ \mathsf{F}{\in})|\!)$$
$$min_R \circ p\lhd \circ \Lambda(\bigcup_{i\in I}(f_i \circ q_i?))* \supseteq min_R \circ p\lhd \circ (thin_{R'} \circ \Lambda(\bigcup_{i\in I}(f_i \circ q_i?) \circ {\in}))* \circ \{\cdot\}$$

Proof. From Lemma 4.28, $min_R \circ p\lhd = p? \circ min_{R ; \preceq_p}$ holds, where $\preceq$ is the linear order on which *True* is strictly smaller than *False*. Because of Lemma 4.29 and the premise that p is more collapsing than g_p, $R \mathbin{;} \preceq_p \supseteq R \cap =_p \supseteq R \cap =_{g_p} \supseteq R'$ holds. Thus, from the thinning theorem (Theorem 3.17 or Theorem 3.18), it is sufficient to prove that $\bigcup_{i\in I}(f_i \circ q_i?)$ is monotonic on R'°, which is a consequence of the combination of Lemmas 4.5, 4.6, 4.7, 4.8, 4.13, and 4.16. □

Theorem 4.20 provides a way to solve a wide class of combinatorial optimization problems. We can read the theorem as follows. First, we can solve a simple problem to find an R-minimum element without any requirement on solutions. The problem will be solved easily, because each f_i is monotonic on R. Then, the theorem states that we can solve a more complicated problem having additional constraint by considering appropriate case analyses.

Since Theorem 4.20 is formalized in terms of *thin*, we need to provide an implementation of *thin*. When R is well-supported, we can adopt $\Lambda mnl_{R'}$ for this purpose

because of Lemmas 4.18 and 4.19. In the following, we implicitly assume that R is well-supported and consider that $thin_{R'}$ is implemented by $\Lambda mnl_{R'}$.

Theorem 4.20 states that $(R\cap=_{g_p}\cap=_{g_q})$-minimal solutions are sufficient to obtain an optimal solution if the premises are satisfied. Notice that $(R \cap =_{g_p} \cap =_{g_q})$ is the order where two candidates a and b are compared by R if and only if both elements belong to the same equivalent class of $=_{g_p} \cap =_{g_q}$, i.e., $g_p(a) = g_p(b) \wedge g_q(a) = g_q(b)$ holds. Therefore, the derived programs can be recognized as dynamic programming algorithms. In each recursion, we fill a table whose key and value are respectively an equivalent classes of $=_{g_p} \cap =_{g_q}$ and R-minimal solutions of in the class.

Let us examine the time complexity of the resulted programs of Theorem 4.20. Assume all of the costs to compute each f_i, g_p, g_q, R, and p to be constant, and let k be the number of equivalent classes of $=_{g_p} \cap =_{g_q}$. In addition, assume R is a linear order; then, it is sufficient to keep only one element for each equivalent class, and thus, k candidates are considered in each step. Now the time complexity of the derived programs is $\mathrm{O}(knI)$, where n is the number of recursions, which is the size of the input structure for the case of catamorphisms, and I is the number of choices of each step. Since parameters except k are determined from the specification, the choices of g_p and g_q determine the efficiency of the derived program. If R is not a linear order, the theorem might not be effective. For example, $thin_{R'}$ can discard no candidate when R is an equivalence relation. But the theorem is usually effective in practice. It is worth noting that memoization may improve efficiency, especially the solutions are large structures and each choice corresponds to a construction of a structure.

Theorem 4.20 is a generalization of the results about maximum marking problems. Recall Theorem 3.21, where the constraint is $(accept \circ \mathsf{foldr}_{\oplus,e})\lhd$. Observe that $\mathsf{foldr}_{\oplus,e}$, which is less collapsing than $accept \circ \mathsf{foldr}_{\oplus,e}$ (Lemma 4.2), is incremental on the generation of markings. Therefore, Theorem 4.20 is applicable for the maximum marking problems considered here, and yields linear-time algorithms if the range of $\mathsf{foldr}_{\oplus,e}$ is finite. In short, incrementality condition is the key to efficient algorithms.

By virtue of the calculational laws we have prepared, we can prove several variants of Theorem 4.20. For example, we can prove that the following equation holds under a similar premise to Theorem 4.20, in which strict monotonicity is required instead of monotonicity.

$$\Lambda mnl_R \circ p\lhd \circ \Lambda(\![\bigcup_{i\in I}(f_i \circ q_i?)]\!) = \Lambda mnl_R \circ p\lhd \circ (\![\Lambda min_{R'} \circ \Lambda(\bigcup_{i\in I}(f_i \circ q_i?) \circ \mathsf{F}\in)]\!)$$

Moreover, it is not difficult to extend Theorem 4.20 so as to deal with multi-objective optimization problems. The following is one for the case of catamorphisms.

Theorem 4.21. The following is given: an index set I; predicates p_j $(1 \le j \le k)$ and q_i $(i \in I)$; functions f_i $(i \in I)$, g_{p_j} $(1 \le j \le k)$, and g_{q_i} $(i \in I)$; total well-supported quasi-orders R_j $(1 \le j \le k)$. Assume the following three: (1) each function f_i is completely monotonic on each R_j; (2) each p_j and each q_i are respectively more

collapsing than g_{p_j} and g_{q_i}; (3) all of g_{p_j} and g_{q_i} are incremental on each f_i. Then, the following equation holds.

$$\begin{aligned}
&\Lambda mnl_{R_1} \circ p_1 \lhd \circ \cdots \circ \Lambda mnl_{R_k} \circ p_k \lhd \circ \Lambda(\llbracket \textstyle\bigcup_{i \in I}(f_i \circ q_i?) \rrbracket) \\
&= \Lambda mnl_{R_1} \circ p_1 \lhd \circ \cdots \circ \Lambda mnl_{R_k} \circ p_k \lhd \circ (\llbracket \Lambda mnl_{R'} \circ \Lambda(\textstyle\bigcup_{i \in I}(f_i \circ q_i?) \circ \mathsf{F}\in) \rrbracket) \\
&\quad R' \stackrel{\text{def}}{=} (R_1 \mathbin{;} R_2 \mathbin{;} \cdots \mathbin{;} R_k) \cap (\textstyle\bigcap_{1 \le j \le k} =_{g_{p_j}}) \cap (\textstyle\bigcap_{i \in I} =_{g_{q_i}}).
\end{aligned}$$

Proof. The proof is similar to that of Theorem 4.20, and we just outline it. Let $\preceq$ be the order on which *True* is strictly smaller than *False*.

$$\begin{aligned}
&\Lambda mnl_{R_1} \circ p_1 \lhd \circ \cdots \circ \Lambda mnl_{R_k} \circ p_k \lhd \circ \Lambda(\llbracket \textstyle\bigcup_{i \in I}(f_i \circ q_i?) \rrbracket) \\
&= \{ \text{ pushing out filters (Lemma 4.28) } \} \\
&\quad p_1 \lhd \circ \cdots \circ p_k \lhd \circ \Lambda mnl_{R_1 ; \preceq_{p_1} ; \cdots ; \preceq_{p_k}} \circ \cdots \circ \Lambda mnl_{R_k ; \preceq_{p_k}} \circ \Lambda(\llbracket \textstyle\bigcup_{i \in I}(f_i \circ q_i?) \rrbracket) \\
&= \{ \text{ compressing minimals (Lemma 4.31: note } R_i \mathbin{;} \preceq_{p_i} \mathbin{;} \cdots \mathbin{;} \preceq_{p_k} \text{ is total) } \} \\
&\quad p_1 \lhd \circ \cdots \circ p_k \lhd \circ \Lambda mnl_{(R_1 ; \preceq_{p_1} ; \cdots ; \preceq_{p_k}) ; \cdots ; (R_k ; \preceq_{p_k})} \circ \Lambda(\llbracket \textstyle\bigcup_{i \in I}(f_i \circ q_i?) \rrbracket) \\
&= \{ \text{ minimal-based thinning theorem (Theorem 3.19) } \} \\
&\quad p_1 \lhd \circ \cdots \circ p_k \lhd \circ \Lambda mnl_{(R_1 ; \preceq_{p_1} ; \cdots ; \preceq_{p_k}) ; \cdots ; (R_k ; \preceq_{p_k})} \circ (\llbracket \Lambda mnl_{R'} \circ \Lambda(\textstyle\bigcup_{i \in I}(f_i \circ q_i?) \circ \mathsf{F}\in) \rrbracket) \\
&= \{ \text{ inverse of the steps above } \} \\
&\quad \Lambda mnl_{R_1} \circ p_1 \lhd \circ \cdots \circ \Lambda mnl_{R_k} \circ p_k \lhd \circ (\llbracket \Lambda mnl_{R'} \circ \Lambda(\textstyle\bigcup_{i \in I}(f_i \circ q_i?) \circ \mathsf{F}\in) \rrbracket) \qquad \square
\end{aligned}$$

4.2 Deriving Algorithms for Shortest Path Problems and Their Variants

In this section, we demonstrate effectiveness of our calculational laws through derivations of algorithms for shortest path problems and their variants. Here we consider three kinds of problems: shortest path problems, regular-language-constrained shortest path problems [Rom88, BJM00], and resource-constrained shortest path problems [Jok66].

We assume that there is no cycle whose weight is negative in the underlying graph. Then, all quasi-orders used here are well-supported, because weights of paths are lower bounded. Therefore, we can use Λmnl_R to implement $thin_R$, and actually we implicitly recognize $thin_R$ as Λmnl_R.

4.2.1 Shortest Path Problems

As seen in Chapter 3, the following is the specification of the shortest path problem, where s is the source and t is the destination.

$$SP \stackrel{\text{def}}{=} (min_{\le_w} \circ endWith_t \lhd \circ from_s \lhd \circ \Lambda(\bigcup_{e \in E} extend_e)*)\ []$$

$$\begin{aligned}
extend_e(p) &\stackrel{\text{def}}{=} p \mathbin{+\!\!+} [e] \quad \text{if } dst(p) = hd(e) \\
endWith_v(p) &\stackrel{\text{def}}{=} dst(p) = v \\
from_s([]) &\stackrel{\text{def}}{=} False \\
from_s([e] \mathbin{+\!\!+} p) &\stackrel{\text{def}}{=} hd(e) = s
\end{aligned}$$

For simplicity of presentations, we treat $dst([]) \stackrel{\text{def}}{=} s$ in this section. Then, the definition of the shortest path problem is simplified as follows.

$$SP = (min_W \circ endWith_t \lhd \circ \Lambda(\bigcup_{e \in E} extend_e)*)\ []$$

Bellman-Ford Algorithm

The specification shown above is already in a form that Theorem 4.20 is applicable. Let us confirm the premises of the theorem.

The first one is the monotonicity condition. To avoid treating partial functions $extend_e$, we decompose each $extend_e$ into a total function $addEdge_e$ with a predicate $endWith_v$.

$$\begin{aligned} extend_e &= addEdge_e \circ endWith_{hd(e)}? \\ addEdge_e(p, v) &\stackrel{\text{def}}{=} p +\!\!+ [e] \end{aligned}$$

The predicate $endWith_{hd(e)}$ checks whether we can extend a path by adding edge e, and $addEdge_e$ really extends a path by adding edge e. It is evident that $addEdge_e$ is completely monotonic on $\leq_w$.

Next, it is necessary to prepare a function that is less collapsing than $endWith_v$ and incremental on $addEdge_e$. Observe that $endWith_v$ only cares destinations rather than whole paths. Therefore, dst is less collapsing than $endWith_v$. Let us confirm the incrementality condition.

$$\begin{aligned} (dst \circ addEdge_e)(p) &= \{ \text{ definition of } addEdge_e \ \} \\ & \quad\ dst(p +\!\!+ [e]) \\ &= \{ \text{ definition of } dst \ \} \\ & \quad\ tl(e) \\ &= \{ \text{ Let } g_e(_) \stackrel{\text{def}}{=} tl(e) \ \} \\ & \quad\ (g_e \circ dst)(p) \end{aligned}$$

Thus, dst is certainly incremental on each $addEdge_e$.

Now that all premises of Theorem 4.20 is fulfilled, the theorem enables us to obtain a dynamic programming algorithm.

$$\begin{aligned} SP &= \{ \text{ definition of } SP \ \} \\ & \quad (min_{\leq_w} \circ endWith_t \lhd \circ \Lambda(\bigcup_{e \in E} extend_e)*)\ [] \\ &= \{ \text{ decompose } extend_e \ \} \\ & \quad (min_{\leq_w} \circ endWith_t \lhd \circ \Lambda(\bigcup_{e \in E}(addEdge_e \circ endWith_{hd(e)}?))*)\ [] \\ &= \{ \text{ Theorem 4.20 } \} \\ & \quad (min_{\leq_w} \circ endWith_t \lhd \circ \\ & \quad\ (thin_{\leq_w \cap =_{dst}} \circ \Lambda(\bigcup_{e \in E}(addEdge_e \circ endWith_{hd(e)}?) \circ \in))*)\ \{[]\} \\ &= \{ \text{ folding } extend_e \ \} \\ & \quad (min_{\leq_w} \circ endWith_t \lhd \circ (thin_{\leq_w \cap =_{dst}} \circ \Lambda(\bigcup_{e \in E} extend_e \circ \in))*)\ \{[]\} \end{aligned}$$

The quasi-order $\leq_w \cap =_{dst}$ compares paths if their destination is the same; thus the derived algorithm recursively computes candidates of shortest paths from s to every

vertex, until we cannot find better paths any more. This algorithm is exactly the Bellman-Ford algorithm, whose time complexity is $\mathrm{O}(VE)$.

Dijkstra Algorithm

Dijkstra algorithm is effective when weights of edges are positive. The idea is to delay considering paths that may not be optimal. To express such delay, we give a counter that counts the number of repetitions accomplished.

$$\begin{aligned} &SP = (min_{\leq_w} \circ endWith_t \triangleleft \circ \mathsf{P}\pi_1 \circ \Lambda(next \cup \textstyle\bigcup_{e\in E} extendD_e)*)\ ([],0) \\ &\quad extendD_e(p,k) \stackrel{\text{def}}{=} (p +\!\!+ [e], k+1) \quad \text{if } hd(e) = dst(p) \wedge k \geq \chi(tl(e)) \\ &\quad next(p,k) \stackrel{\text{def}}{=} (p, k+1) \end{aligned}$$

The function $extendD_e$ does not construct a path to the vertex $tl(e)$ until the $\chi(tl(e))$-th repetition, and the function $\chi : V \to \mathbb{Z}_+$ manages the delay. Note that the expression $\mathsf{P}\pi_1 \circ \Lambda(\bigcup_{e\in E} extendD_e)*$ certainly enumerates all paths, because $extendD_e$ does equivalent computation to $extend_e$ after sufficient number of repetitions.

We cannot apply Theorem 4.20 directly to the equation above; thus, let us calculate a bit.

$$\begin{aligned} SP \ &= \ \{ \text{ delayed variant of } SP \ \} \\ &\quad (min_{\leq_w} \circ endWith_t \triangleleft \circ \mathsf{P}\pi_1 \circ \Lambda(next \cup \textstyle\bigcup_{e\in E} extendD_e)*)\ ([],0) \\ &= \ \{ \text{ trivial (projection and map) } \} \\ &\quad (\mathsf{P}\pi_1 \circ min_{\leq_{w\circ\pi_1}} \circ (endWith_t \circ \pi_1)\triangleleft \circ \Lambda(next \cup \textstyle\bigcup_{e\in E} extendD_e)*)\ ([],0) \end{aligned}$$

We have derived a form that Theorem 4.20 is applicable to, and we would like to confirm the premises.

First, decompose $extendD_e$ into a total function with a filter.

$$\begin{aligned} &extendD_e = addEdge'_e \circ properD_e? \\ &\quad addEdge'_e(p,k) \stackrel{\text{def}}{=} (p +\!\!+ [e], k+1) \\ &\quad properD_e(p,k) = endWith_{hd(e)}(p) \wedge k \geq \chi(k) \end{aligned}$$

It is easy to see that $addEdge'$ is completely monotonic on $\leq_{w\circ\pi_1}$.

Since $properD_e$ is more collapsing than $(dst \times id)$, we would like to confirm the incrementality condition. It is trivial to confirm that $(dst \times id)$ is incremental on $next$. It is also incremental on $addEdge'$, as the following calculation shows.

$$\begin{aligned} ((dst \times id) \circ addEdge'_e)\ (p,k) \ &= \ \{ \text{ definition of } addEdge'_e \ \} \\ &\quad (dst \times id)(p +\!\!+ [e], k+1) \\ &= \ \{ \text{ definition of } dst \ \} \\ &\quad (tl(e), k+1) \\ &= \ \{ \text{ Let } g_e(_,k) \stackrel{\text{def}}{=} (tl(e), k+1) \ \} \\ &\quad (g_e \circ (dst \times id))(p,k) \end{aligned}$$

In addition, $endWith_t \circ \pi_1$ is also more collapsing than $(dst \times id)$. In summary, an algorithm is derived as follows.

$$
\begin{array}{ll}
& SP \\
= & (\mathsf{P}\pi_1 \circ min_{\leq_{w \circ \pi_1}} \circ (endWith_t \circ \pi_1)\triangleleft \circ \Lambda(next \cup \bigcup_{e \in E} extendD_e)*) \ ([], 0) \\
= & \{ \text{ decompose } extendD_e \ \} \\
& (\mathsf{P}\pi_1 \circ min_{\leq_{w \circ \pi_1}} \circ (endWith_t \circ \pi_1)\triangleleft \\
& \circ \Lambda(next \cup \bigcup_{e \in E}(addEdge'_e \circ properD_e?))*) \ ([], 0) \\
= & \{ \text{ Theorem 4.20, and let } W \stackrel{\text{def}}{=} \leq_{w \circ \pi_1} \cap =_{(dst \times id)} \ \} \\
& (\mathsf{P}\pi_1 \circ min_{\leq_{w \circ \pi_1}} \circ (endWith_t \circ \pi_1)\triangleleft \\
& \circ (thin_W \circ \Lambda(next \cup \bigcup_{e \in E}(addEdge'_e \circ properD_e?) \circ \in))*) \ \{([], 0)\} \\
= & \{ \text{ folding } extendD_e \text{ and trivial (projection and map) } \} \\
& (min_{\leq_w} \circ endWith_t \triangleleft \circ \mathsf{P}\pi_1 \circ (thin_W \circ \Lambda(next \cup \bigcup_{e \in E} extendD_e \circ \in))*) \ \{([], 0)\}
\end{array}
$$

In the calculation, we did not use any specific property of χ. This fact tells us that the correctness of the derived algorithm does not depend on selection of χ. In Dijkstra algorithm, χ results in the near-order ranking from the source vertex. Since all weights are positive, the nearest vertex that has not been visited yet is certainly the next nearest vertex in each step of computation. The time complexity of Dijkstra algorithm is $\mathrm{O}(V \log V + E)$, if we implement it efficiently using a Fibonacci heap. A^* search algorithms are also delayed variant but uses another delay.

4.2.2 Regular-Language-Constrained Shortest Path Problems

Next, let us consider a bit more complicated problems, regular-language-constrained shortest path problems [Rom88, BJM00]. Regular-language-constrained shortest path problems are the problems to compute the shortest path among those whose label is in a regular language. The following is the formal definition of the regular-language shortest path problems.

Definition 4.22 (regular-language-constrained shortest path problem)**.** Given a graph $G = (V, E)$ with a weight function $w{:}E \to \mathbb{R}$ and a labeling function $l{:}E \to \Sigma$, a source $s \in V$, a destination $t \in V$, and a regular language L, a *regular language constrained shortest path problem* is the problem to find the shortest path p from s to t satisfying $l(p) \in L$. □

Let us consider a DFA $\mathcal{A} = (S, \Sigma, \tau, S_I, S_F)$ such that $\mathcal{L}_\mathcal{A} = L$ to formalize the problem in a calculational style. The calculational definition of the regular-language-constrained shortest path problem, denoted by LSP, is the following.

$$
\begin{array}{rcl}
LSP & \stackrel{\text{def}}{=} & (min_{\leq_w} \circ accept\triangleleft \circ endWith_t\triangleleft \circ \Lambda(\bigcup_{e \in E} extend_e)*) \ [] \\
accept(p) & \stackrel{\text{def}}{=} & rp_\mathcal{A}(l(p)) \in S_F
\end{array}
$$

The definition is almost the same as that of shortest path problems, except for the additional constraint *accept*.

Label-correcting Algorithm

First, let us derive a label-correcting (Bellman-Ford like) algorithm.

Since regular-language-constrained problem has plural constraints, use of Theorem 4.21 would be appropriate. It is worth noting that we can reuse the calculation in the previous subsection, and it is sufficient to find a function that is less collapsing than *accept* and incremental on *addEdge*. From the definition of *accept*, $rp_{\mathcal{A}} \circ l$ is an apparent candidate: *accept* is more collapsing than $rp_{\mathcal{A}} \circ l$ from Lemma 4.2. Let us calculate to confirm its incrementality condition.

$$\begin{array}{lll} (rp_{\mathcal{A}} \circ l \circ addEdge_e)(p) & = & \{ \text{ definition of } addEdge_e \text{ and } l \ \} \\ & & rp_{\mathcal{A}}(l(p) +\!\!+ [l(e)]) \\ & = & \{ \text{ definition of } rp \ \} \\ & & \tau(rp_{\mathcal{A}}(l(p)), l(e)) \\ & = & \{ \text{ Let } \tau'_e(x) \stackrel{\text{def}}{=} \tau(x, l(e)) \ \} \\ & & (\tau'_e \circ rp_{\mathcal{A}} \circ l)(p) \end{array}$$

Now that all premises are confirmed, Theorem 4.21 derives the following program, where the quasi-order W' is defined as $W' \stackrel{\text{def}}{=} \leq_w \cap =_{dst} \cap =_{rp_{\mathcal{A}} \circ l}$.

$$(min_{\leq_w} \circ accept\triangleleft \circ endWith_t\triangleleft \circ (thin_{W'} \circ \Lambda(\bigcup_{e \in E} extend_e \circ \in))*) \ \{[]\}$$

In the program, paths are compared only if both of their destinations and representative states are the same. The time complexity of the derived algorithm is $O(S^2VE)$.

From other point of view, the program seeks the shortest path on a new graph whose each vertex corresponds to an element of $E \times S$. In other words, the new graph corresponds to the product of the original graph and the DFA $\mathcal{A}$. Recognize the original graph as an NFA that accepts a sequence of edges if and only if it is a path from s to t on the graph. Then, the product of the NFA and $\mathcal{A}$ is an automaton that accepts paths from s to t whose label is in L. The derived program looks for the minimum-weighed transition from the initial state to a finial state on the product. This idea was pointed out by Romeuf [Rom88].

Label-setting Algorithm

Next, let us consider a label-setting (Dijkstra-like) algorithm. Since it is sufficient to solve shortest path problems on the graph obtained by product construction, Dijkstra algorithm will be applicable when weights of edges are positive. Let us confirm this observation by calculating a label-setting algorithm for regular-language-constrained shortest path problems.

As similar to the previous case, most of premises to apply Theorem 4.21 have been confirmed in the previous section, and thus, it is sufficient to confirm incre-

mentality condition of $rp_{\mathcal{A}} \circ l \circ \pi_1$ on $addEdge'_e$.

$$\begin{array}{rcl} (rp_{\mathcal{A}} \circ l \circ \pi_1 \circ addEdge'_e)(p,k) & = & \{ \text{ definition of } addEdge'_e,\ \pi_1 \text{ and } l \ \} \\ & & rp_{\mathcal{A}}(l(p) +\!\!+ [l(e)]) \\ & = & \{ \text{ definition of } rp \ \} \\ & & \tau(rp_{\mathcal{A}}(l(p)), l(e)) \\ & = & \{ \text{ Let } \tau'_e(x) \stackrel{\text{def}}{=} \tau(x, l(e)) \ \} \\ & & (\tau'_e \circ rp_{\mathcal{A}} \circ l \circ \pi_1)(p,k) \end{array}$$

Now that we have confirmed the incrementality condition, Theorem 4.21 derives the following program, where the quasi-order W'' is defined as $W'' \stackrel{\text{def}}{=} \leq_{W \circ \pi_1} \cap =_{(dst \times id)} \cap =_{rp_{\mathcal{A}} \circ l \circ \pi_1}$.

$$(min_{\leq_w} \circ accept \triangleleft \circ endWith_t \triangleleft \circ \mathsf{P}\pi_1 \circ (thin_{W''} \circ \Lambda(next \cup \bigcup_{e \in E} extendD_e \circ \in))*) \ \{([], 0)\}$$

Now that we have confirmed the correctness of delayed variants, we can obtain a Dijkstra-like algorithm by choosing an appropriate delay. The time complexity of the derived algorithm is $\mathrm{O}(SV \log(SV) + SE)$, if we implement it efficiently using a Fibonacci heap. As the calculation shows, we can also use another searching algorithms such as A^* algorithms.

The point is that the delay by χ does not affect representative states of the DFA $\mathcal{A}$. In other words, we can consider improvement of efficiency, namely changing a way to search shortest paths, independently from the constraint.

4.2.3 Resource-Constrained Shortest Path Problems

As the final example in this chapter, let us consider another variant of shortest path problems, resource-constrained shortest path problems [Jok66]. Resource-constrained shortest path problems are problems to compute the shortest path whose cost is less than the given limit.

Definition 4.23 (resource-constrained shortest path problem)**.** Given a graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$ and a cost function $c : E \to \mathbb{Z}_+$, a source $s \in V$, a destination $t \in V$, and the limit $C \in \mathbb{N}$, a *resource-constrained shortest path problems* is the problem to find the shortest path p from s to t such that $c(p) \leq C$. □

The following is the calculational definition.

$$CSP \stackrel{\text{def}}{=} (min_{\leq_w} \circ costless_C \triangleleft \circ endWith_t \triangleleft \circ \Lambda(\bigcup_{e \in E} extend_e)*) \ []$$

$$costless_C(p) \stackrel{\text{def}}{=} c(p) \leq C$$

Label-correcting algorithm

Let us derive a label correcting algorithm. We would like to go similar way to the case of regular-language-constrained shortest path problems; however, in this case, it is better to calculate a bit in advance.

$$
\begin{array}{l}
\mathit{CSP} \\
= \{ \text{ definition } \} \\
\quad (\mathit{min}_{\leq_w} \circ \mathit{costless}_C \lhd \circ \mathit{endWith}_t \lhd \circ \Lambda(\bigcup_{e\in E} \mathit{extend}_e)*)\ [] \\
= \{ \text{ trivial (filter and power-transpose) } \} \\
\quad (\mathit{min}_{\leq_w} \circ \mathit{endWith}_t \lhd \circ \Lambda(\mathit{costless}? \circ (\bigcup_{e\in E} \mathit{extend}_e)*)\ [] \\
= \{ \text{ Theorem 2.22 (the premise will be confirmed later) } \} \\
\quad (\mathit{min}_{\leq_w} \circ \mathit{endWith}_t \lhd \circ \Lambda(\bigcup_{e\in E}(\mathit{costless}_C? \circ \mathit{extend}_e))* \circ \mathit{costless}_C?)\ [] \\
= \{ \ \mathit{costless}_C([]) = \mathit{True}, \text{ and unfolding } \mathit{extend}_e \ \} \\
\quad (\mathit{min}_{\leq_w} \circ \mathit{endWith}_t \lhd \circ \Lambda(\bigcup_{e\in E}(\mathit{costless}_C? \circ \mathit{addEdge}_e \circ \mathit{endWith}_{hd(e)}?))*)\ [] \\
= \{ \ \mathit{costless}_C \circ \mathit{addEdge}_e = \mathit{addEdge}_e \circ \mathit{costless}_{C-c(e)} \ \} \\
\quad (\mathit{min}_{\leq_w} \circ \mathit{endWith}_t \lhd \circ \Lambda(\bigcup_{e\in E}(\mathit{addEdge}_e \circ \mathit{costless}_{C-c(e)}? \circ \mathit{endWith}_{hd(e)}?))*)\ []
\end{array}
$$

The premise of Theorem 2.22 is the following equation, which is evident because the cost of each edge is positive.

$$\mathit{costless}_C? \circ \bigcup_{e\in E} \mathit{extend}_e = \mathit{costless}_C? \circ (\bigcup_{e\in E} \mathit{extend}_e) \circ \mathit{costless}_C?$$

Well, let us use Theorem 4.21. It is sufficient to find a function that is less collapsing than $\mathit{costless}_{C-c(e)}$ and incremental on $\mathit{addEdge}$. Apparently, the function c, which computes the cost of a path, is less collapsing than $\mathit{costless}_{C-c(e)}$. In addition, the function c satisfies the incrementality condition.

$$
\begin{array}{rcl}
(c \circ \mathit{addEdge}_e)(p) & = & \{ \text{ definition of } \mathit{addEdge}_e \text{ and } c \ \} \\
 & & c(p) + c(e) \\
 & = & \{ \text{ let } g_e(n) \stackrel{\text{def}}{=} c(e) + n \ \} \\
 & & (g_e \circ c)(p)
\end{array}
$$

All premises have been confirmed, and Theorem 4.21 derives the following algorithm, where the quasi-order $W^\dagger \stackrel{\text{def}}{=} \leq_w \cap =_{dst} \cap =_c$ is used to discard unnecessary candidates.

$$\mathit{CSP} = (\mathit{min}_{\leq_w} \circ \mathit{endWith}_t \lhd \circ (\mathit{thin}_{W^\dagger} \circ \Lambda(\bigcup_{e\in E}(\mathit{extend}_e \circ \mathit{costless}_{C-c(e)}?) \circ \in))*)\ \{[]\}$$

The order $W^\dagger$ compares paths of the same destination and cost. Thus, the derived algorithm computes candidates from the source vertex to every vertex of every cost recursively, until no better paths are found. This is a known dynamic-programming algorithm [Jok66], and its time complexity is $\mathrm{O}(C^2VE)$.

Label-setting algorithm

We can improve the efficiency by a label-setting algorithm when weights of edges are positive. The calculation is summarized as follows. Though the calculation seems a bit complicated, each step is almost the same as the calculations we have seen.

$$
\begin{aligned}
& \mathit{CSP} \\
= \ & \{ \text{ definition of } \mathit{CSP} \text{ (delayed variant) } \} \\
& (\mathit{min}_{\leq_w} \circ \mathit{costless}_C\lhd \circ \mathit{endWith}_t\lhd \circ \mathsf{P}\pi_1 \circ \Lambda(\mathit{next} \cup \textstyle\bigcup_{e\in E} \mathit{extendD}_e)*)\ ([], 0) \\
= \ & \{ \text{ let } \mathit{opt} = \pi_1 \circ \mathit{min}_{\leq_{w\circ\pi_1}} \circ (\mathit{endWith}_t \circ \pi_1)\lhd \ \} \\
& (\mathit{opt} \circ (\mathit{costless}_C \circ \pi_1)\lhd \circ \Lambda(\mathit{next} \cup \textstyle\bigcup_{e\in E} \mathit{extendD}_e)*)\ ([], 0) \\
= \ & \{ \text{ Theorem 2.22 } \} \\
& (\mathit{opt} \circ \Lambda(\mathit{next} \cup \textstyle\bigcup_{e\in E}(\mathit{extendD}_e \circ (\mathit{costless}_{C-c(e)} \circ \pi_1)?))*)\ ([], 0) \\
= \ & \{ \text{ Theorem 4.20, where } W^\ddagger = \leq_{w\circ\pi_1} \cap =_{(dst\times id)} \cap =_{c\circ\pi_1} \ \} \\
& (\mathit{opt} \circ (\mathit{thin}_{W^\ddagger} \circ \Lambda(\mathit{next} \cup \textstyle\bigcup_{e\in E}(\mathit{extendD}_e \circ (\mathit{costless}_{C-c(e)} \circ \pi_1)?) \circ \in))*)\ \{([], 0)\}
\end{aligned}
$$

Since we have confirmed the correctness of the delayed algorithm, Dijkstra-like search yields an efficient algorithm whose time complexity is $O(CV \log(CV) + CE)$. We can also use A^* algorithms.

Lastly, we would like to derive a bit more efficient algorithm. In the program above, a quasi-order $W^\ddagger = \leq_{w\circ\pi_1} \cap =_{(dst\times id)} \cap =_{c\circ\pi_1}$ is used to discard unnecessary candidates, which satisfies monotonicity condition guaranteed by Theorem 4.21. We can use a bit stronger order $W^{⁂} = \leq_{w\circ\pi_1} \cap =_{(dst\times id)} \cap \leq_{c\circ\pi_1}$, whose monotonicity condition can be proved by Lemmas 4.7 and Lemma 4.13. Since $x \leq_{cost} y$ implies $\mathit{costless}_{C-c(e)}(y) \Rightarrow \mathit{costless}_{C-c(e)}(x)$, $\mathit{costless}_{C-c(e)}$ is monotonic on $\leq_{cost}$; besides, $\mathit{addEdge}$ is monotonic on $\leq_{cost}$. Hence, the following algorithm is correct, which is more efficient than the previous one.

$$(\mathit{opt} \circ (\mathit{thin}_{W^{⁂}} \circ \Lambda(\mathit{next} \cup \bigcup_{e\in E}(\mathit{extendD}_e \circ (\mathit{costless}_{C-c(e)} \circ \pi_1)?) \circ \in))*)\ \{([], 0)\}$$

The algorithm above is almost the same as the one proposed by Desrochers and Soumis [DS88].

4.3 Summary and Discussions

In this chapter, we have developed calculational laws to derive efficient algorithms for combinatorial optimization problems. We have proposed calculational laws to construct and confirm monotonicity conditions. The keys to efficient algorithms are the use of functions and the incrementality condition. We have demonstrated effectiveness of our calculational laws through derivation of several algorithms including known efficient algorithms for shortest path problems and their variants.

We have mainly discussed derivations of dynamic programming algorithms. Since dynamic programming is one of the most important techniques for constructing efficient algorithms, there are many studies for formalizing it [Bel57,Iba73,Mor82,Hel89,

dM92, BdM96, Cur96, Cur97, KV06] and automatically deriving it [ALS91, BPT92, SHTO00, LS03, GMS04]. Most of them mention the importance of monotonicity conditions. If monotonicity condition holds, naive memoization yields a dynamic programming algorithm. However, most studies treat monotonicity conditions as an assumption. We concentrated to construct monotonicity conditions.

In Section 4.2, we derived algorithms for shortest path problems and their variants. Systematic derivation of graph algorithms is known to be difficult, and there were many studies about this issue, for example [MR93, BvdEvG94, SHT98, Rav99a, Dur02, MHT07]. One of our purposes is to provide a calculational law that can derive graph algorithms.

Our work is highly motivated by the works about *maximum marking problems* [ALS91, BPT92, SHTO00, Bir01]. We aimed at generalizing them so that they can cope with more problems, such as problems on graphs. We pointed out that the incrementality condition is the key to dynamic programming algorithms, and generalize the results so as to deal with even graph iterating problems.

We have intensively used equivalence relations to derive monotonicity condition and derived dynamic programming algorithms. Greedy algorithms are usually more efficient than dynamic programming algorithms, and derivation of greedy algorithms is an important topic. At a glance, we can derive greedy algorithms if we obtain monotonicity condition without breaking totality of quasi-orders, and our calculational laws are also helpful for this purpose. However, since we have concentrated on monotonicity conditions, it is only better-local algorithms [Cur96, Cur03], in which better partial solutions yield better ones in each step, that we can derive on our laws. There are other kinds of greedy algorithms, called best-local algorithms (the best partial solution yields the best one in each step) and best-global algorithms (the greedy strategy finally yields the best solution, while partial solutions may not be the best), which cannot be characterized by monotonicity conditions. Therefore, it is interesting further direction to formalize effective methods to derive greedy algorithms.

4.4 Supplemental Lemmas

Lemma 4.24 (Proposition 9.3 of [BdM96])**.** Given a relator F, a total function f, a functions g and h, and a relation R, assume that there exists a function h' such that both of $f \circ h = h' \circ \mathsf{F}(f \vartriangle g)$ and $R \circ h' \supseteq h' \circ \mathsf{F}(R \times id)$ hold. Then, $R_f \circ h \supseteq h \circ \mathsf{F}(R_f \cap =_g)$ holds. □

Lemma 4.25 (Exercise 7.8 of [BdM96])**.**

$$R \cap S^\circ \supseteq min_R \circ min_S^\circ$$

Proof.

$$
\begin{array}{rcl}
min_R \circ min_S^\circ & = & \{ \text{ trivial } (\cap) \ \} \\
& & (min_R \circ min_S^\circ) \cap (min_R \circ min_S^\circ) \\
& \subseteq & \{ \ min_R \subseteq \in \ \} \\
& & (min_R \circ \ni) \cap (min_S \circ \ni)^\circ \\
& = & \{ \text{ definition of } min \ \} \\
& & ((\in \cap R/\ni) \circ \ni) \cap ((\in \cap S/\ni) \circ \ni)^\circ \\
& \subseteq & \{ \text{ trivial } (\cap) \ \} \\
& & (R/\ni \circ \ni) \cap (S/\ni \circ \ni)^\circ \\
& \subseteq & \{ \text{ property of } / \ \} \\
& & R \cap S^\circ
\end{array}
$$

□

Lemma 4.26. For any relations R and S, $\overset{R;S}{<}$ is equivalent to $\overset{R}{<} ; S$.

Proof.

$$
\begin{array}{rcl}
\overset{R;S}{<} & = & \{ \text{ definition of } \overset{R;S}{<} \ \} \\
& & (S \cap (\overline{S^\circ} \cup R)) \cap (\overline{S^\circ} \cup (S \cap \overline{R^\circ})) \\
& = & \{ \text{ distributivity } \} \\
& & S \cap (\overline{S^\circ} \cup (R \cap S \cap \overline{R^\circ})) \\
& = & \{ \text{ simplification } \} \\
& & S \cap (\overline{S^\circ} \cup (R \cap \overline{R^\circ})) \\
& = & \{ \text{ definition of } \overset{R}{<} \text{ and } \Rightarrow \ \} \\
& & S \cap (S^\circ \Rightarrow \overset{R}{<}) \\
& = & \{ \text{ definition of } \overset{R}{<} ; S \ \} \\
& & \overset{R}{<} ; S
\end{array}
$$

□

Lemma 4.27. For a polynomial functor F and relations R and S, $\overset{\mathsf{F}(R;S)}{<} \subseteq \overset{\mathsf{F}S}{<} \cup (\mathsf{F}S \cap \overset{\mathsf{F}R}{<})$ holds.

Proof.

$$
\begin{array}{rcl}
\overset{\mathsf{F}(R;S)}{<} & = & \{ \text{ definition } \} \\
& & \mathsf{F}(S \cap (S^\circ \Rightarrow R)) \cap \overline{\mathsf{F}(S^\circ \cap (S \Rightarrow R^\circ))} \\
& \subseteq & \{ \ \mathsf{F} \text{ is polynomial, and Lemmas 2.7 and 2.9 } \} \\
& & \mathsf{F}S \cap (\mathsf{F}S^\circ \Rightarrow \mathsf{F}R) \cap (\overline{\mathsf{F}S^\circ} \cup \overline{\mathsf{F}(S \Rightarrow R^\circ)}) \\
& \subseteq & \{ \ (S \Rightarrow R^\circ) \supseteq R^\circ \ \} \\
& & \mathsf{F}S \cap (\mathsf{F}S^\circ \Rightarrow \mathsf{F}R) \cap (\overline{\mathsf{F}S^\circ} \cup \overline{\mathsf{F}R^\circ}) \\
& = & \{ \text{ simplify } \} \\
& & (\mathsf{F}S \cap \overline{\mathsf{F}S^\circ}) \cup (\mathsf{F}S \cap \mathsf{F}R \cap \overline{\mathsf{F}R^\circ}) \\
& = & \{ \text{ definition of the strict part } \} \\
& & \overset{\mathsf{F}S}{<} \cup (\mathsf{F}S \cap \overset{\mathsf{F}R}{<})
\end{array}
$$

□

Lemma 4.28. Let $\preceq$ be the linear order on which *True* is strictly smaller than *False*. Then, for any quasi-order R and predicate p, the following equation holds.

$$min_R \circ p\triangleleft = p? \circ min_{R;\preceq_p}$$

Proof. Prepare two sets X and Y such that $p\lhd(X) = \emptyset$ and $p\lhd(Y) = Y$. Then, it is sufficient to show the following equation.

$$\Lambda min_R \circ p\lhd(X \cup Y) = \Lambda(p? \circ min_{R;\preceq_p})(X \cup Y)$$

When Y is the empty set, then both sides yield the empty sets. When Y is not empty, the left hand side yields $\Lambda min_R(Y)$, and from the definition of $R ; \preceq_p$, the right hand side also yields $\Lambda min_R(Y)$. □

Lemma 4.29. For any quasi-orders R and S, $R ; S \supseteq R \cap \overset{S}{\equiv}$ holds.

Proof.

$$\begin{aligned}
R ; S \supseteq R \cap \overset{S}{\equiv} \quad \Leftrightarrow \quad & \{ \text{ definition of } R ; S \text{ and } \overset{S}{\equiv} \} \\
& S \cap (S^\circ \Rightarrow R) \supseteq R \cap S \cap S^\circ \\
\Leftarrow \quad & \{ S^\circ \Rightarrow R \supseteq R \} \\
& S \cap R \supseteq R \cap S \cap S^\circ \\
\Leftarrow \quad & \{ \text{ trivial } (\cap) \} \\
& \mathit{True}
\end{aligned}$$

□

Lemma 4.30. For relations R and S, $mnl_R \circ \Lambda mnl_{R;S} = mnl_{R;S}$ holds if $\overline{S} \cap \overline{S^\circ} \subseteq R^\circ \Rightarrow R$ holds.

Proof. It is evident that $mnl_R \circ \Lambda mnl_{R;S} \subseteq mnl_{R;S}$ holds; thus we prove $mnl_R \circ \Lambda mnl_{R;S} \supseteq mnl_{R;S}$.

$$\begin{aligned}
& mnl_R \circ \Lambda mnl_{R;S} \supseteq mnl_{R;S} \\
\Leftrightarrow \quad & \{ \text{ Lemma 3.24 } \} \\
& mnl_{R;S} \cap (R^\circ \Rightarrow R)/mnl^\circ_{R;S} \supseteq mnl_{R;S} \\
\Leftrightarrow \quad & \{ \text{ trivial } (\cap) \} \\
& (R^\circ \Rightarrow R)/mnl^\circ_{R;S} \supseteq mnl_{R;S} \\
\Leftrightarrow \quad & \{ \text{ property of } / \} \\
& R^\circ \Rightarrow R \supseteq mnl_{R;S} \circ mnl^\circ_{R;S} \\
\Leftarrow \quad & \{ \text{ Lemma 4.25 } \} \\
& R^\circ \Rightarrow R \supseteq ((R ; S)^\circ \Rightarrow (R ; S)) \cap ((R ; S)^\circ \Rightarrow (R ; S))^\circ \\
\Leftrightarrow \quad & \{ \text{ definition of } ; \text{ and } \Rightarrow \} \\
& R^\circ \Rightarrow R \supseteq (\overline{S^\circ} \cup (S \cap \overline{R^\circ}) \cup (S \cap (\overline{S^\circ} \cup R))) \cap (\overline{S} \cup (S^\circ \cap \overline{R}) \cup (S^\circ \cap (\overline{S} \cup R^\circ))) \\
\Leftrightarrow \quad & \{ \text{ simplification } \} \\
& R^\circ \Rightarrow R \supseteq (\overline{S^\circ} \cap \overline{S}) \cup (S \cap S^\circ \cap (\overline{R^\circ} \cup R) \cap (\overline{R} \cup R^\circ)) \\
\Leftrightarrow \quad & \{ \text{ property of } \cup \} \\
& (R^\circ \Rightarrow R \supseteq \overline{S^\circ} \cap \overline{S}) \wedge (R^\circ \Rightarrow R \supseteq S \cap S^\circ \cap (\overline{R^\circ} \cup R) \cap (\overline{R} \cup R^\circ)) \\
\Leftrightarrow \quad & \{ \text{ premise and trivial } (\cap) \} \\
& \mathit{True}
\end{aligned}$$

□

Lemma 4.31. For well-supported quasi-orders R and S, the following equation holds if $a\, R\, b \vee b\, R\, a$ implies $a\, S\, b \vee b\, S\, a$.

$$\Lambda mnl_R \circ \Lambda mnl_S = \Lambda mnl_{R;S}$$

Proof. From Lemma 4.30, $\Lambda mnl_R \circ \Lambda mnl_{R;S} = \Lambda mnl_{R;S}$ holds, because the premise of Lemma 4.30 certainly holds: $R^\circ \Rightarrow R \supseteq \overline{R^\circ} \cap \overline{R} \supseteq \overline{S^\circ} \cap \overline{S}$. Therefore, from Lemma 3.28, it is sufficient to prove that $\Lambda mnl_S(X) \supseteq \Lambda mnl_{R;S}(X)$ holds for all set X. From Lemma 3.25, it is sufficient to prove $(S^\circ \Rightarrow S) \supseteq ((R\,;S)^\circ \Rightarrow (R\,;S))$, which is proved as follows.

$$
\begin{array}{ll}
\multicolumn{2}{l}{(S^\circ \Rightarrow S) \supseteq ((R\,;S)^\circ \Rightarrow (R\,;S))} \\
\Leftrightarrow & \{ \text{ definition of } ; \text{ and } \Rightarrow \} \\
& \overline{S^\circ} \cup S \supseteq (\overline{S^\circ} \cup (S \cap \overline{R^\circ})) \cup (S \cap (\overline{S^\circ} \cup R)) \\
\Leftrightarrow & \{ \text{ property of } \cup \} \\
& (\overline{S^\circ} \cup S \supseteq \overline{S^\circ}) \wedge (\overline{S^\circ} \cup S \supseteq S \cap \overline{R^\circ}) \wedge (\overline{S^\circ} \cup S \supseteq S \cap (\overline{S^\circ} \cup R)) \\
\Leftrightarrow & \{ \text{ trivial } (\cap \text{ and } \cup) \} \\
& \mathit{True}
\end{array}
$$

□

Chapter 5

A Generic Framework for Optimal Path Querying

Imagine we are planning a trip to a historic city, in which we intend to look round famous sights. How can we find the best route for strolling the city? Finding the shortest route would not be very difficult, because well-known algorithms for shortest path problems will be applicable. However, what we truly want will not be the shortest route in practice: we might want to find a route whose transportation expense is less than a certain limit; we might want to take a rest at a certain cafeteria on afternoon; we might want to walk for a while to enjoy scenery; we might walk slower than usual after visiting a certain temple on a hill, etc. After all, it is a complicated problem—how can we find the best route?

The objective of this chapter is to provide a generic framework for *optimal path querying*. We intend to find the optimal route that is identified by a given criterion of optimality. Optimal path queries are important from both theoretical and practical aspects. In theory, a lot of optimization problems result in routing problems, such as shortest path problems and their variants. In practice, optimal path queries have a lot of practical applications. The trip-planning problem described above is a direct application. To provide a routing with quality-of-service guarantees is another application [KK01], where optimal path queries are necessary to find a routing that satisfies additional requirements such as band width and latency. Querying graph-structured databases is also an application of optimal path queries. While regular path queries [MW95, FFG06] are known to be useful methods, infinite number of paths may match a query and it would be necessary to extract the optimal one such as the smallest one.

Since optimal path queries are important, there are a lot of studies on this topic from algorithmic viewpoints [Jok66, DS88, Rom88, Pun91, BJM00, BJ04, VD05, SJH06, CZ07]. However, it is difficult for nonspecialists to utilize such studies for their objective. Recall the trip-planning problem. Even finding the shortest route via given sights is a nontrivial problem, and it is an instance of regular-language-

The result of this chapter was published as [MMHT08a].

constrained shortest path problems [Rom88]. The first additional requirement, the expense must be less than a limit, is an instance of resource-constrained shortest path problems [Jok66]. The second one, take a rest at the cafeteria on afternoon, is a combination of time-window constraints [DS88] and regular-language constraints. It is not easy to find an existing algorithm that copes with a requirement; furthermore, we need to deal with combinations of many requirements.

In the previous chapter, we demonstrated derivations of efficient algorithms for shortest path problems, regular-language-constrained shortest path problems, and resource-constrained shortest path problems. We demonstrated our calculational laws bring uniform derivations of algorithms for these problems. In this chapter, we propose a generic framework for optimal path queries based on the derivation. We propose a domain-specific language (in short, DSL) to describe optimal path queries, together with an algorithm to find an optimal path specified in our language.

5.1 Designing a Domain-Specific Language for Optimal Path Querying

Here, let us discuss the design of our domain-specific language for optimal path querying. As summarized in the introduction, two important requirements for our DSL are effectiveness and expressiveness: the DSL should be evaluated efficiently and can be express many practical optimal path queries.

First, let us consider expressiveness. As outlined as the trip planning problem, we would like to specify a criterion of optimality not only by requirements for paths but also by the cost of a path. Such flexibility is also important when we consider to reduce other problems into optimal path problems. For this reason, we adopt recursive functions for describing criteria of optimality. We use two kinds of recursive functions, numeric-valued functions and boolean-valued functions, because we will express both costs and requirements by recursive functions. We do not adopt other formal frameworks, such as regular expressions, because they may not be useful to describe costs of paths, while they will be useful to specify requirements; besides, it is usually not difficult to translate them into recursive functions.

Next, let us consider effectiveness. As demonstrated in the derivations in the previous section, the key to efficient algorithms is the way how we specify crucial case analyses. Our calculational laws, which are useful for specifying proper case analyses, indicate that a key to appropriate case analyses is incrementality condition. In this case, since we can enumerate all candidates (paths) by recursively extending paths, incrementality condition naturally holds if a criterion of optimality is specified by recursive functions on a path.

In addition to incrementality condition, we should guarantee monotonicity condition for the primitive case. Recall that our calculational laws construct complicated monotonic conditions from simple and primitive ones. Thus, the primitive monotonicity condition that constitute the "core" of complicated cases is necessary. In

$prog$	$::=$	minimize $h(x)$ subject to $p(x)$ where $decl \cdots decl$	{ program }
$decl$	$::=$	$p(\epsilon) = b;\ p(x +\!\!+ [a]) = \phi;$	{ boolean-valued function }
	$\mid$	$f(\epsilon) = n;\ f(x +\!\!+ [a]) = e;$	{ integer-valued function }
	$\mid$	$h(\epsilon) = n;\ h(x +\!\!+ [a]) = e';$	{ objective function }
ϕ	$::=$	$b \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid p(x) \mid q(a) \mid e \leq n$	{ boolean-valued expression }
e	$::=$	$n \mid e \oplus e \mid f(x) \mid h(x) \mid w(a) \mid$ if ϕ then e else e	{ integer-valued expression }
e'	$::=$	$n \mid e' \oplus e' \mid h(x) \mid w(a) \mid$ if ϕ then e' else e'	{ body of objective function }
$\oplus$	$::=$	$+ \mid \times \mid \uparrow \mid \downarrow$	{ arithmetic operator }

Figure 5.1. Syntax of our query language, where $b \in Bool$ and $n \in \mathbb{Z}_+$ are constant values, $q : E \to Bool$ is an atomic predicate, $w : E \to \mathbb{Z}_+$ is a weight function, and a and x are respectively variables of type E and E^*.

this case, we make use of the fact that usual arithmetic operators, namely $+$, $\times$, $\uparrow$, and $\downarrow$, are monotonic on $\leq$ in the sense that $a \geq b$ implies $a \oplus e \geq b \oplus e$ when a, b, and e are nonnegative, where $\oplus$ is the operator.

Furthermore, we need to guarantee another property. Recall that our calculational laws will introduce relations like $R \cap =_f$, which means that candidates having different f values are not compared. Therefore, if the range of f is infinite, in other words we will consider infinite number of cases, then the termination of the resulted algorithm is generally not guaranteed, especially when the number of candidates is infinite. Actually optimal path querying is the case. There are infinite number of paths, and thus, guaranteeing finiteness of f is the matter of termination of querying algorithms. While boolean-valued functions naturally have finite range, ranges of numeric-valued functions are naturally infinite and problematic. We solve this problem by restricting the syntax of our DSL a bit.

5.2 Language for Optimal Path Querying

We treat $G = (V, E)$ as the underlying graph in the rest of this chapter.

5.2.1 Language Description

Figure 5.1 shows the syntax of our query language. At the top of a program, we specify the objective function and the requirement for feasible paths. Both objective functions and requirements are expressed by using recursive functions on a path. There are two kinds of recursive functions: boolean-valued and nonnegative-integer-valued functions. Declarations of boolean-valued functions consist of constants, basic boolean operations, function calls, atomic predicates, and inequalities. Right-hand-side expressions of inequalities must be constant nonnegative integers, which is the key restriction to guarantee termination of our querying algorithm. Decla-

rations of integer-valued functions consist of constants, additions, multiplications, minimum and maximum operations, function calls, weights of edges, and conditional expressions. Note that constant values and values of weight functions should be nonnegative, and thus, the range of each integer-valued function is nonnegative. The objective function is an integer-valued function that must not call other functions outside operands of inequalities. Atomic predicates and weights are predefined, and distinguished from recursive functions by types of their arguments.

The semantics is straightforward. Given an objective function h, a requirement p, and a graph G, the program results in one of the minimum-h-valued paths in G among those satisfying p.

In the declaration of each recursive function, we assume that the size of recursive parameters must decrease, such as $f(x \mathbin{+\!\!+} [a]) = \cdots g(x) \cdots h(x) \cdots$; hence all recursive functions trivially terminate. Declarations like $f(x \mathbin{+\!\!+} [a]) = \cdots g(x \mathbin{+\!\!+} [a]) \cdots$ are prohibited, and we should unfold $g(x \mathbin{+\!\!+} [a])$. Meanwhile, we will use such declarations as a syntactic sugar. Declarations like $f(x \mathbin{+\!\!+} [a] \mathbin{+\!\!+} [b]) = \cdots g(x) \cdots$ are also prohibited, and we should use an auxiliary function: $f(x' \mathbin{+\!\!+} [b]) = \cdots f'(x') \cdots ; \cdots ; f'(x \mathbin{+\!\!+} [a]) = \cdots g(x) \cdots$. In addition, we impose the following assumption, which is required to utilize Dijkstra algorithm.

Requirement 5.1. For an objective function h and any path $x \mathbin{+\!\!+} [a]$, $h(x) \leq h(x \mathbin{+\!\!+} [a])$ holds. □

It is not difficult to confirm Requirement 5.1 in a sound manner. Given the objective function h, the following function *check* takes the description of its recursive case and returns *True* only if it satisfies Requirement 5.1.

$$
\begin{array}{lcl}
\mathit{check}[\![h(x \mathbin{+\!\!+} [a]) = e';]\!] & \stackrel{\text{def}}{=} & \mathit{chk}[\![e']\!] \\
\quad \mathit{chk}[\![n]\!] & \stackrel{\text{def}}{=} & \mathit{False} \\
\quad \mathit{chk}[\![e'_1 + e'_2]\!] & \stackrel{\text{def}}{=} & \mathit{chk}[\![e'_1]\!] \vee \mathit{chk}[\![e'_2]\!] \\
\quad \mathit{chk}[\![e'_1 \times e'_2]\!] & \stackrel{\text{def}}{=} & \mathit{chk}[\![e'_1]\!] \wedge \mathit{chk}[\![e'_2]\!] \\
\quad \mathit{chk}[\![e'_1 \uparrow e'_2]\!] & \stackrel{\text{def}}{=} & \mathit{chk}[\![e'_1]\!] \vee \mathit{chk}[\![e'_2]\!] \\
\quad \mathit{chk}[\![e'_1 \downarrow e'_2]\!] & \stackrel{\text{def}}{=} & \mathit{chk}[\![e'_1]\!] \wedge \mathit{chk}[\![e'_2]\!] \\
\quad \mathit{chk}[\![h(x)]\!] & \stackrel{\text{def}}{=} & \mathit{True} \\
\quad \mathit{chk}[\![w(a)]\!] & \stackrel{\text{def}}{=} & \mathit{False} \\
\quad \mathit{chk}[\![\textsf{if } \phi \textsf{ then } e'_1 \textsf{ else } e'_2]\!] & \stackrel{\text{def}}{=} & \mathit{chk}[\![e'_1]\!] \wedge \mathit{chk}[\![e'_2]\!]
\end{array}
$$

Recall that all weights and constant integers are nonnegative; therefore, for the case of addition, it is sufficient to check that either of operand contains a recursive call. For the case of multiplication, the possibility of a multiplication with 0 is problematic, and thus, we check that both operands contain a recursive call. Note that $y \geq x \wedge z \geq x$ implies $y \times z \geq x$ for any $x \in \mathbb{Z}_+$. The other cases are similar.

5.2.2 Writing Optimal Path Queries by the Language

Shortest Path Problems with Transit Costs

As an example, let us consider a shortest path problem with transit costs. The objective of the problem is to find the lowest-cost route from the starting point to the destination, where we should pay an additional cost to ride on a train. The following is a description of the problem in our language, where the starting point is s, the destination is t, and the additional cost is C.

$$
\begin{array}{l}
\textsf{minimize } \mathit{cost}(x) \textsf{ subject to } \mathit{constraint}(x) \textsf{ where} \\
\quad \mathit{constraint}([\,]) = \mathit{False}; \\
\quad \mathit{constraint}(x +\!\!+ [a]) = \mathit{start}_s(x +\!\!+ [a]) \wedge \mathit{to}_t(a); \\
\quad \mathit{start}_s([\,]) = \mathit{False}; \\
\quad \mathit{start}_s(x +\!\!+ [a]) = \mathit{start}_s(x) \vee (\mathit{empty}(x) \wedge \mathit{from}_s(a)); \\
\quad \mathit{empty}([\,]) = \mathit{True}; \\
\quad \mathit{empty}(x +\!\!+ [a]) = \mathit{False}; \\
\quad \mathit{walk}([\,]) = \mathit{True}; \\
\quad \mathit{walk}(x +\!\!+ [a]) = \neg \mathit{train}(a); \\
\quad \mathit{cost}([\,]) = 0; \\
\quad \mathit{cost}(x +\!\!+ [a]) = \mathit{cost}(x) + w(a) + (\textsf{if } \mathit{walk}(x) \wedge \mathit{train}(a) \textsf{ then } C \textsf{ else } 0)
\end{array}
$$

In the description, w is a weight function, and the functions from_s, to_t, and train are atomic predicates. The definitions of from_s and to_t are $\mathit{from}_s(e) \stackrel{\text{def}}{=} (hd(e) = s)$ and $\mathit{to}_t(e) \stackrel{\text{def}}{=} (tl(e) = t)$, respectively. The predicate *train* checks whether we are riding on a train.

The requirement for solutions is checked by the predicate *constraint*. It is worth noting that use of our language is not restricted to point-to-point optimal path problems and it is necessary to examine two endpoints of paths explicitly. Actually *constraint* does so. The starting point is examined by the recursive function start_s, which checks whether a path starts from s by using an auxiliary function *empty*, and the endpoint is checked by the atomic predicate to_t.

The objective function for this problem is *cost*, which is the characteristic part of this problem. The function *cost* uses a recursive function *walk* to check the necessity of transit costs. The function *walk* checks whether we have not been riding on a train.

Length-Constrained Shortest Path Problems

Next, let us consider a length-constrained shortest path problem, in which the objective is to find the shortest path that consists of fewer edges than a given limit. The following is a description of a length-constrained shortest path problem, where

the starting point is s, the destination is t, and the limit is K.

$$
\begin{array}{l}
\text{minimize } wsum(x) \text{ subject to } constraint(x) \text{ where} \\
\quad constraint([\,]) = False; \\
\quad constraint(x +\!\!+ [a]) = start_s(x +\!\!+ [a]) \wedge to_t(a) \wedge (len(x +\!\!+ [a]) \leq K); \\
\quad start_s([\,]) = False; \\
\quad start_s(x +\!\!+ [a]) = start_s(x) \vee (empty(x) \wedge from_s(a)); \\
\quad empty([\,]) = True; \\
\quad empty(x +\!\!+ [a]) = False; \\
\quad wsum([\,]) = 0; \\
\quad wsum(x +\!\!+ [a]) = wsum(x) + w(a); \\
\quad len([\,]) = 0; \\
\quad len(x +\!\!+ [a]) = len(x) + 1;
\end{array}
$$

The recursive functions $start_s$ and $empty$ are the same as the previous example. The integer-valued function len computes the number of edges used. The objective function $wsum$ computes the summation of weights of edges.

Expressiveness of the Language

While our language is simple, it can express a large number of known problems. Here we enumerate some of them. It is worth noting that their combinations can be expressed in our language. We assume only nonnegative integers are used in descriptions. For a large number of problems, negative weights of edges can be removed in safe by re-weighting method by Johnson [Joh77].

Fact 5.2. The following problems can be expressed in our language: point-to-point shortest path problems; resource-constrained shortest path problems [Jok66] (find the minimum-cost path whose weight is less than a given limit); shortest path problems with time windows [DS88] (find the shortest path where each vertex can be used only during its time window); regular-language-constrained shortest path problems [Rom88] (find the shortest path whose label is in a given regular language); time-table queries [BJ04] (regular-language-constrained shortest path problems where weights of edges depend on time); shortest path problems with forbidden paths [VD05] (find the shortest path that does not contain given forbidden paths); approach-dependent, time-dependent, label-constrained shortest path problems [SJH06] (time-table queries where weights of edges depend on their preceding vertexes). □

Since recursive functions are available, it is not difficult to describe these problems in our language. For example, regular-language-constrained shortest path problems can be expressed in our language based on the binary encoding technique: arrange a set of boolean-valued functions to simulate the transition function of the DFA representing the given regular language.

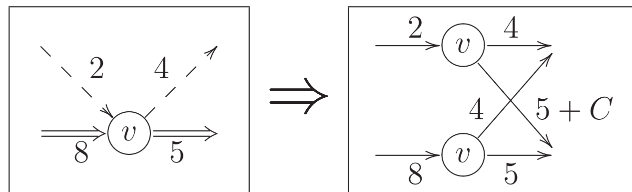

Figure 5.2. Reducing the shortest path problem with transit costs into a usual shortest path problem. Double-lines arrows stand for edges of riding on a train, broken arrows stand for edges of walking, and C is the additional cost for a transit.

While our language is expressive, there are several problems that cannot be dealt with. For example, we cannot describe maximization problems such as maximum capacity path problems [Pun91]. Context-free-language-constrained shortest path problems [BJM00] (find the shortest path whose label is in a given context-free language) are also out of its domain, because recursive functions must traverse over a path in the fixed direction in our language.

5.3 Deriving Optimal Path Querying Algorithm

In this section, we introduce our optimal path querying algorithm with some improvements. Before introducing our algorithm, we would like to outline our derivation from an algorithmic viewpoint rather than the calculational viewpoint for providing intuitive understanding.

Our derivation can be seen as reduction of optimal path problems into minimum-weighted path problems. As an example, consider the shortest path problem with transit costs, which we have seen in Section 5.2. Notice that for determining optimality, it is necessary to distinguish whether we are riding on a train or not, because it affects the cost of paths. That distinction naturally raises a case analysis, and based on this case analysis, we can reduce the problem into a shortest path problem, as shown in Figure 5.2. To distinguish these two cases, we divide the vertex v into two: one corresponds to the case of riding on a train, and the other corresponds to the case of not riding on a train. Then, the shortest path on the right graph is the optimal path on the left one.

Although this approach is applicable for a large set of problems, there are two issues. The first issue is correctness. How can we provide an appropriate reduction, or, equivalently, how can we specify appropriate case analyses? Here, our calculational laws are helpful. As discussed, incrementality condition implies successful case analyses, and our DSL is designed so that incrementality condition is fulfilled. The second issue is efficiency. In general, the reduction makes the underlying graph much larger and derives practically inefficient algorithms. We solve the second issue by "on-the-fly" evaluation, that is, we do not really construct the larger graph.

In the following, we regard each expression in the description as a function that

takes a path and returns a value. In addition, we will use the following notations. For integer-valued functions f and g in a query description, $f \rightsquigarrow g$ denotes that the definition of g syntactically contains function calls of f in its non-predicate part. For example, $f \rightsquigarrow g$ holds when g is defined by $g(x +\!\!+ [a]) =$ $\mathsf{if}\ \cdots\ \mathsf{then}\ \cdots f(x) \cdots\ \mathsf{else}\ \cdots$, while $f \rightsquigarrow g$ may not hold when g is defined by $g(x +\!\!+ [a]) = \mathsf{if}\ (f(x) \leq n)\ \mathsf{then}\ \cdots\ \mathsf{else}\ \cdots$. The transitive closure of $\rightsquigarrow$ is denoted by $\overset{+}{\rightsquigarrow}$. Intuitively $f \overset{+}{\rightsquigarrow} g$ means a result of g would contain a result of f. We use the same notation for integer-valued expressions by regarding them as integer-valued functions.

5.3.1 Deriving Case-Analysing Function

Now, let us derive our optimal path querying algorithm. The key to efficient querying algorithm is to specify a function that represents successful case analyses. In other words, given the objective function h and the constraint p, we would like to find a function g such that (i) $\leq_h \cap =_g$ satisfies monotone condition, and (ii) $g(x) = g(y)$ implies $p(x) = p(y)$. Note that incrementality condition of g is important to guarantee the first requirement.

Let $\{p_0, p_1, \ldots, p_m\}$ and $\{f_0, f_1, \ldots, f_n\}$ be respectively the whole set of boolean-valued functions and integer-valued functions (including the objective function) in the query description. Consider a function g defined as follows.

$$g(x) \overset{\text{def}}{=} (f_0(x), f_1(x), \ldots, f_n(x), p_0(x), p_1(x), \ldots, p_m(x))$$

The function g is a candidate of the function to describe case analyses because of the following two reason: first, g satisfies incrementality condition because g is a recursive function (or, more specifically, a catamorphism) on a path; second, g retains sufficient information to determine the optimality of a path in the sense that the two requirement above are fulfilled. However, the range of g may be infinite. As mentioned, finiteness of cases is necessary to guarantee the termination of derived algorithm.

To resolve this problem, we set a cut-off to each integer-valued function. For each integer-valued function f, we define its cut-off version $\hat{f} : E^* \to (\mathbb{Z}_+ \cup \{\infty\})$ as follows, where a set I stands for all inequalities in the description and we interpret $\max(\emptyset)$ as $-\infty$.

$$\hat{f}(x) \overset{\text{def}}{=} \begin{cases} f(x) & \text{if } f(x) \leq \max\{n \mid (e \leq n) \in I \wedge f \overset{+}{\rightsquigarrow} e\} \\ \infty & \text{otherwise} \end{cases}$$

The value of $\hat{f}$ is the same as that of f up to the boundary and turns into ∞ when it exceeds the boundary. The boundary for $\hat{f}$ is the maximum right-hand-side value of the inequality whose left-hand-side value would include some $\hat{f}$ value. Although the function $\hat{f}$ forgets some information of f, it is not a problem: when values that are

larger than boundaries are used, integer-valued expressions result in larger values and inequalities yield *False*; therefore, their exact values are unnecessary.

By using the cut-off values, we can obtain a proper case-analyzing function *state* as follows.

$$\mathit{state}(x) \stackrel{\text{def}}{=} (\hat{f}_0(x), \hat{f}_1(x), \ldots, \hat{f}_n(x), p_0(x), p_1(x), \ldots, p_m(x))$$

As required, the range of the function *state* is finite. The appropriateness of *state* is verified by the following lemmas.

Lemma 5.3. For each boolean-valued expression ϕ in the description, $\phi(x +\!\!+ [a]) = \phi(y +\!\!+ [a])$ holds if $\mathit{state}(x) = \mathit{state}(y)$ holds.

Proof. Inequalities are the only nontrivial construction. We will prove it by contradiction.

Assume that $e(x +\!\!+ [a]) \leq n < e(y +\!\!+ [a])$ holds for an inequality "$e \leq n$". Then, there exists an integer-valued function $g \rightsquigarrow e$ such that $g(x) < g(y)$ holds. Since $\mathit{state}(x) = \mathit{state}(y)$ holds, $g(x) < g(y)$ implies $\hat{g}(x) = \hat{g}(y) = \infty$. From the definition of $\hat{g}$ and the fact $g \rightsquigarrow e$, $\hat{g}(x) = \infty$ implies $g(x) > n$, which contradicts $e(x +\!\!+ [a]) \leq n$. □

Lemma 5.4. If $\mathit{state}(x) = \mathit{state}(y)$ holds for two sequences x and y, then $\mathit{state}(x +\!\!+ [a]) = \mathit{state}(y +\!\!+ [a])$ holds for any edge a.

Proof. From Lemma 5.3, $p(x +\!\!+ [a]) = p(y +\!\!+ [a])$ holds for all boolean-valued functions p in a description; thus, it is sufficient to show $\hat{f}(x +\!\!+ [a]) = \hat{f}(y +\!\!+ [a])$ holds for all integer-valued functions f in a description. We will prove it by contradiction.

Assume that $\hat{f}(x +\!\!+ [a]) < \hat{f}(y +\!\!+ [a])$ holds. From Lemma 5.3, the same branches are chosen at all conditional expressions in the computations of $\hat{f}(x +\!\!+ [a])$ and $\hat{f}(y +\!\!+ [a])$. Therefore, there exists an integer-valued function $g \rightsquigarrow f$ such that $g(x) < g(y)$ and g is certainly called when $\hat{f}(x +\!\!+ [a])$ and $\hat{f}(y +\!\!+ [a])$ are evaluated. Since $\mathit{state}(x) = \mathit{state}(y)$ holds, $g(x) < g(y)$ implies $\hat{g}(x) = \hat{g}(y) = \infty$. Now let u be the boundary used in $\hat{f}$; then, $\hat{g}(x) = \infty$ implies $g(x) > u$, because $f \stackrel{+}{\rightsquigarrow} e$ implies $g \stackrel{+}{\rightsquigarrow} e$ for any expression e. However, $g(x) \leq f(x +\!\!+ [a])$ holds from construction of f, which implies $u < f(x +\!\!+ [a])$. In summary, $\hat{f}(x +\!\!+ [a]) = \infty$ holds and it contradicts $\hat{f}(x +\!\!+ [a]) < \hat{f}(y +\!\!+ [a])$. □

Lemma 5.5. For an edge a and two paths x and y such that $\mathit{state}(x) = \mathit{state}(y)$ and $\mathit{dst}(x) = \mathit{dst}(y)$ hold, $x +\!\!+ [a]$ is feasible if and only if $y +\!\!+ [a]$ is feasible.

Proof. Since $\mathit{dst}(x) = \mathit{dst}(y)$ holds, $x +\!\!+ [a]$ is a path if and only if $y +\!\!+ [a]$ is a path. Moreover, $\mathit{state}(x) = \mathit{state}(y)$ holds; hence, from Lemma 5.3, $x +\!\!+ [a]$ is feasible if and only if $y +\!\!+ [a]$ is feasible. □

Lemma 5.6. For the objective function h and two sequences x and y, assume that both $\mathit{state}(x) = \mathit{state}(y)$ and $h(x) \leq h(y)$ hold. Then, for any edge a, $h(x +\!\!+ [a]) \leq h(y +\!\!+ [a])$ holds. □

Proof. The value of $h(x \mathbin{+\!\!+} [a])$ is determined from the value of recursive function calls and values determined from a. Recall that the body of h includes no function calls except for that of h. Moreover, from Lemma 5.3, the same branches are chosen to evaluate $h(x \mathbin{+\!\!+} [a])$ and $h(y \mathbin{+\!\!+} [a])$. Now the difference of $h(x \mathbin{+\!\!+} [a])$ and $h(y \mathbin{+\!\!+} [a])$ comes from only the value of $h(x)$ and $h(y)$. Since $h(x) \leq h(y)$ holds from an assumption, we can conclude $h(x \mathbin{+\!\!+} [a]) \leq h(y \mathbin{+\!\!+} [a])$ from construction of h. □

Lemmas 5.4, 5.5, and 5.6 tell us that it is sufficient to retain the minimum-weighted path among those having the same *state*-value. Given the objective function h, for any paths x and y such that all of $state(x) = state(y)$, $dst(x) = dst(y)$, and $h(x) \leq h(y)$ hold, $y \mathbin{+\!\!+} z$ is a feasible path only if $x \mathbin{+\!\!+} z$ is feasible; moreover, $h(x \mathbin{+\!\!+} z) \leq h(y \mathbin{+\!\!+} z)$ holds. From the viewpoint of our calculational laws, Lemma 5.4 corresponds to the incrementality condition, Lemma 5.5 means that the constraint function is more collapsing than *state* with *dst*, and 5.6 provides the proof of monotonicity condition. The key point is to reduce the number of cases to finite without breaking incrementality condition.

5.3.2 Optimal Path Querying Algorithm

To simplify our optimal path querying algorithm, we prepare an auxiliary function *pstate* defined as follows.

$$pstate(x) \stackrel{\text{def}}{=} (state(x), dst(x))$$

Notice that $pstate(x) = pstate(y)$ is equivalent to $state(x) = state(y) \wedge dst(x) = dst(y)$. Therefore, *pstate* can determine whether two paths should be compared in the algorithm. We will provide a more discussion about what *pstate* stand for in the next subsection.

Our optimal path querying algorithm is the following, where h is the objective function.

Procedure 5.7 (Optimal path querying algorithm)**.**
Input: a graph.
Output: an optimal path if it exists.
(1) Let W be $\{[]\}$ and N be $\emptyset$.
(2) Exit if $W = \emptyset$. // *There exists no feasible path.*
(3) Extract the minimum-h-valued path x from W.
(4) If x is feasible, return x. // *x is an optimal path.*
(5) Add $pstate(x)$ to N.
(6) For each path $z \in \{x \mathbin{+\!\!+} [a] \mid a \in E\}$,
(6-a) If $pstate(z) \in N$, do nothing.
(6-b) If $\forall y \in W : pstate(z) \neq pstate(y)$, add z to W.
(6-c) If $\exists y \in W : pstate(z) = pstate(y) \wedge h(z) < h(y)$, replace y by z.
(7) Go to (2). □

Theorem 5.8. Procedure 5.7 is correct; in other words, Procedure 5.7 always terminates and returns an optimal path if it exists.

Proof. Notice that W never contains two paths whose *pstate* values are the same. Moreover, W never contains a path whose *pstate* value is in N. Therefore, the size of N increases strictly. Since N is always a subset of the range of *pstate*, which is finite, the algorithm terminates.

From Lemmas 5.4, 5.5, and 5.6, it is sufficient to consider extensions of the minimum-h-valued path for each equivalent class raised from the value of *pstate*. Actually, W surely retains the minimum-h-valued path found for each equivalent class. Moreover, from Requirement 5.1, paths are examined in increasing order of their h-values in the step (4); thus, paths discarded in the step (6-a) are unnecessary for finding the optimal path, because another path of the same *pstate*-value and less (or equal) h-value was considered before. Besides, the feasible path found firstly is the optimal path because of the same reason. In summary, the algorithm is correct. □

The problem is reduced into a minimum-weighted path problem on a larger graph, whose vertexes corresponds to the range of *pstate*. The path x on the original graph is recognized as a path from the vertex $pstate([])$ to the vertex $pstate(x)$ on the larger graph, and Procedure 5.7 finds the minimum-weighed path from $pstate([])$ to the vertex v such that $pstate(x) = v$ implies feasibility of x. Moreover, Procedure 5.7 scans the larger graph in a "on-the-fly" manner, that is, it does not explicitly construct the larger graph. The procedure uses the function *pstate* as a compressed representation of the larger graph, identify a vertex only if the procedure runs across it, and terminates when an optimal path is found. Therefore, most of the inefficiency raised from generating the larger graph is eliminated.

Data Structures for Efficient Implementation

While an ideal hash set provides an efficient implementation of N, the priority queue W requires a bit complicated data structure. The data structure should support efficient implementation of inserting an element, extracting the minimum-weighted element, decreasing the weight of an element, and finding an element of the specified *pstate*-value. We prepare a Fibonacci heap with an ideal hash map for W. The Fibonacci heap stores paths according to their weights. The hash map associates each *pstate*-value to the element having the value in the Fibonacci heap. Notice that we should rearrange pointers in the hash set after the Fibonacci heap is manipulated. Therefore, we prepare back pointers from elements in the Fibonacci heap to the entries of the hash map, and keep their consistency.

After all, operations for W except extract-minimum can be done in (amortized) constant time. Extracting the minimum-weighted element from W takes $\mathrm{O}(\log n)$ time, where n is the size of W.

Computational Complexity

Let k be the size of the range of *state*; then the size of the range of *pstate* is at most $k|V|$. We assume that all results of recursive functions are memoized.

In Procedure 5.7, the steps (2) to (7) are executed at most $k|V|$ times. Each execution of the steps (2) to (5) costs $O(\log(k|V|))$ time. Each execution of the step (6) costs amortized $O(1)$ time for a path. The number of such paths is at most $k|E|$, because each edge is used at most k times. In summary, the time complexity of Procedure 5.7 is $O(k|V|\log(k|V|) + k|E|)$.

It is worth noting that the value k depends only on the description of the query; hence, the time complexity of Procedure 5.7 is polynomial in the size of the graph. However, the value k would be exponential to the size of the description. For example, a traveling salesman problem requires a description of at least $O(|V|)$ size, and then the value k becomes $O(2^{|V|})$; thus, it is an exponential time algorithm, yet it is much more efficient than the trivial algorithm that takes $O(|V|!)$ time.

5.3.3 Relationship to Product Construction

Well, let us change our viewpoint. From other point of view, our construction of the larger graph can be rephrased as product construction of finite automata. This view is useful for discussing further improvement of our algorithm.

First of all, the function *state* can be recognized as a representative function of a DFA that takes a sequence of edges and checks the feasibility of the sequence. Let S be the range of *state*, and define the transition function δ and the set of final states S_F by $\delta(\mathit{state}(x), a) = \mathit{state}(x +\!\!+ [a])$ and $S_F = \{\mathit{state}(x) \mid x \in E^* \wedge p(x)\}$, where p is the requirement for feasibility. Then, the DFA $\mathcal{S} = (S, E, \delta, \{\mathit{state}([])\}, S_F)$ accepts all sequences of edges each of which is feasible if it is a path.

Figure 5.3 shows the DFA corresponds to the function *state* for the shortest path problem with transit cost, where, for simplicity, we regard the recursive function *walk* as *state*. In the DFA, the initial state is s_0, which corresponds to the case where *walk*-value is *True*, that is, we are not on a train. When we ride on a train, the state becomes s_1, which corresponds to the case where *walk*-value is *False*. It turns into s_0 when we get off a train. Both s_0 and s_1 are final states, because *walk*-values do not affect to feasibility.

Next, let us interpret a graph as an NFA that accept a sequence of edges if and only if it is a path on the graph. Let the transition relation τ be $\tau = \{(v', (v, e)) \mid v, v' \in V \wedge e \in E \wedge \mathit{hd}(e) = v \wedge \mathit{tl}(e) = v'\}$. Then, $\mathcal{G} = (V, E, \tau, V, V)$ forms an NFA that accepts all paths on the graph.

Finally, consider the product of $\mathcal{S}$ and $\mathcal{G}$, say $\mathcal{G}' = (S \times V, E, \tau', \{(\mathit{state}([]), v) \mid v \in V\}, S_F \times V)$. Then, the automaton $\mathcal{G}'$ accepts a sequence of edges if and only if it is a feasible path. Moreover, each transition has a unique cost because the branch used to compute the cost can be determined by the preceding state. Therefore, the optimal path querying problem is reduced into the minimum-weighed word problem

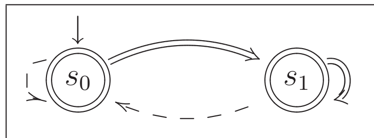

Figure 5.3. A DFA representing the structure of case analyses of the shortest path problem with transit costs. The state s_1 corresponds to the situation of riding on a train, and s_0 corresponds to the others. Double-lines arrows are transitions when the DFA takes an edge of riding on a train, and broken arrows are transitions when it takes an edge of not riding on a train. s_0 is the initial state, and both s_0 and s_1 are final states.

on $\mathcal{G}'$. It is worth noting that the function *pstate* used in Procedure 5.7 is the representative function of $\mathcal{G}'$. Therefore, Procedure 5.7 exactly finds the minimum-weighed word on $\mathcal{G}'$.

As an example, consider the product of the DFA in Figure 5.3 and the DFA corresponding to the underlying graph (e.g., the left of Figure 5.2). The product corresponding to the graph that outlines a reduction into a shortest path problem (e.g., the right of Figure 5.2). The weight of each edge is given by the objective function *cost*.

The key to the correspondence between our derivation and product construction is the finiteness of the range of *state*. Since the range of *state* is finite, *state* can be rephrased as a DFA and our derivation is equivalent to product construction. An important fact is that since we have rephrased our derivation in terms of finite automata, we can utilize known rich results about finite automata to analyze and improve our optimal path querying algorithm.

5.3.4 Improving Efficiency of the Optimal Path Querying Algorithm

We have shown an algorithm to find the optimal path. While the algorithm is correct, there is room for improvement. For example, when we want to find the shortest path from a vertex s, paths that do not start from s are useless. However, naive execution of Procedure 5.7 results in enumeration of such useless paths. Here, we propose improvements to remove such inefficiency.

Our improvements is based on analyses of the DFA corresponding to the function *state*. It is worth noting that direct minimization of the DFA is incorrect. For two paths x and y, assume that $x +\!\!+ z$ is feasible if and only if $y +\!\!+ z$ is feasible. Then, we may attempt to merge $\mathit{state}(x)$ with $\mathit{state}(y)$. However, $h(x) \leq h(y)$ may not imply $h(x +\!\!+ z) \leq h(y +\!\!+ z)$, where h is the objective function, because different conditional branches may be used to compute $h(x +\!\!+ z)$ and $h(y +\!\!+ z)$; thus, merging $\mathit{state}(x)$ with $\mathit{state}(y)$ will break the property in Lemma 5.6. In other words, we should be careful about the branches in the objective function.

To simplify the discussion, we assume the objective function h is described in the following form, in which each of e'_i and ϕ_i contains no conditional expressions.

$$\begin{array}{lcl} h([]) & = & n; \\ h(x \mathbin{+\!\!+} [a]) & = & \textsf{if } \phi_1 \textsf{ then } e'_1 \\ & & \textsf{else if } \phi_2 \textsf{ then } e'_2 \\ & & \qquad \vdots \\ & & \textsf{else if } \phi_{m-1} \textsf{ then } e'_{m-1} \\ & & \textsf{else } e_m \end{array}$$

It is easy to rewrite the description of the objective function into this form. For notational convenience, we define $\phi'_k \stackrel{\text{def}}{=} \phi_k \wedge \bigwedge_{i=1}^{k-1} \neg\phi_i$ for $1 \leq k \leq m-1$ and $\phi'_m \stackrel{\text{def}}{=} \bigwedge_{i=1}^{m-1} \neg\phi_i$. The predicate ϕ'_k stands for the condition when the k-th branch is chosen.

Let $\mathcal{S} = (S, E, \delta, \{\mathit{state}([])\}, S_F)$ be the DFA defined in the previous subsection. Label each transition by an integer $i \in \{1, \ldots, m\}$, which stands for the branch used to compute the objective function. Then, $\mathcal{S}' = (S, E \times \{1, \ldots, m\}, \delta', \{\mathit{state}([])\}, S_F)$ also forms a DFA, where the transition function δ' is defined as follows.

$$\delta'(\mathit{state}(x), (a, i)) \stackrel{\text{def}}{=} \mathit{state}(x \mathbin{+\!\!+} [a]) \ \text{ if } \phi'_i(x \mathbin{+\!\!+} [a])$$

Now let us introduce our improvements.

Lemma 5.9. If $L_{\mathcal{S}'[\{\mathit{state}(x)\}]} = \emptyset$ holds for a sequence x, $x \mathbin{+\!\!+} y$ is not feasible for any sequence y.

Proof. It is evident because $y \in L_{\mathcal{S}'[\{\mathit{state}(x)\}]}$ is equivalent to that $x \mathbin{+\!\!+} y$ is feasible if we ignore the labels. □

Lemma 5.10. For the objective function h and two sequences x and y, assume all of $L_{\mathcal{S}'[\{\mathit{state}(x)\}]} = L_{\mathcal{S}'[\{\mathit{state}(y)\}]}$, $\mathit{dst}(x) = \mathit{dst}(y)$, and $h(x) \leq h(y)$ hold; then, for any sequences z, $x \mathbin{+\!\!+} z$ is feasible if $y \mathbin{+\!\!+} z$ is feasible, and $h(x \mathbin{+\!\!+} z) \leq h(y \mathbin{+\!\!+} z)$ holds.

Proof. We will introduce a stronger lemma (Lemma 5.11) next. □

Lemmas 5.9 and 5.10 enable us to find unnecessary paths. These lemmas respectively show how to find paths whose extensions yield no feasible path and those whose extensions yield no better path than others. Lemma 5.9 states that if $L_{\mathcal{S}'[\{\mathit{state}(x)\}]} = \emptyset$ holds for a path x, then we can immediately discard the path. Lemma 5.10 states that we can compare paths x and y and discard the worse one if $L_{\mathcal{S}'[\{\mathit{state}(x)\}]} = L_{\mathcal{S}'[\{\mathit{state}(y)\}]}$ holds. We can compute such information about states beforehand by implementing checks on emptiness and inclusion of regular languages; then, we can reduce the number of paths considered for finding the optimal path.

From the viewpoint of algorithm development, we could introduce a stronger lemma than Lemma 5.10.

Lemma 5.11. For the objective function h and two sequences x and y, assume all of $L_{\mathcal{S}'[state(x)]} \supseteq L_{\mathcal{S}'[state(y)]}$, $dst(x) = dst(y)$, and $h(x) \leq h(y)$ hold; then, for any sequences z, $x \mathbin{+\!\!+} z$ is feasible if $y \mathbin{+\!\!+} z$ is feasible, and $h(x \mathbin{+\!\!+} z) \leq h(y \mathbin{+\!\!+} z)$ holds.

Proof. It is evident that $L_{\mathcal{S}'[\{state(x)\}]} \supseteq L_{\mathcal{S}'[\{state(y)\}]}$ means that feasibility of $y \mathbin{+\!\!+} z$ implies feasibility of $x \mathbin{+\!\!+} z$, as similar to the case of Lemma 5.9.

Consider $z' \in L_{\mathcal{S}'[\{state(x)\}]}$, and let z be an sequence obtained by removing labels from z'. Since $z' \in L_{\mathcal{S}'[\{state(x)\}]}$ holds, the branches used for computing $h(x \mathbin{+\!\!+} z)$ from $h(x)$ is the same as that for $h(y \mathbin{+\!\!+} z)$ from $h(y)$. Therefore, $h(x \mathbin{+\!\!+} z) \leq h(y \mathbin{+\!\!+} z)$ holds from the construction of h, as similar to the proof of Lemma 5.6. □

However, naive implementation of Lemma 5.11 is inefficient, because it is not easy to find better (or worse) paths from the priority queue W in Procedure 5.7; hence, Lemma 5.10 would not be appropriate for automatic implementation. Lemma 5.11 is useful for more restrictive settings where it is easy to point out better/worse paths than the given path.

5.3.5 Correspondence to Existing Algorithms

Dijkstra-like algorithms have been proposed for several classes of optimal path problems, and some of them are equivalent to Procedure 5.7 with the improvements, when the specification of the problem can be written in our language. In other words, our algorithm is a generalization of existing algorithms.

Fact 5.12. Procedure 5.7 with the improvements by Lemmas 5.9 and 5.11 is equivalent to the following algorithms except for implementation of the priority queue: Dijkstra algorithm for point-to-point shortest path problems; the generalized permanent labeling algorithm by Desrochers and Soumis [DS88] for shortest path problems with time windows; the heap-Dijkstra algorithm by Sherali et al. [SJH06] for approach-dependent, time-dependent, label-constrained shortest path problems. □

In the algorithms above, like ours, problems are reduced into shortest path problems and solved by implicit application of Dijkstra algorithm. Our construction of *pstate* certainly corresponds to their reductions, and overheads are removed by our improvements.

5.3.6 How the Querying Algorithm Works for Examples

Shortest Path Problem with Transit Costs

From the description of the shortest path problem with transit costs, the function *state* is derived as follows.

$$state(x) \stackrel{\text{def}}{=} (\widehat{cost}(x), constraint(x), start_s(x), empty(x), walk(x))$$
$$\widehat{cost}(x) \stackrel{\text{def}}{=} \infty$$

The values of $\widehat{cost}$, p, *empty* are not essential: $\widehat{cost}$ always returns ∞ because of absence of inequalities; *constraint* always yields *False* until an optimal path is found; *empty* yields *False* for any non-empty paths. In summary, the auxiliary function *state* distinguishes paths by values of $start_s$ and *walk*.

Lemma 5.9 tells us that a path x is unnecessary if neither $empty(x)$ nor $start_s(x)$ holds, which corresponds to the case where x does not start from the vertex s. Thus, after the improvement, the algorithm enumerates paths starting from s, and paths are distinguished based on the current vertex and whether we are riding on a train. The number of paths considered in the step (6) of Procedure 5.7 is at most $4|V|+2$, which becomes $2|V|+2$ after the improvement.

Length-Constrained Shortest Path Problem

The *state* for a length-constrained shortest path problem is obtained as follows.

$$\begin{aligned}
state(x) &\stackrel{\text{def}}{=} (\widehat{wsum}(x), \widehat{len}(x), constraint(x), start_s(x), empty(x)) \\
\widehat{wsum}(x) &\stackrel{\text{def}}{=} \infty \\
\widehat{len}(x) &\stackrel{\text{def}}{=} \begin{cases} len(x) & \text{if } len(x) \leq K \\ \infty & \text{otherwise} \end{cases}
\end{aligned}$$

As the same as the previous example, values of $\widehat{wsum}$, *constraint*, and *empty* are not important. Paths are distinguished by the values of $start_s$ and $\widehat{len}$: whether the path starts from s and how much edges are used (or, whether more than K edges are used).

Lemma 5.9 tells us that a path x is unnecessary if neither $empty(x)$ nor $start_s(x)$ holds or $\widehat{len}(x)$ is ∞. The former is the case where x does not start from s, and the latter is the case that x consists of more than K edges. Lemma 5.11 tells us that a path x is unnecessary if there exists a path y such that all of the following four hold: the destinations of x and y are the same; all values of *constraint*, $start_s$, *empty* are the same for x and y; $\widehat{len}(x) \geq \widehat{len}(y)$; $wsum(x) \geq wsum(y)$. This is the case that the *wsum*-value of x is worse than that of y while x uses more edges than y.

The number of paths considered in the step (6) of Procedure 5.7 is at most $2(K+2)|V|+2$, which becomes $(K+1)|V|+2$ after the improvements.

5.4 Optimal Path Querying System

In this section, we report our implementation of optimal path querying system and experimental results. The system is available from the author's website[1].

[1] `http://www.ipl.t.u-tokyo.ac.jp/~morihata/OPQ.tar.gz`

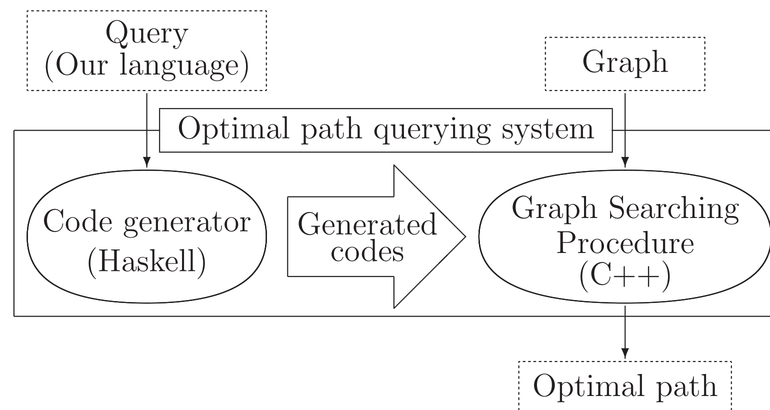

Figure 5.4. Overview of the optimal path querying system

5.4.1 Implementation of Optimal Path Querying System

Figure 5.4 shows the overview of the system. Querying on the system consists of two stages. The first stage is code generation, where the system analyses the query description and generates codes that consist of information for efficient querying. The second is graph searching, where the system performs Procedure 5.7 using the generated codes.

This two-staged implementation enhances modularity. These two stages are essentially independent, because the first stage only cares about descriptions of queries while the second one concentrates on searching on graphs. Moreover, since Procedure 5.7 is essentially Dijkstra algorithm, though a bit generalized, it would be possible to substitute another library for the second stage. Besides, separating these two would be practical to use our system as a component of a system.

Code Generator

The code generator is made of three hundred lines of Haskell codes excluding codes for its parser. It generates C++ codes, which mainly includes the following four: the definitions of recursive functions, the definition of the auxiliary function *pstate*, the requirement for feasible paths, and the improvements.

We only implemented the improvement of Lemma 5.9. To make the computation of this part faster, we binarize each integer-valued function by whether the value exceeds the boundary.

Graph Searching Procedure

Procedure 5.7 is implemented in C++. As mentioned, this part could be replaced by another library. The program is made of three hundred lines of codes.

In the implementation, we use a heap instead of a Fibonacci heap for implementing the priority queue, because Fibonacci heaps are inefficient in practice. Hence,

```
minimize wsum(x)
s.t. len(x) <= 20 && start0(x) && to1(x)
  where
  wsum([])     = 0;
  wsum(x++[e]) = w(e) + wsum(x);
  len([])     = 0;
  len(x++[e]) = 1 + len(x);
  empty([])     = true;
  empty(x++[e]) = false;
  start0([])     = false;
  start0(x++[e]) = st0(x) || (empty(x) && from(e,0));
  to1([])     = false;
  to1(x++[e]) = to(e,1);
```

Figure 5.5. A query description of a length-constrained shortest path problem.

the time complexity of our implementation is $O(k|E|\log(k|V|))$, where k is the size of the range of *state*.

5.4.2 Sample Codes

Figures 5.5 and 5.6 respectively show the query description and the generated C++ codes for a length-constrained shortest path problem. In the query description in Figure 5.5, both `from(e,0)` and `to(e,1)` are atomic predicates, where `0` and `1` are identifiers of vertexes. Therefore, it is a query to find the minimum-weighted path from the vertex `0` to the vertex `1` among those consist of less than or equal to 20 edges. In the codes in Figure 5.6, a constructor `val(val v,edge e)` performs the computations and memoizations of recursive functions. The definition of *pstate* is encoded as a function object `val_eq_t` and a hash function `val_hash_t`. The former is used in the heap, and the latter is used in hash sets. The function `constraint` is the function to check feasibility. The function `unnecessary` is the function to find unnecessary paths, which is obtained from the improvement of Lemma 5.9.

5.4.3 Experiments

To evaluate effectiveness of our implementation, we did some experiments. The environment of our experiments is the following: dual Intel Quad-Core Xeon 3.0 GHz CPUs; 8 GB memory; Mac OS X; g++ 4.2.2 and ghc 6.6.1. Only one core was used while the machine had eight cores.

We tested the following four queries: SP (find the point-to-point shortest path), 3-SP (find the point-to-point shortest path that passes a specified vertex), TRANS (find the shortest path with transit costs), and TLEN (find the shortest path that uses less or equal to twenty edges of riding on a train). For each specification, the code generation step finished immediately (less than 0.1 second). In addition to them, we prepare an implementation of the point-to-point shortest path querying

```
struct val {
  int wsum;
  int len;
  bool empty;
  bool st0;
  bool to1;
  node n;
  val() {
    n = -1;
    wsum = 0;
    len = 0;
    empty = true;
    st0 = false;
    to1 = false;
  }
  val(val v,edge e) {
    wsum = (e.w+v.wsum);
    len = (1+v.len);
    empty = false;
    st0 = (v.st0 || (v.empty && e.in==0));
    to1 = e.out==1;
    n = e.out;
  }
  int weight() const { return wsum; }
  static val ninf() {
    val v;
    v.wsum = INT_MIN;
    return v;
  }
};
struct val_hash_t {
  unsigned long operator()(const val &v) const {
    unsigned long long k = v.n;
    k = (k * PRIME_FOR_HASH + v.empty) % PRIME_FOR_HASH_;
    k = (k * PRIME_FOR_HASH + v.st0) % PRIME_FOR_HASH_;
    k = (k * PRIME_FOR_HASH + v.to1) % PRIME_FOR_HASH_;
    k = (k * PRIME_FOR_HASH + ((v.len<=20)?v.len:21)) % PRIME_FOR_HASH_;
    return (unsigned long)k;
  }
} val_hash;
struct val_eq_t {
  bool operator()(const val &v, const val &w) const {
    return (v.n == w.n &&
            v.empty == w.empty &&
            v.st0 == w.st0 &&
            v.to1 == w.to1 &&
            ((v.len<=20)?v.len:21) == ((w.len<=20)?w.len:21));
  }
} val_eq;
inline bool constraint( val v ) {
  return ((v.len<=20) && (v.st0 && v.to1));
}
inline bool unnecessary( val a ) {
  return ((a.empty && a.st0 && a.to1 && (a.len>20)) ||
          (a.empty && a.st0 && (!a.to1) && (a.len>20)) ||
          (a.empty && (!a.st0) && a.to1 && (a.len>20)) ||
          (a.empty && (!a.st0) && (!a.to1) && (a.len>20)) ||
          ((!a.empty) && a.st0 && a.to1 && (a.len>20)) ||
          ((!a.empty) && a.st0 && (!a.to1) && (a.len>20)) ||
          ((!a.empty) && (!a.st0) && a.to1 && (a.len<=20)) ||
          ((!a.empty) && (!a.st0) && a.to1 && (a.len>20)) ||
          ((!a.empty) && (!a.st0) && (!a.to1) && (a.len<=20)) ||
          ((!a.empty) && (!a.st0) && (!a.to1) && (a.len>20)));
}
```

Figure 5.6. The C++ codes generated from the query in Figure 5.5.

Table 5.1. Size of graphs

	RND_1	RND_2	RND_3	RND_4	NY	FLA	CAL	EUSA
V	131,072	131,072	1,048,576	1,048,576	264,346	1,070,376	1,890,815	3,598,623
E	524,288	2,097,152	2,097,152	4,194,304	733,846	2,712,798	4,657,742	8,778,114

Table 5.2. Experimental results (unit: second): the bracketed numbers show the number of essential states of the DFA corresponding to *state*.

Query	RND_1	RND_2	RND_3	RND_4	NY	FLA	CAL	EUSA
SP-boost	0.11	0.24	1.04	1.55	0.10	0.48	1.06	2.60
SP (1)	0.31	0.77	1.68	3.14	0.29	1.29	2.67	6.52
3-SP (2)	0.84	2.24	3.95	9.11	0.88	4.12	8.90	21.82
TRANS (2)	0.38	1.02	2.02	3.94	0.40	1.79	3.80	9.60
TLEN (42)	1.52	3.36	28.42	20.08	8.10	10.06	14.99	25.02

for comparison. The implementation is based on Dijkstra algorithm in C++ Boost Graph Library [SLL01], and denoted by "SP-boost".

We used eight graphs. We generated four graphs, where the startpoint, endpoint, and weight of each edge were given randomly. These four are denoted by RND_1 (relatively small), RND_2 (dense), RND_3 (sparse), and RND_4 (relatively large). We borrowed four graphs from the benchmarks of the 9th DIMACS implementation challenge[2]. They were travel time data of NY (New York City), FLA (Florida), CAL (California and Nevada), and EUSA (Eastern USA). The sizes of graphs are shown in Table 5.1. In these graphs, edges had no category information (such as "train"), and we added the information irresponsibly. We regarded each vertex whose identifier is odd as a station and each edge from a station to a station as an edge of riding on a train. For each graph, we uniformly generated 1000 pairs of a starting point and a destination (and another vertex for 3-SP), and measured the average computational times.

The results are shown in Table 5.2. The bracketed numbers in the first column are the numbers of states of the DFA corresponding to *state* after applying the improvement of Lemma 5.9, and show the theoretical ratios of computation times. To see precise ratios, we count the number of "essential" states, that is, we neglect states that represent the empty path or the optimal path. The other columns show computational times excluding times for inputting the graph and outputting the results.

On one hand, even for the road network of eastern USA, our system returned results of queries in a minute, which is only several times slower than the point-to-point shortest path querying by an existing library. This fact would demonstrate promise of our framework. On the other hand, SP runs two or three times slower than

[2] 9th DIMACS Implementation Challenge - Shortest Paths. 2006. http://www.dis.uniroma1.it/~challenge9/.

SP-boost. The difference is the overhead of generality. Especially, Procedure 5.7 requires a data structure that is a bit more complicated than an ordinary heap, as discussed in Section 5.3.2, and it affects efficiency. It is worth noting that the ratio does not go worse even when the graph gets larger.

The results indicate that more detailed experiments would be necessary for application-specific uses of our framework. First, observe that practical computational times are not exactly propositional in the theoretical complexities shown by the numbers of states. TRANS is much faster than 3-SP, and ratios of computational times between TLEN and others are relatively small in comparison with ratios of numbers of states. Moreover, computational times depend on combinations of queries and graphs. For example, the theoretical inefficiency of TLEN comes to the surface when there are few shortcutting routes because of the necessity to consider paths of many edges.

Someone may notice that a 3-SP problem can be solved by a composition of two SP queries. When we want to find a shortest route from a vertex v_1 to a vertex v_2 via a vertex v_3, it is sufficient to find the shortest paths from v_1 to v_3 and from v_3 to v_2. However, 3-SP is about three times slower than SP on our system. This indicates that there are rooms for further improvement.

5.5 Summary and Discussions

In this chapter, we have developed a framework for optimal path querying. Inspired from the derivation of algorithms for shortest path problems and their variants, we designed a DSL for optimal path querying and proposed an optimal path querying algorithm. The key to an efficient algorithm is incrementality condition for utilizing our calculational laws and finiteness of the searching space for guaranteeing termination. From the viewpoint of finite state automata, the derivation of our querying algorithm can be understood as the product construction. This viewpoint helps us to introduce improvements of our querying algorithm. By putting together, we derived an efficient querying algorithm that is a generalization of existing querying algorithms. We also implemented our idea as an optimal path querying system. Our implementation demonstrates promise of our framework in the sense that optimal path querying is only several times slower than the shortest path querying.

Since optimal path querying is an important topic, there are many studies about solving variants of shortest path problems, such as problems specified by additional constraints and variation of costs [Jok66, DS88, Rom88, Pun91, MPRS99, BJM00, BJ04, VD05, SJH06]. Optimal path querying systems were also proposed. For example, regular-language-constrained shortest path queries on a time-dependent network are available on the route planner of TRANSIMS system [BBJ+02, BBJ+07]. We aimed to construct a general framework that includes many of them. However, there are still several problems that our framework cannot deal with even though efficient algorithms are known, for example maximum capacity path problems [Pun91] and

context-free-language-constrained shortest path problems [BJM00]. Optimal path querying on spatial databases is also a topic we have not taken into consideration. Chan and Zhang [CZ07] proposed a generic optimal path querying algorithm on spatial databases. Roughly speaking, the algorithm can deal with problems that satisfy Bellman's principle of optimality [Bel57], namely each subpath of the optimal path is the optimal subpath. Our framework does not require such property, while it seems difficult to implement our framework on spatial databases.

Ogawa et al. [OHS03] also proposed a framework to query and analyze graphs efficiently. Their framework is, similar to ours, based on the work about maximum marking problems by Sasano et al. [SHTO00]. While their framework can solve general problems rather than path queries, it requires the underlying graph to be tree decomposable [ALS91, Bod96, FFG02] and users to write recursive functions on the structure of tree decompositions. Such requirements make the use of the framework hard. Ikarashi et al. [ITNH08] proposed another framework for dealing with general graph problems rather than path problems. They defined the modal μ-calculus on natural numbers and discussed graph problems as model checking problems on the calculus. However, it is not clear how practically useful their framework is; besides, for path problems, our algorithm is more efficient than theirs.

Another way to generically solve constrained shortest path problems is the use of ranking shortest path algorithms [Mar84, Epp98]. We can find an optimal path by enumerating paths from shorter ones until a feasible path is found. Although this procedure works for any constrained shortest path problems, neither computational complexity nor termination of the procedure is hard to guarantee.

We reduced optimal path querying problems into minimum-weighed word problems on finite state automata. Actually, our derivation follows classical results about correspondence among finite state automata, dynamic programming, and shortest path problems. Karp and Held [KH67] showed that finite state automata provided a good characterization of dynamic programming algorithms. Ibaraki [Iba73, Iba74, Iba78] extended their results, and showed that once a problem is specified in a certain specific form by using finite state automata, we can reduce the problem into a minimum-weighed word problem and solve it by an algorithm corresponding to its form, which is exactly an algorithm for shortest path problems. Therefore, the issue is the way to specify the problem by a finite state automaton.

We used product construction for reducing optimal path querying problems into minimum-weighed word problems. Actually, the use of product construction together with on-the-fly construction for optimal path querying is not our new invention. Romeuf [Rom88] showed that regular-language-constrained shortest path problems can be reduced into shortest path problems by product construction. Barrett et al. [BBJ+02] and Sherali et al. [SJH06] used on-the-fly construction of larger graphs in their optimal path querying algorithms. Similar methods were also adopted by de Moor et al. [dMLW03] and Liu et al. [LRY+04] to obtain algorithms for regular path queries.

In summary, our result is not quite new from the algorithmic viewpoint. Our

main contribution is to generalize these ideas to cope with a wider class of problems, such as problems concerning integer-valued constraints, and to design a DSL-based interface to the algorithmic results, and to provide a unified framework for optimal path querying. The unified view makes it easy for nonspecialists to enjoy these results.

Since we reduced optimal path problems into shortest path problems, known results about shortest path problems would be useful for our framework. For instance, use of A* search algorithms or the bidirectional Dijkstra search algorithm, instead of the use of Dijkstra algorithm, would improve efficiency. Ranking shortest path algorithms [Mar84, Epp98] might be also useful to obtain nearly-optimal results.

As an extension of our framework, Kita [Kit08] considered combination of optimal path problems and maximum marking problems. She discussed algorithms for optimal path querying in which the objective-function value of a path is the minimum/maximum marking of the edges on the path, and the marking should satisfy a given requirement. It is relatively easy to find the path whose minimum feasible marking is minimum, if feasibility of marking can be checked by a finite state automaton, because such problems are easily reduced into minimum-word problems as similar to problems discussed in this chapter. However, it is difficult to find the path whose maximum feasible marking is minimum, because we hardly reduce such problems into minimum-word problems. It is future work to provide clear and efficient algorithms together with useful domain-specific languages for such generalizations.

Chapter 6

Calculational Laws for Parallel Programming

The theme of this chapter is systematic development of divide-and-conquer algorithms. In a divide-and-conquer algorithm, we first divide the problem into some independent subproblems, solve each subproblem independently, and finally merge the results of independent subproblems to obtain the result of the whole problem. The divide-and-conquer method is important not only for developing efficient sequential algorithms but also for developing efficient parallel algorithms. Divide-and-conquer algorithms are suitable for parallel computations, because independent subproblems that can be computed in parallel. Divide-and-conquer parallel programs have their merit of being suitable for cache-efficient implementation or implementation on distributed memory environments. Furthermore, divide-and-conquer parallel programs are efficient in the sense that they will show a good scalability with respect to number of processors. Since recursive divisions yield a lot of independent subproblems in divide-and-conquer algorithms, each processor is likely to sufficiently participate in the computation.

Scalability with respect to number of processors is one of the most important properties in parallel programming. If a parallel program has good scalability, it shows magical speedup that other optimization techniques can hardly gain. If scalability is poor, parallelization is nothing but an anti-optimization, because parallelization usually requires overheads such as communication costs and synchronization costs.

In this chapter, we would like to provide a methodology to develop divide-and-conquer algorithms that will raise efficient parallel programs on exclusive-read exclusive-write parallel random access machines (in short, EREW PRAM). Here, "efficient" means "*cost optimal*", that is, the program shows linear speedup with respect to number of processors up to a large number of processors. The main issues are the following three.

The main results of this chapter was published as [MM08] and [MMHT09].

- How can we develop efficient and generic parallel algorithms?
- How can we characterize efficient divide-and-conquer parallel programs?
- How can we provide calculational laws that are useful to develop divide-and-conquer parallel programs?

In Section 6.1, we review known results on lists. One of the most interesting results is *the third list-homomorphism theorem*, which is a folk theorem in calculational programming community. The theorem states that if a function can be written in two certain forms of sequential recursive programs, then there exists a cost-optimal divide-and-conquer parallel program to compute the function. We confirm that the third list-homomorphism theorem is certainly effective for developing parallel programs.

In Section 6.2, we consider node-valued binary trees. On binary trees, it is known that *parallel tree contraction* [MR85] provides a framework for developing cost-optimal parallel algorithms. After reviewing a parallel tree contraction algorithm, we provide a characterization of functions that can be parallelized based on parallel tree contraction. Then, we introduce the third list-homomorphism on binary trees, which proves that if a function traversing a tree can be written in two certain forms of sequential programs, then there exists a cost-optimal parallel program to compute the function. The main idea is to write tree-iterating functions as list-iterating functions so that we can utilize theories on lists. We focus on paths from the root of the tree to leaves, express them by *Huet's zippers*, which are lists containing one-hole contexts of trees, and introduce the notion of path-based computations.

In Section 6.3, we generalize the result in Section 6.2 to cope with all polynomial data structures rather than lists or binary trees. We generalize all of the parallel tree contraction algorithm, path-based computations, and the third list-homomorphism theorem up to polynomial data structures. These results are exactly a generalization of the known theories on lists, which indicates appropriateness of our approach.

6.1 Parallel Programming on Lists

6.1.1 List Homomorphisms

First, let us consider divide-and-conquer parallel programs on lists. As an example, consider summing up the elements in a list $[a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8]$. It is easy to develop sequential algorithms. Both of the rightward summation

$$((((((a_1 + a_2) + a_3) + a_4) + a_5) + a_6) + a_7) + a_8$$

and the leftward summation

$$a_1 + (a_2 + (a_3 + (a_4 + (a_5 + (a_6 + (a_7 + a_8))))))$$

are sequential algorithms. In this case, it is not difficult to think of the divide-and-conquer summation

$$((a_1 + a_2) + (a_3 + a_4)) + ((a_5 + a_6) + (a_7 + a_8))$$

where we divide the list at its center and compute each part in parallel. The divide-and-conquer summation is a parallel algorithm, and it computes the sum of a list of length n in $\lceil \log n \rceil$ steps if a sufficient number of processors are available.

The key to such a divide-and-conquer parallel algorithm is associativity. Compare the two sequential algorithms with the divide-and-conquer algorithm. The only differences are the structure of parentheses, and the associativity of $+$, namely $a + (b + c) = (a + b) + c$, enables us to rearrange the parentheses.

This observation, an associative operator raises a divide-and-conquer parallel algorithm, is formalized as the notion of *list homomorphisms* [Bir87]. List homomorphisms are an expressive computation pattern for divide-and-conquer parallel programs on lists.

Definition 6.1 (list homomorphism [Bir87])**.** A function $h : A^* \to B$ is said to be a *list homomorphism* if there exist a function $\phi : A \to B$ and an associative operator $(\odot) : (B \times B) \to B$ such that

$$\begin{array}{lcl} h([]) & = & \iota_{\odot} \\ h([a]) & = & \phi(a) \\ h(x +\!\!+ y) & = & h(x) \odot h(y) \end{array}$$

hold, where $\iota_{\odot}$ is the unit of $\odot$. In this case, we write $h = \mathsf{hom}_{\odot,\phi}$. □

An associative operator $\odot$ characterizes a list homomorphism. The associativity of $\odot$ guarantees that the result of computation is not affected by the place to divide the list. List homomorphisms are useful to develop parallel programs on lists, and actually many studies were done on list homomorphisms [Bir87, Col94, Col95, Gib96, Gor96, HIT97].

It is easy to see that $\mathsf{hom}_{\odot,\phi}$ for a list of length n can be evaluated on p processors in $O(n/p + \log p)$ time, if both ϕ and $\odot$ is a constant-time computation. Thus, list homomorphisms are cost optimal up to $O(n/\log n)$ processors. In other words, list homomorphisms show good scalability with respect to number of processors.

Recall the summation as an example. The summation function *sum* satisfies the following equations.

$$\begin{array}{lcl} sum([]) & = & 0 \\ sum([a]) & = & a \\ sum(x +\!\!+ y) & = & sum(x) + sum(y) \end{array}$$

Hence, *sum* is a list homomorphism, namely $sum = \mathsf{hom}_{+,id}$.

6.1.2 The Third List-Homomorphism Theorem

The third list-homomorphism theorem [Gib96] is a folk theorem in calculational programming community. The theorem shows a necessary and sufficient condition to be a list homomorphism.

Theorem 6.2 (the third list-homomorphism theorem [Gib96])**.** A function h is a list homomorphism if and only if there exist two operators $\oplus$ and $\otimes$ such that the following equations hold.

$$\begin{array}{rcl} h([a] +\!\!+ x) & = & a \oplus h(x) \\ h(x +\!\!+ [a]) & = & h(x) \otimes a \end{array} \qquad \square$$

The third list-homomorphism theorem states that if we can compute a function in both leftward and rightward manners, there exists a divide-and-conquer parallel algorithm to evaluate the function. What the theorem proves is not only existence of parallel programs but a way to develop parallel programs systematically. The following lemma plays a central role in parallelization.

Lemma 6.3 ([Gib96, MMM+07])**.** Assume that the following equations hold for a function h.

$$\begin{array}{rcl} h([a] +\!\!+ x) & = & a \oplus h(x) \\ h(x +\!\!+ [a]) & = & h(x) \otimes a \end{array}$$

Then, $h = \mathsf{hom}_{\odot,\phi}$ holds, where $\odot$ and ϕ are defined as follows.

$$\begin{array}{l} \phi(a) = h([a]) \\ a \odot b = h(h^\circ(a) +\!\!+ h^\circ(b)) \end{array} \qquad \square$$

Lemma 6.3 states that we can derive a parallel program from two sequential programs through their converse. Since a converse of a function is a relation in general and difficult to reason, we will consider a right inverse rather than the converse.

6.1.3 Developing Parallel Programs for Lists with the Third List-Homomorphism Theorem

Summation

As a first example, let us review the summation function *sum*. As mentioned, *sum* is both leftward and rightward.

$$\begin{array}{rcl} sum([\,]) & = & 0 \\ sum([a] +\!\!+ x) & = & a + sum(x) \\ sum(x +\!\!+ [a]) & = & sum(x) + a \end{array}$$

Therefore, the third list-homomorphism theorem proves that *sum* is a list homomorphism. Next, let us derive a parallel program of *sum* based on Lemma 6.3. The

lemma indicates that a right inverse brings a parallel program. It is easy to develop a right inverse of *sum*. For example, the function *wrap* is a right inverse of *sum*, because $sum(wrap(sum(x))) = sum([sum(x)]) = sum(x)$ holds. Then, to obtain ϕ and $\odot$ such that $sum = \mathsf{hom}_{\odot,\phi}$ holds, we calculate as follows.

$$\begin{array}{lcl}
\phi(a) & = & \{ \text{ Lemma 6.3 } \} \\
 & & sum([a]) \\
 & = & \{ \text{ definition of } sum \text{ } \} \\
 & & a \\
a \odot b & = & \{ \text{ Lemma 6.3 } \} \\
 & & sum(sum^\circ(a) +\!\!+ sum^\circ(b)) \\
 & = & \{ \text{ refine } sum^\circ \text{ to } wrap \text{ } \} \\
 & & sum([a] +\!\!+ [b]) \\
 & = & \{ \text{ definition of } sum \text{ } \} \\
 & & a + b
\end{array}$$

Therefore, $\phi = id$ and $\odot = +$ hold. In summary, we have successfully derived a parallel program for *sum*, which is exactly the divide-and-conquer summation.

Maximum Initial-segment Sum

Next, let us consider the maximum initial-segment sum problem, in which we would like to compute the maximum of summations of initial segments of a list. For example, maximum initial-segment sum of a list $[1, -2, 1, 6, -5, 2, 1,]$ is $1 + (-2) + 1 + 6 = 6$, because $[1, -2, 1, 6]$ is the initial segment of the maximum sum.

Let us program a solution of maximum initial segment problem, say *mis*, in both leftward and rightward manner. It is relatively easy to develop a leftward program.

$$\begin{array}{lcl}
mis([]) & = & 0 \\
mis([a] +\!\!+ x) & = & 0 \uparrow (a + mis(x))
\end{array}$$

Since a non-empty initial segment of $[a] +\!\!+ x$ must contain a, maximum initial segment sum of $[a] +\!\!+ x$ is the maximum of 0 (empty list) and $a + mis(x)$ (maximum sum segment containing a). Giving a rightward program is a bit difficult, and someone may think of the following program.

$$\begin{array}{lcl}
mis([]) & = & 0 \\
mis(x +\!\!+ [a]) & = & mis(x) \uparrow sum(x +\!\!+ [a])
\end{array}$$

That is, the maximum initial-segment sum of a list $x +\!\!+ [a]$ is either the maximum initial segment sum of x or the whole list $x +\!\!+ [a]$. Although the program above is correct, it is not a rightward program of *mis* because of the call of another function *sum*. *Tupling transformation* [Fok89, Chi93, HITT97] is effective for such situations.

Consider a function $ms(x) \stackrel{\text{def}}{=} (mis(x), sum(x))$; then ms is rightward, as the following calculations show.

$$\begin{array}{lcl} ms([\,]) & = & \{ \text{ definition of } ms \ \} \\ & & (mis([\,]), sum([\,])) \\ & = & \{ \text{ definition of } mis \text{ and } sum \ \} \\ & & (0,0) \\ ms(x +\!\!+ [a]) & = & \{ \text{ definition of } ms \ \} \\ & & (mis(x +\!\!+ [a]), sum(x +\!\!+ [a])) \\ & = & \{ \text{ definition of } mis \text{ and } sum \ \} \\ & & (mis(x) \uparrow (sum(x) + a), sum(x) + a) \\ & = & \{ \text{ definition of } ms \ \} \\ & & \mathbf{let}\ (i,s) = ms(x)\ \mathbf{in}\ (i \uparrow (s+a), s+a) \end{array}$$

Besides, the function ms is leftward, which is a direct consequence that both mis and sum are leftward. In summary, ms is both leftward and rightward.

$$\begin{array}{lcl} ms([\,]) & = & 0 \\ ms([a] +\!\!+ x) & = & \mathbf{let}\ (i,s) = ms(x)\ \mathbf{in}\ (0 \uparrow (a+i),\ a+s) \\ ms(x +\!\!+ [a]) & = & \mathbf{let}\ (i,s) = ms(x)\ \mathbf{in}\ (i \uparrow (s+a),\ s+a) \end{array}$$

Therefore, we will consider deriving a parallel program for ms.

What is a right inverse of ms? Given two values, say i and s, a right inverse of ms computes a list whose maximum initial-segment sum is i and sum is s. The following function $ms^\bullet$ might be a right inverse of ms.

$$ms^\bullet(i,s) \stackrel{\text{def}}{=} [i, s-i]$$

The sum of $[i, s-i]$ is certainly s, and the maximum initial-segment sum is hopefully i. Actually $ms^\bullet$ is a right inverse of ms, as the following calculation shows.

$$\begin{array}{lcl} ms(ms^\bullet(ms(x))) & = & \{ \text{ definition of } ms \text{ and } ms^\bullet \ \} \\ & & ms([mis(x), sum(x) - mis(x)]) \\ & = & \{ \text{ definition of } ms \ \} \\ & & (0 \uparrow mis(x) \uparrow sum(x), sum(x)) \\ & = & \{ \text{ claim: } 0 \le mis(x) \wedge mis(x) \ge sum(x) \ \} \\ & & (mis(x), sum(x)) \\ & = & \{ \text{ definition of } ms \ \} \\ & & ms(x) \end{array}$$

It is not difficult to confirm the claim $0 \le mis(x) \wedge mis(x) \ge sum(x)$. $0 \le mis(x)$ is evident from the definition of mis, and $mis(x) \ge sum(x)$ can be easily proved by induction.

Now that we have obtained a right inverse of ms, we can derive a list homomorphism $\mathsf{hom}_{\odot,\phi}$ that is equivalent to ms.

$$
\begin{array}{rcl}
\phi(a) & = & \{\ \text{Lemma 6.3}\ \} \\
 & & ms([a]) \\
 & = & \{\ \text{definition of } ms\ \} \\
 & & (0 \uparrow a, a) \\
(i_1, s_1) \odot (i_2, s_2) & = & \{\ \text{Lemma 6.3}\ \} \\
 & & ms(ms^\bullet(i_1, s_1) +\!\!\!+\ ms^\bullet(i_2, s_2)) \\
 & = & \{\ \text{definition of } ms^\bullet\ \} \\
 & & ms([i_1, s_1 - i_1, i_2, s_2 - i_2]) \\
 & = & \{\ \text{definition of } ms\ \} \\
 & & (0 \uparrow i_1 \uparrow s_1 \uparrow (s_1 + i_2) \uparrow (s_1 + s_2), s_1 + s_2) \\
 & = & \{\ \text{claim: } 0 \le i_1 \wedge i_1 \ge s_1 \wedge i_2 \ge s_2\ \} \\
 & & (i_1 \uparrow (s_1 + i_2), s_1 + s_2)
\end{array}
$$

It is sufficient to consider the case where both operands of $\odot$ are in the range of ms, and thus, the claim certainly holds. In summary, the following parallel program is obtained.

$$
\begin{array}{l}
mis(x) = \pi_1(ms(x)) \\
\begin{array}{lcl}
ms([]) & = & (0, 0) \\
ms([a]) & = & (0 \uparrow a, a) \\
ms(x +\!\!\!+\ y) & = & ms(x) \odot ms(y) \\
(i_1, s_1) \odot (i_2, s_2) & = & (i_1 \uparrow (s_1 + i_2), s_1 + s_2)
\end{array}
\end{array}
$$

We have developed the operator $\odot$ with no attention to its associativity. Although the associativity of $\odot$ is not apparent, Theorem 6.2 and Lemma 6.3 guarantee the associativity of $\odot$. This is the effectiveness of the third list-homomorphism theorem.

Maximum Segment Sum

Next, we consider a bit more complicated example, the maximum segment sum problem seen in Section 3.3.

We first try writing a program to compute maximum segment sum of a list in both leftward and rightward manner.

$$
\begin{array}{lcl}
mss([]) & = & 0 \\
mss([a] +\!\!\!+\ x) & = & mis([a] +\!\!\!+\ x) \uparrow mss(x) \\
mss(x +\!\!\!+\ [a]) & = & mss(x) \uparrow mts(x +\!\!\!+\ [a])
\end{array}
$$

In the program above, mts is a function that computes the maximum tail-segment sum, which is the dual problem of the maximum initial segment sum problem.

As similar to the case of mis, the program of mss above is neither leftward nor rightward, and thus, we apply tupling transformation. mss requires mis and mts,

and recall that *mis* requires *sum* for its rightward definition; thus we consider a function $mmms(x) \stackrel{\text{def}}{=} (mss(x), mts(x), mis(x), sum(x))$, which is both leftward and rightward.

$$
\begin{array}{lcl}
mmms([\,]) & = & (0,0,0,0) \\
mmms([a] +\!\!+\, x) & = & \textbf{let } (m,t,i,s) = mmms(x) \\
 & & \textbf{in } (0 \uparrow (a+i) \uparrow m,\ (a+s) \uparrow t,\ 0 \uparrow (a+i),\ a+s) \\
mmms(x +\!\!+\, [a]) & = & \textbf{let } (m,t,i,s) = mmms(x) \\
 & & \textbf{in } (m \uparrow (t+a) \uparrow 0,\ (t+a) \uparrow 0,\ i \uparrow (s+a),\ s+a)
\end{array}
$$

Now we would like to derive its parallel program based on Lemma 6.3. However, it is not trivial at all to develop a right inverse of *mmms*. A right inverse of *mmms* takes four values, say m, t, i, and s, and returns a list whose sum is s, maximum initial-segment sum is i, maximum tail-segment sum is t, and maximum segment sum is m. In fact, the following function $mmms^\bullet$ is a right inverse of *mmms*.

$$mmms^\bullet(m,t,i,s) = [i, s-i-t, m, t-m]$$

Let us confirm its correctness.

$$
\begin{array}{ll}
\multicolumn{2}{l}{mmms(mmms^\bullet(mmms(x)))} \\
= & \{ \text{ definition of } mmms \text{ and } mmms^\bullet \ \} \\
 & mmms([mis(x), sum(x)-mis(x)-mts(x), mss(x), mts(x)-mss(x)]) \\
= & \{ \text{ definition of } mmms \ \} \\
 & (0 \uparrow mis(x) \uparrow (sum(x)-mis(x)-mts(x)) \uparrow mss(x) \uparrow (mts(x)-mss(x)) \uparrow \\
 & \ (sum(x)-mts(x)) \uparrow (sum(x)-mis(x)-mts(x)+mss(x)) \uparrow mts(x) \uparrow \\
 & \ (sum(x)-mts(x)+mss(x)) \uparrow (sum(x)-mis(x)) \uparrow sum(x)), \\
 & \ sum(x) \uparrow (sum(x)-mis(x)) \uparrow mts(x) \uparrow (mts(x)-mss(x)) \uparrow 0, \\
 & \ 0 \uparrow mis(x) \uparrow (sum(x)-mts(x)) \uparrow (sum(x)-mts(x)+mss(x)) \uparrow sum(x), \\
 & \ sum(x)) \\
= & \left\{ \text{claims: } \begin{array}{l} 0 \le mis(x) \le mss(x),\ 0 \le mts(x) \le mss(x), \\ sum(x) \le mis(x), \text{ and } sum(x) \le mts(x) \end{array} \right\} \\
 & (mss(x),\ mts(x),\ mis(x) \uparrow (sum(x)-mts(x)+mss(x)),\ sum(x)) \\
= & \{ \text{ claim: } sum(x)+mss(x) \le mis(x)+mts(x) \ \} \\
 & (mss(x),\ mts(x),\ mis(x),\ sum(x)) \\
= & \{ \text{ definition of } mmms \ \} \\
 & mmms(x)
\end{array}
$$

We have required claims two times in the calculation. While the former is evident, the latter is nontrivial. Let us prove the latter by induction. $sum([\,]) + mss([\,]) \le mis([\,]) + mts([\,])$ holds apparently. Now assume that $sum(x) + mss(x) \le mis(x) + mts(x)$ holds; then, $sum([a] +\!\!+\, x) + mss([a] +\!\!+\, x) \le mis([a] +\!\!+\, x) + mts([a] +\!\!+\, x)$ is

proved by the following calculation.

$$
\begin{array}{ll}
& \mathit{sum}([a] +\!\!+\, x) + \mathit{mss}([a] +\!\!+\, x) \\
= & \{ \text{ definition of } \mathit{sum}, \mathit{mss}, \text{ and } \mathit{mis} \ \} \\
& a + \mathit{sum}(x) + (0 \uparrow (a + \mathit{mis}(x)) \uparrow \mathit{mss}(x)) \\
= & \{ \text{ distributivity of } + \text{ over } \uparrow \ \} \\
& (a + \mathit{sum}(x)) \uparrow (a + \mathit{sum}(x) + a + \mathit{mis}(x)) \uparrow (a + \mathit{sum}(x) + \mathit{mss}(x)) \\
\leq & \{ \text{ hypothesis } \} \\
& (a + \mathit{sum}(x)) \uparrow (a + \mathit{sum}(x) + a + \mathit{mis}(x)) \uparrow (a + \mathit{mis}(x) + \mathit{mts}(x)) \\
= & \{ \text{ distributivity of } + \text{ over } \uparrow \ \} \\
& (a + \mathit{sum}(x)) \uparrow (a + \mathit{mis}(x) + ((a + \mathit{sum}(x)) \uparrow \mathit{mts}(x))) \\
= & \{ \text{ definition of } \mathit{mts} \ \} \\
& (a + \mathit{sum}(x)) \uparrow (a + \mathit{mis}(x) + \mathit{mts}([a] +\!\!+\, x)) \\
\leq & \{ \ a + \mathit{sum}(x) = \mathit{sum}([a] +\!\!+\, x) \leq \mathit{mts}([a] +\!\!+\, x) \ \} \\
& \mathit{mts}([a] +\!\!+\, x) \uparrow (a + \mathit{mis}(x) + \mathit{mts}([a] +\!\!+\, x)) \\
= & \{ \text{ distributivity of } + \text{ over } \uparrow \ \} \\
& (0 \uparrow (a + \mathit{mis}(x))) + \mathit{mts}([a] +\!\!+\, x) \\
= & \{ \text{ definition of } \mathit{mis} \ \} \\
& \mathit{mis}([a] +\!\!+\, x) + \mathit{mts}([a] +\!\!+\, x)
\end{array}
$$

In summary, the function $\mathit{mmms}^\bullet$ is certainly a right inverse of mmms.

Lastly, we derive a parallel program of $\mathit{mmms} = \mathsf{hom}_{\odot,\phi}$ based on Lemma 6.3.

$$
\begin{array}{lll}
\phi(a) & = & \{ \text{ Lemma 6.3 } \} \\
& & \mathit{mmms}([a]) \\
& = & \{ \text{ definition of } \mathit{mmms} \ \} \\
& & (a \uparrow 0, \ a \uparrow 0, \ a \uparrow 0, a)
\end{array}
$$

$$
\begin{array}{ll}
& (m_1, t_1, i_1, s_1) \odot (m_2, t_2, i_2, s_2) \\
= & \{ \text{ Lemma 6.3 } \} \\
& \mathit{mmms}(\mathit{mmms}^\bullet(m_1, t_1, i_1, s_1) +\!\!+\, \mathit{mmms}^\bullet(m_2, t_2, i_2, s_2)) \\
= & \{ \text{ definition of } \mathit{mmms}^\bullet \ \} \\
& \mathit{mmms}([i_1, s_1 - i_1 - t_1, m_1, t_1 - m_1, i_2, s_2 - i_2 - t_2, m_2, t_2 - m_2]) \\
= & \{ \text{ definition of } \mathit{mmms} \text{ (the detailed calculation is omitted) } \} \\
& (m_1 \uparrow m_2 \uparrow (t_1 + i_2), \ (t_1 + s_2) \uparrow t_2, \ i_1 + (s_1 + i_2), \ s_1 + s_2)
\end{array}
$$

We have omitted the detailed calculation of the last step, in which we simply the definition of the operator $\odot$. It is not very difficult, but too lengthy to describe on papers. In any case, the obtained parallel program is exactly the known efficient parallel program of the maximum segment sum problem [Col94, Gor96, HIT97]. As the same as the previous case, the derived operator $\odot$ is proved to be associative by its construction.

So far, we have developed a parallel program for the maximum segment sum problem. Although our development is systematic, there are many difficult steps: developing a right inverse, verifying the correctness of the right inverse, and simplifying the operators raised from Lemma 6.3. Systems to aid in dealing with such difficulties are called for.

Table 6.1. Execution times of list homomorphisms (unit: millisecond)

#processors	sequential	1	2	4	8	16	32	48	64
max	313	697	369	186	94	52	32	118	118
mis	329	540	270	137	68	34	26	33	107
mss	353	1299	652	331	165	98	50	40	39
atoi	414	1599	803	401	200	106	59	52	38

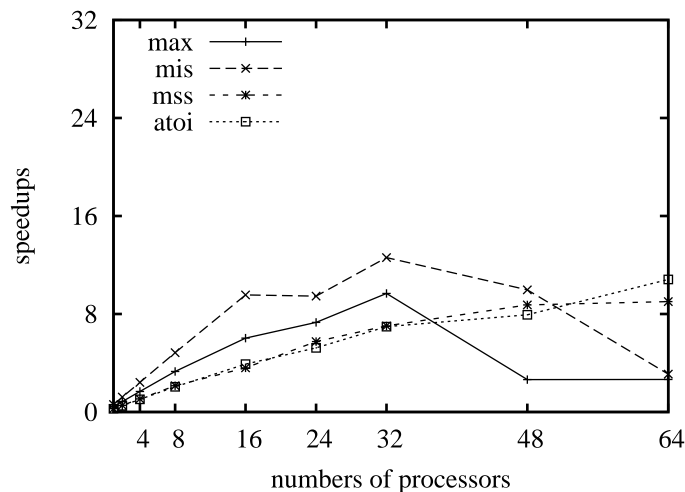

Figure 6.1. Speedup ratios against sequential implementations

6.1.4 Experiments

To demonstrate the scalability of list homomorphisms, we did some experiments. It is worth noting that list homomorphisms do not rely on a specific parallel programming environment. What list homomorphisms guarantees is correctness of divide-and-conquer algorithms. In other words, efficiency of parallel programs depends on the implementation of list homomorphisms. We adopted C++ parallel skeleton library SkeTo [MIEH06], in which we could use list homomorphisms directly as components of parallel programs. We can make use of other parallel programming frameworks, such as OpenMP [CJdP07] and Intel threading building blocks [Rei07], by implementing list homomorphisms on them.

We prepared four examples: *max* that finds the maximum value in a list, *mis* that computes the maximum prefix sum of a list, *mss* that computes the maximum segment sum of a list, and *atoi* that converts a sequence of characters into a decimal number. For comparison, we prepared a hand-written sequential program for each example. We used a PC-cluster of uniform PCs connected with Gigabit Ethernet, each of which consisted of dual Xeon 2.8 GHz CPUs and a 2 GB shared memory. The OS, compiler, and libraries used were respectively Linux 2.6.18-AMD64, gcc 4.1.2, MPICH 1.2.7-p1, and SkeTo unpublished developers' version. The input of

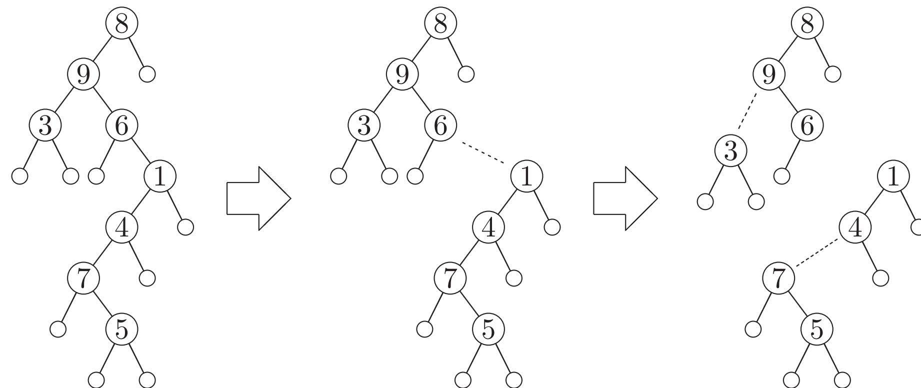

Figure 6.2. Dividing a binary tree aggressively

each program is an array of 2^{27} length containing integers.

Table 6.1 shows execution times with respect to numbers of processors, and the speedup ratios against sequential programs are plotted in Figure 6.1. The time for initial data distributions are excluded from execution times.

First, the parallel programs written with SkeTo library are slower than sequential implementations when both are evaluated on a single processor. One reason is the overhead of the use of SkeTo, and another is the overhead of parallelization. As an example, consider the case of *mis*. The parallel program for *mis* computes two values, while the sequential program of *mis* computes only one value. Therefore, it is natural that the parallel program is about two times slower than the sequential one.

Against the overhead, list homomorphisms are certainly scalable with respect to the number of processors. The parallel implementations are faster than sequential implementations when more than four processors are available; moreover, the implementations show good speedup until no more speedup is practically hopeless.

6.2 Parallel Programming on Binary Trees

Next, let us consider parallel programming on binary trees.

As an example, consider summing up all values in a tree. Someone may think of a divide-and-conquer parallel algorithm raised by subtree structures, that is, computing independent subtrees in parallel. However, such a naive parallel algorithm is not generally scalable with respect to numbers of processors. Its speedup is limited by the height of the input tree, and thus, it is not cost optimal, especially when the input tree is ill-balanced. To obtain better scalability, we need to introduce more aggressive divisions, like Figure 6.2. In this case, aggressive divisions yield a scalable parallel algorithm, which computes the summation in time logarithmic in the size of the tree with a sufficient number of processors.

Although the aggressive divisions bring a cost-optimal divide-and-conquer par-

allel summation algorithm, there are mainly two difficulties to generalize the algorithm. One is the algorithm to find appropriate division. The key to cost-optimal parallel computations is a division that yields two structures of almost the same size. However, it is nontrivial to find the "center" of a tree efficiently. The other difficulty is that a division raises a structure that is not a binary tree. Recall Figure 6.2. After the divisions, the upper part (containing 8, 9, and 6) and the right part (containing 1 and 4) are not binary trees. Therefore, for computation based on the aggressive divisions, we should specify computations on a structure that is not a binary tree. While there are no problems when we summing up values, it is problematic when we consider more complicated computations on trees.

First, in Section 6.2.1, we review a cost-optimal parallel algorithm called *parallel tree contraction algorithm*, and develop divide-and-conquer parallel algorithms based on the parallel tree contraction algorithm in Section 6.2.2. After that, in Section 6.2.3, we introduce the third list-homomorphism theorem on binary trees, and show some examples and experiments in Section 6.2.4.

6.2.1 Parallel Tree Contraction

In this section, we review *parallel tree contraction*, a framework of constructing efficient parallel algorithms on trees. The parallel tree contraction problem is a problem to provide a scheduling of contraction operations so that they can collapse a tree efficiently in parallel with no conflict. Once an efficient parallel tree contraction algorithm is developed, we can achieve many computations on a tree efficiently in parallel by processing computations according to the schedule of contraction operations. First, Miller and Reif [MR85] introduced the notion of parallel tree contraction to develop an efficient parallel algorithm for evaluating expressions defined with $+$, $-$, $\times$, and $/$. After that, parallel tree contraction is recognized as an important framework for constructing various parallel algorithms on trees. Many studies have been done for efficient parallel tree contraction algorithms [CV88, GR89, ADKP89, Rei93, MW97] and for implementation of various computations on them [DK92, GCS94, Ski96, MHT06, Mat07b].

Let us review a parallel tree contraction algorithm proposed by Abrahamson et al. [ADKP89], which is called the SHUNT[1] contraction algorithm. The problem is to collapse a tree in $O(\log n)$ steps, where each step consists of a set of independent SHUNT operations defined as follows.

Definition 6.4 (SHUNT)**.** Specified a leaf, a SHUNT operation removes the leaf and its parent and connects the sibling of the leaf to its grandparent. □

Figure 6.3 shows the behavior of a SHUNT operation. We call two SHUNT operations are independent if no nodes simultaneously concern both of them. On one hand, we require the SHUNT operations to be independent; otherwise, the

[1] The name "SHUNT" is later given in [Rei93].

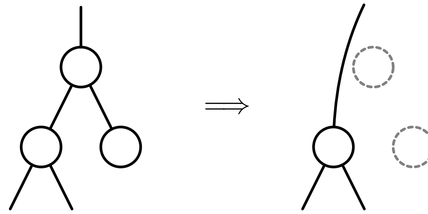

Figure 6.3. A SHUNT operation

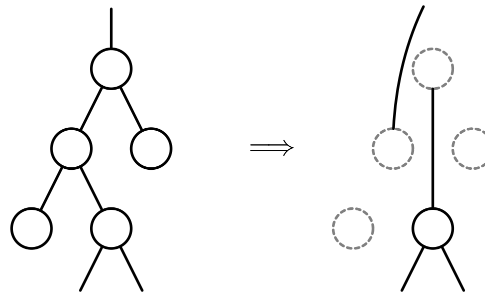

Figure 6.4. A conflict of two SHUNT operations

tree structure will be broken as Figure 6.4. On the other hand, we would like to apply many SHUNT operations simultaneously so as to accomplish the reduction in $O(\log n)$ steps. Therefore, we need to develop a good conflict-free scheduling of SHUNT operations. Abrahamson et al. [ADKP89] showed that a numbering on leaves resolves this problem.

Procedure 6.5 (SHUNT contraction for binary trees)**.**
(1) Number all leaves from left to right.
(2) Do SHUNT for all odd-numbered left leaves.
(3) Do SHUNT for all odd-numbered right leaves.
(4) Halve all numbers of leaves.
(5) Go to (2) until the tree consists of only one node. □

Throughout the Procedure 6.5, all SHUNT operations in a step are independent. Moreover, the Procedure 6.5 finishes the collapse of a tree of n nodes in $O(\log n)$ steps, because steps (2) and (3) halve the number of leaves.

Next, let us consider computations on binary trees rather than a collapse. Here we introduce *algebraic computations* as a general computation pattern on trees, which are computations to evaluate tree-shaped expressions. Note that algebraic computations form not only binary trees but also non-binary trees.

Definition 6.6 (algebraic computations [ADKP89])**.** A set of values S and a set of functions F are given, where each element of F takes a fixed-sized tuple of elements of S according to its arity and results in an element of S. An *algebraic computation* defined by (S, F) is a computation to evaluate expressions whose operators are elements of F and values are elements of S. □

Abrahamson et al. gave a sufficient condition for parallelizing algebraic computations when the arity of each function is two.

Theorem 6.7 ([ADKP89])**.** Assume that there are a set of values S and two sets of indexed functions $F : (S \times S) \to S$ and $G : S \to S$ such that the following conditions hold.

- Any element of F and G can be evaluated in O(1) time.
- For all $f_i \in F$ and $a \in S$, there exist functions $g_j, g_k \in G$ such that $f_i(x, a) = g_j(x)$, $f_i(a, x) = g_k(x)$, and the indexes j and k can be computed in O(1) time from i and a.
- For all $g_i, g_j \in G$, there exists a function $g_k \in G$ such that $g_i(g_j(x)) = g_k(x)$ holds and the index k can be computed in O(1) time from i and j.

Then, an algebraic computation defined by (S, F) can be computed in $\mathrm{O}(n/p + \log p)$ time on an EREW PRAM with p processors, where n is the size of the expression. □

Consider an algebraic computation as a tree whose internal nodes are functions in F and leaves are values in S. Then, evaluating the algebraic computation is equivalent to collapse the tree with computing the value of the tree. Therefore, we can perform the computation efficiently in parallel according to the scheduling raised by Procedure 6.5, if we can process computations corresponding SHUNT operations. Here, notice that each function $g_i \in G$ corresponds to a one-hole context of a tree representing an algebraic computation. Then, the third premise of Theorem 6.7 can be recognized as the requirement that we can merge two one-hole contexts into one, which is the operation that the SHUNT operation exactly does. From a computational viewpoint, each $g_i \in G$ corresponds to a continuation. We cannot accomplish the whole computation for a one-hole context and its continuation is left as a function. The third premise means that we can perform computations on such continuations in the sense that a composition of two continuations yields a continuation of the same size. Theorem 6.7 states that we can obtain a cost-optimal parallel computation if we can perform such computation on continuations.

6.2.2 m-Bridges

While the SHUNT contraction algorithm is cost optimal, it is not a divide-and-conquer algorithm. Moreover, the algorithm might not be efficient enough for practical use, especially for parallel computation on distributed-memory environments. The reason of inefficiency is lack of locality. On distributed-memory environments, it is important to arrange data in advance so that each processor can achieve its computation with a little communication to other processors. However, in the SHUNT contraction algorithm, it is difficult to predict which nodes will be processed by a processor, and thus, it seems necessary that processors should communicate each other for every contraction step.

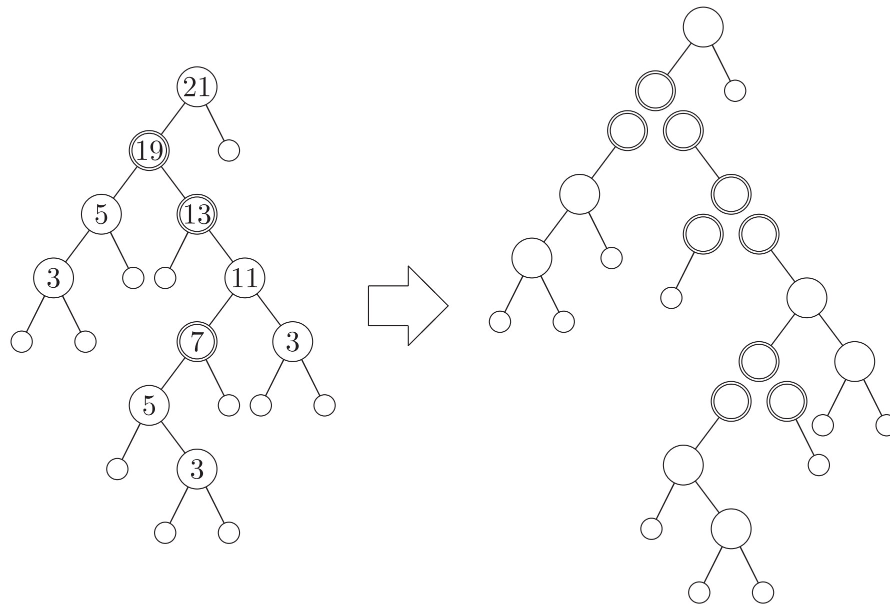

Figure 6.5. 6-critical nodes and 6-bridge: the number on each internal node shows the size of the subtree rooted by the node, and nodes of concentric circles are 6-critical nodes.

To resolve the problem, here we would like to introduce the *m-bridge* technique [Rei93, Mat07b], which enables us to divide a tree efficiently into almost the same size and develop cost-optimal divide-and-conquer parallel algorithms on binary trees.

First, we introduce the notion of m-bridges.

Definition 6.8 (m-critical node)**.** Given a natural number $m > 1$, an internal node v in a rooted tree is said to be *m-critical* if each child of v, say v', satisfies $\lceil \mathit{size}(v)/m \rceil > \lceil \mathit{size}(v')/m \rceil$, where $\mathit{size}(v)$ denotes the size of the subtree rooted by v. □

Definition 6.9 (m-bridge)**.** For a tree t and a natural number $m > 1$, an *m-bridge* of t is one of the largest connected subgraph of t such that its internal nodes except its root are not m-critical. □

Figure 6.5 shows m-critical nodes and m-bridges of a trees. It is worth noting that m-critical nodes of a tree of n nodes can be computed on $\mathrm{O}(n/p + \log p)$ time on an EREW PRAM with p processors [Rei93].

The following lemmas show important properties of m-critical nodes and m-bridges.

Lemma 6.10 ([Rei93])**.** The common ancestor of two m-critical nodes is m-critical. □

Lemma 6.11 ([Rei93])**.** The size of each m-bridge is at most $m+1$. □

Lemma 6.12 ([Rei93])**.** The number of m-critical nodes in a tree of n nodes is at most $2n/m-1$ for $n>m$. □

For an at most k-ary tree of n nodes, consider the m-bridge of the tree, where $m=\lceil n/p\rceil$ and p is the number of processors. From Lemmas 6.11 and 6.12, the size of each bridge is at most $m+1$, namely around n/p, and the number of bridges is at most $2kn/m-1$, namely around $2kp$. These bounds indicate that there will be sufficiently many bridges of sufficiently large size. Therefore, we can fulfill load balancing by distributing bridges to processors. The observation is formalized as the following lemma.

Lemma 6.13. There exists a partition $B_1, B_2, \ldots, B_p$ of $\lceil n/p\rceil$-bridges of an at most k-ary tree of n-nodes such that the partition satisfies the following conditions, if $n \geq 4p^2 > 1$ holds, where $\#B$ denotes the summation of sizes of bridges in B.

- For all $1 \leq i, j \leq p$, $\#B_i \leq 2 \times \#B_j$ holds.
- For all $1 \leq i \leq p$, B_i contains at least one bridge whose size is larger than or equal to $n/(4k-2)p^2$.

Proof. We prove the first condition by contradiction. Assume that $\#B_i > 2 \times \#B_j$ holds. Without loss of generality, we assume that $\#B_i$ and $\#B_j$ are respectively the largest and smallest ones. If B_i contains more than two bridges, then moving its smallest bridge to B_j will provide a better partition. Thus, we can assume that B_i consists only one bridge, and from Lemma 6.11, $\#B_i \leq \lceil n/p\rceil + 1$ holds. Since B_i is the largest one,

$$\sum_{1\leq k\leq p} \#B_k < \frac{\#B_i}{2} + (p-1)\#B_i < \left(p-\frac{1}{2}\right)\left(\frac{n}{p}+2\right) = n + 2p - \frac{n}{2p} - 1 < n$$

holds. However, since $B_1, B_2, \ldots, B_p$ is a partition of $\lceil n/p\rceil$-bridges of a tree of n nodes, $\Sigma_{1\leq k\leq p}\#B_k$ must be larger than n, and a contradiction occurs.

Next, we prove the second condition by contradiction. From the first condition, each B_j satisfies $\#B_j > n/2p$. Now assume that the size of the largest bridge of B_j is less than $n/(4k-2)p^2$. Then, since each B_j retains at least one bridge, for the number of bridges m,

$$m > (p-1) + \frac{n/2p}{n/(4k-2)p^2} = (p-1) + (2k-1)p = 2kp - 1 \geq \frac{2kn}{\lceil n/p\rceil} - 1$$

holds. However, from Lemma 6.12, the number of bridges m is at most $2kn/\lceil n/p\rceil - 1$, and a contradiction occurs. □

As seen, the m-bridge technique provides a clever way to distribute trees to processors. Therefore, we can obtain efficient divide-and-conquer parallel algorithms,

if we can perform computations on bridges. However, a bridge is not a complete subtree of the original tree and it seems difficult to perform computation on bridges. Fortunately, we can provide a respectable sufficient condition for achieving computations on bridges. Recall Theorem 6.7, in which computations on one-hole contexts are considered. Each bridge contains at most two critical nodes from Lemma 6.10, and one is its root whenever a bridge has two critical nodes. Thus, each bridge is either a tree or a one-hole context when we neglect all critical nodes and edges connecting critical nodes. Therefore, it is sufficient for computations on bridges to perform computations on one-hole contexts, which is the premise of Theorem 6.7.

In summary, the m-bridge technique enables us to develop efficient divide-and-conquer parallel algorithms on binary trees, when the premise of Theorem 6.7 is fulfilled. Matsuzaki [Mat07b, Mat07a] gave more detailed discussions including implementation techniques.

6.2.3 The Third List-Homomorphism Theorem on Binary Trees

So far, we have considered parallel algorithms on trees. Here we would like to develop parallel programming on node-valued binary trees.

$$\begin{array}{rcl} \textbf{data}\ \mathit{Tree}_A & = & \mathit{Leaf} \\ & | & \mathit{Node}(A, \mathit{Tree}_A, \mathit{Tree}_A) \end{array}$$

The goal of this subsection is to develop "the third list-homomorphism theorem" for node-valued binary trees. Recall the third list-homomorphism theorem, which states that a function is a list homomorphism if and only if it is both leftward and rightward. "The third list-homomorphism theorem" on tree states that we can efficiently compute a function in a divide-and-conquer manner if it is both *downward* and *upward*. For this purpose, it is necessary to formalize downward computations, upward computations, and divide-and-conquer parallel computations on trees.

As seen, divide-and-conquer parallel computations on trees can be formalized by computations on one-hole contexts. Once computations on one-hole contexts are specified, the m-bridge technique yields divide-and-conquer parallel computations. To represent one-hole contexts, we use the notion of *Huet's zippers* [Hue97]. Zippers also provide a representation of paths, and thus, we can formalize downward and upward computations based on zippers.

Now let us introduce zippers. A zipper is a list whose elements are contexts left after a walk. According to a downward walking from the root of a tree, we construct a zipper as follows: When we go down-right from a node, we add its left child to the zipper; When we go down-left, we add the right child to the zipper. For example, Figure 6.6 shows a correspondence between a zipper and a walk from the root to the black node.

The zipper structures for node-valued binary trees can be specified by the following type, where components of the sum type respectively correspond to the left

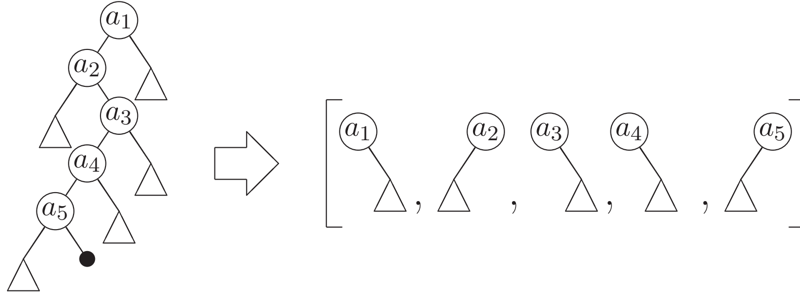

Figure 6.6. A zipper structure, which expresses a path from the root to the black leaf

child and the right child.

$$\mathbf{type}\ \mathit{Zipper}_A = ((A, \mathit{Tree}_A) + (A, \mathit{Tree}_A))^*$$

When a walk ends at a leaf, a zipper stores the whole tree. The following function *z2t* restores a tree from a zipper.

$$\begin{array}{lcl} \mathit{z2t}([\,]) & = & \mathit{Leaf} \\ \mathit{z2t}([\mathsf{L}(n,l)] +\!\!+ r) & = & \mathit{Node}(n, l, \mathit{z2t}(r)) \\ \mathit{z2t}([\mathsf{R}(n,r)] +\!\!+ l) & = & \mathit{Node}(n, \mathit{z2t}(l), r) \end{array}$$

When a walk ends at an internal node, a zipper corresponds to a one-hole context of a tree. Look at Figure 6.6 again. On one hand, the zipper represents the tree with its path from the root to the black leaf. On the other hand, the zipper also represents the one-hole context in which the black circle represents the hole. We basically regard a zipper as a tree and call the hole *terminal leaf*.

We would like to summarize correspondences of a zipper, a tree (or a one-hole context), and a path: A zipper corresponds to a tree with a terminal leaf (a one-hole context) or a path from the root to the terminal leaf; An initial segment of a zipper corresponds to a one-hole context containing the root or a path from the root to a node; A tail segment of a zipper corresponds to a subtree containing the terminal leaf (a one-hole context whose hole is the same as the hole of the original one) or a path from a node to the terminal leaf.

Next we formalize downward and upward computations on binary trees.

Consider the following function *sumTree*, which sums up all values in a tree.

$$\begin{array}{lcl} \mathit{sumTree}(\mathit{Leaf}) & = & 0 \\ \mathit{sumTree}(\mathit{Node}(n,l,r)) & = & n + \mathit{sumTree}(l) + \mathit{sumTree}(r) \end{array}$$

First we would like to develop its downward version. Since an initial segment of a zipper corresponds to a path from the root to an internal node, the following function $\mathit{sumTree}_\downarrow$ is downward in the sense that it performs its computation from the root to the terminal leaf.

$$\begin{array}{lcl} \mathit{sumTree}_\downarrow([\,]) & = & 0 \\ \mathit{sumTree}_\downarrow(x +\!\!+ [\mathsf{L}(n,l)]) & = & \mathit{sumTree}_\downarrow(x) + n + \mathit{sumTree}(l) \\ \mathit{sumTree}_\downarrow(x +\!\!+ [\mathsf{R}(n,r)]) & = & \mathit{sumTree}_\downarrow(x) + n + \mathit{sumTree}(r) \end{array}$$

Notice that we use the function *sumTree*, because types of *sumTree* and $sumTree_\downarrow$ are different: *sumTree* has the type $Tree_\mathbb{R} \to \mathbb{R}$ and computes the summation of values in a tree; $sumTree_\downarrow$ has the type $Zipper_\mathbb{R} \to \mathbb{R}$ and computes the summation of values in a one-hole context.

Similarly, we can develop its upward version. Since a tail segment of a zipper corresponds to a path from an internal node to the terminal leaf, the following function $sumTree_\uparrow$ traverses a tree from its terminal leaf to its root.

$$
\begin{array}{lcl}
sumTree_\uparrow([\,]) & = & 0 \\
sumTree_\uparrow([\mathsf{L}(n,l)] +\!\!\!+ x) & = & n + sumTree(l) + sumTree_\uparrow(x) \\
sumTree_\uparrow([\mathsf{R}(n,r)] +\!\!\!+ x) & = & n + sumTree_\uparrow(x) + sumTree(r)
\end{array}
$$

In this case, $sumTree_\uparrow$ and $sumTree_\downarrow$ are equivalent to *sumTree* in the sense that $sumTree_\uparrow = sumTree_\downarrow = sumTree \circ z2t$ holds. However, computations on a path may require more information than those on trees. To formalize correspondences between computations on paths and those on trees, we introduce a notion of *path-based computations*.

Definition 6.14 (path-based computation on binary trees)**.** A path-based computation of a function $h : Tree_A \to B$ is a function $h' : Zipper_A \to C$ such that there exists a function $\psi : C \to B$ satisfying the following equation.

$$\psi \circ h' = h \circ z2t \qquad \square$$

The equation above means that h' simulates the computation of h and the result is extracted by ψ. Note that *z2t* is a path-based computation of any function; however it is useless in practice because no significant computations are managed on paths. In other words, it is important to specify an appropriate path-based computation.

Based on the notion of path-based computations, downward and upward computations are defined as follows.

Definition 6.15 (downward computations on binary trees)**.** A path-based computation of $h : Tree_A \to B$, say $h' : Zipper_A \to C$, is said to be *downward* if there exists an operator $(\otimes) : (C \times ((A \times B) + (A \times B))) \to C$ such that the following equations hold.

$$
\begin{array}{lcl}
h'(x +\!\!\!+ [\mathsf{L}(n,t)]) & = & h'(x) \otimes \mathsf{L}(n,h(t)) \\
h'(x +\!\!\!+ [\mathsf{R}(n,t)]) & = & h'(x) \otimes \mathsf{R}(n,h(t))
\end{array} \qquad \square
$$

Definition 6.16 (upward computations on binary trees)**.** A path-based computation of $h : Tree_A \to B$, say $h' : Zipper_A \to C$, is said to be *upward* if there exists an operator $(\oplus) : (((A \times B) + (A \times B)) \times C) \to C$ such that the following equations hold.

$$
\begin{array}{lcl}
h'([\mathsf{L}(n,t)] +\!\!\!+ x) & = & \mathsf{L}(n,h(t)) \oplus h'(x) \\
h'([\mathsf{R}(n,t)] +\!\!\!+ x) & = & \mathsf{R}(n,h(t)) \oplus h'(x)
\end{array} \qquad \square
$$

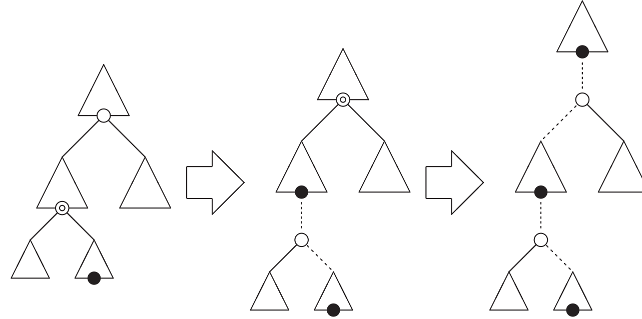

Figure 6.7. Recursive division on a one-hole context: at each step, the one-hole context is divided at the concentric circle node.

Well, we consider divide-and-conquer parallel computations on trees. For divide-and-conquer computations, we would like to divide a tree at an arbitrary place and compute each part in parallel. However, as seen in Figure 6.2, division of a tree does not yield two trees of the original type but yields a one-hole context. Instead of recursive division on trees, let us consider recursive division on one-hole contexts. As shown in Figure 6.7, select a node on a path from the root to the hole, and divide the one-hole context into three at the node: the upper part, the lower part, and a complete subtree with the node. Apparently, we can recursively divide the upper part and the lower part; in addition, we can obtain a one-hole context from the complete subtree by taking off an arbitrary leaf, which leads to recursive division on the subtree. It is worth noting that the division obtained by the m-bridge technique is an instance of divisions obtained by this procedure.

Now, let us characterize divide-and-conquer parallel programs on node-valued binary trees. We require three operations for parallel computation on trees: an operation (say $\odot$) that merges results of two one-hole contexts, an operation (say ϕ) that takes a result of a complete tree and yields the result of the complete tree with its parent, and an operation (say ψ) that computes a result of a complete tree from that of a one-hole context. Since a one-hole context corresponds to a zipper, a computation on a one-hole context can be specified by a path-based computation.

Definition 6.17 (decomposition on binary trees)**.** A *decomposition* of a function $h : \mathit{Tree}_A \to B$ is a triple $(\phi, \odot, \psi)$ that consists of an associative operator $\odot : (C \times C) \to C$ and two functions $\phi : ((A \times B) + (A \times B)) \to C$ and $\psi : C \to B$ such that

$$\psi \circ h' = h \circ \mathit{z2t}$$
$$\begin{array}{lcl} h'([\,]) & = & \iota_\odot \\ h'([\mathsf{L}(n,t)]) & = & \phi(\mathsf{L}(n, h(t))) \\ h'([\mathsf{R}(n,t)]) & = & \phi(\mathsf{R}(n, h(t))) \\ h'(x +\!\!+ y) & = & h'(x) \odot h'(y) \end{array}$$

hold, where $\iota_\odot$ is the unit of $\odot$. In this case, h is said to be *decomposable.* □

It is worth noting that a decomposition is a list homomorphism on zippers. Define a function ϕ' as $\phi'(\mathsf{L}(n,t)) = \phi(\mathsf{L}(n,h(t)))$ and $\phi'(\mathsf{R}(n,t)) = \phi(\mathsf{R}(n,h(t)))$; then, $h' = \mathsf{hom}_{\odot,\phi'}$. Therefore, associativity of $\odot$ is necessary to guarantee that the result of the computation is not affected by the node to divide the tree.

Actually decomposable functions can be efficiently evaluated in parallel based on parallel tree contraction and the m-bridge technique.

Theorem 6.18. If $(\phi, \odot, \psi)$ is a decomposition of a function h and all of ϕ, $\odot$, and ψ are constant-time computations, then h can be evaluated for a tree of n nodes in $\mathrm{O}(n/p + \log p)$ time on an EREW PRAM with p processors.

Proof. It is not difficult to prepare an algebraic computation corresponding to h such that it satisfies the premise of Theorem 6.7. □

The function *sumTree* is decomposable as the following equations show.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{sumPara} = \mathit{sumTree} \circ \mathit{z2t}} \\
\mathit{sumPara}([\,]) & = & 0 \\
\mathit{sumPara}([\mathsf{L}(n,l)]) & = & n + \mathit{sumTree}(l) \\
\mathit{sumPara}([\mathsf{R}(n,r)]) & = & n + \mathit{sumTree}(r) \\
\mathit{sumPara}(x \mathbin{+\!\!+} y) & = & \mathit{sumPara}(x) + \mathit{sumPara}(y)
\end{array}
$$

In other words, $(\phi, +, id)$ is a decomposition of *sumTree*, in which ϕ is defined by $\phi(\mathsf{L}(n,v)) = n + v$ and $\phi(\mathsf{R}(n,v)) = n + v$.

Lastly, we would like to introduce “the third list-homomorphism theorem” on node-valued binary trees, which shows a necessary and sufficient condition of existence of a decomposition.

Theorem 6.19 (the third list-homomorphism theorem on binary trees)**.** A function h is decomposable if and only if there exists a path-based computation of h that is both downward and upward.

Proof. Later we will prove a stronger theorem (Theorem 6.33). □

The statement of Theorem 6.19 is similar to the third list-homomorphism theorem. The observation underlying the theorem is that associative operators bring divide-and-conquer parallel programs not only on lists but also on trees. Moreover, as the same as the case of lists, we can provide the following lemma that is useful to develop parallel programs.

Lemma 6.20. Assume that a function h', which is a path-based computation of h satisfying $\psi \circ h' = h \circ \mathit{z2t}$, is both downward and upward; then, there exists a decomposition $(\phi, \odot, \psi)$ of h such that the following equations hold.

$$
\begin{array}{lcl}
\phi(\mathsf{L}(v,h(t))) & = & h'([\mathsf{L}(v,t)]) \\
\phi(\mathsf{R}(v,h(t))) & = & h'([\mathsf{R}(v,t)]) \\
a \odot b & = & h'(h'^{\circ}(a) \mathbin{+\!\!+} h'^{\circ}(b))
\end{array}
$$

Proof. Later we will prove a stronger lemma (Lemma 6.32). □

6.2.4 Developing Parallel Programs for Binary Trees with the Third List-Homomorphism Theorem

In this subsection, we demonstrate how to develop divide-and-conquer parallel programs based on the third list-homomorphism theorem on trees, namely Theorem 6.19 and Lemma 6.20. Our development consists of two steps. First, we seek an appropriate path-based computation that is both downward and upward. After that, we obtain a decomposition that enables us to utilize parallel tree contraction.

Summation

The first example is the function *sumTree*. Recall that a path-based computation of *sumTree* (say *st*) is both downward and upward, because of $st = sumTree_{\downarrow} = sumTree_{\uparrow}$. Therefore, Theorem 6.19 proves that there is a decomposition of *sumTree*. Lemma 6.20 shows a way to obtain a decomposition $(\phi, \odot, \psi)$. In this case, ψ is the identity function *id*, because $sumTree \circ z2t = st$ holds. Obtaining the function ϕ is easy as the following calculation shows, where C is either L or R.

$$\begin{array}{rcl} \phi(\mathsf{C}(n, sumTree(t))) & = & \{\text{ Lemma 6.20 }\} \\ & & st([\mathsf{C}(n,t)]) \\ & = & \{\text{ definition of } st \} \\ & & n + sumTree(t) \end{array}$$

Thus $\phi(\mathsf{C}(n,r)) = n + r$. The last is an associative operator $\odot$. Lemma 6.20 states that a right inverse of *st* enables us to derive the operator. It is not difficult to give a right inverse.

$$st^{\bullet}(s) = [\mathsf{L}(s, Leaf)]$$

The function $st^{\bullet}$ is a right inverse of *st*, because $st(st^{\bullet}(s)) = st([\mathsf{L}(s, Leaf)]) = s$ holds. Now we can obtain a definition of $\odot$ as follows.

$$\begin{array}{rcl} a \odot b & = & \{\text{ Lemma 6.20 }\} \\ & & st(st^{\bullet}(a) \mathbin{+\!\!+} st^{\bullet}(b)) \\ & = & \{\text{ definition of } st^{\bullet} \} \\ & & st([\mathsf{L}(a, Leaf), \mathsf{L}(b, Leaf)]) \\ & = & \{\text{ definition of } st \} \\ & & a + b \end{array}$$

We have obtained a decomposition of *sumTree*, which is exactly the same as the one we showed before.

Maximum Path Weight

Next, let us consider a small optimization problem to compute the maximum weight of paths from the root to a leaf. For simplicity, we assume that the value of each

node is non-negative. The following sequential program solves the problem.

$$\begin{array}{lcl} maxPath(Leaf) & = & 0 \\ maxPath(Node(n,l,r)) & = & n + (maxPath(l) \uparrow maxPath(r)) \end{array}$$

Our objective is to develop a parallel program to solve the problem. First, we try to obtain a downward definition, and may think of the following program.

$$\begin{array}{lcl} maxPath_{\downarrow}([]) & = & 0 \\ maxPath_{\downarrow}(x +\!\!+ [\mathsf{L}(n,l)]) & = & maxPath_{\downarrow}(x) \uparrow (pathWeight(x) + n + maxPath(l)) \\ maxPath_{\downarrow}(x +\!\!+ [\mathsf{R}(n,r)]) & = & maxPath_{\downarrow}(x) \uparrow (pathWeight(x) + n + maxPath(r)) \\ pathWeight([]) & = & 0 \\ pathWeight(x +\!\!+ [\mathsf{L}(n,l)]) & = & pathWeight(x) + n \\ pathWeight(x +\!\!+ [\mathsf{R}(n,r)]) & = & pathWeight(x) + n \end{array}$$

Notice that the function $maxPath_{\downarrow}$ is not downward, because it uses an auxiliary function $pathWeight$ that computes the weight of the path from the root to the terminal leaf. As seen in the case of lists, tupling transformation is helpful. Consider a function $maxPath'_{\downarrow}$ defined by $maxPath'_{\downarrow}(x) \stackrel{\text{def}}{=} (maxPath_{\downarrow}(x), pathWeight(x))$, which computes the maximum path weight of the tree and the weight of the path to the terminal leaf in the same time. Apparently $maxPath'_{\downarrow}$ is a path-based computation of $maxPath$; in addition, it is downward.

$$\begin{array}{lcl} maxPath'_{\downarrow}([]) & = & (0,0) \\ maxPath'_{\downarrow}(x +\!\!+ [\mathsf{L}(n,l)]) & = & \mathbf{let}\ (m,w) = maxPath'_{\downarrow}(x) \\ & & \mathbf{in}\ (m \uparrow (w + n + maxPath(l)), w + n) \\ maxPath'_{\downarrow}(x +\!\!+ [\mathsf{R}(n,r)]) & = & \mathbf{let}\ (m,w) = maxPath'_{\downarrow}(x) \\ & & \mathbf{in}\ (m \uparrow (w + n + maxPath(r)), w + n) \end{array}$$

Therefore, $maxPath'_{\downarrow}$ seems an appropriate path-based computation for $maxPath$, and we would like to give its upward definition. The following function $maxPath'_{\uparrow}$ is the upward one.

$$\begin{array}{lcl} maxPath'_{\uparrow}([]) & = & (0,0) \\ maxPath'_{\uparrow}([\mathsf{L}(n,l)] +\!\!+ x) & = & \mathbf{let}\ (m,w) = maxPath'_{\uparrow}(x) \\ & & \mathbf{in}\ (n + (m \uparrow maxPath(l)), n + w) \\ maxPath'_{\uparrow}([\mathsf{R}(n,r)] +\!\!+ x) & = & \mathbf{let}\ (m,w) = maxPath'_{\uparrow}(x) \\ & & \mathbf{in}\ (n + (m \uparrow maxPath(r)), n + w) \end{array}$$

Now that we have confirmed that the function $maxPath'_{\downarrow} = maxPath'_{\uparrow}$ (say mp) is both downward and upward, Theorem 6.19 shows existence of its parallel program, which can be derived based on Lemma 6.20.

Obtaining ϕ is straightforward, and $\phi(\mathsf{C}(n,m)) = (n+m, n)$, where C is either L or R. To obtain an associative operator $\odot$, we would like to find a right inverse of mp. It is not very difficult, and the following function $mp^{\bullet}$ is a right inverse of mp.

$$mp^{\bullet}(m,w) = [\mathsf{L}(w, Node(m-w, Leaf, Leaf))]$$

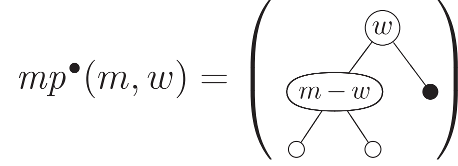

Figure 6.8. An outline of $mp^\bullet(m, w)$: the black circle represents the terminal leaf.

Figure 6.8 shows an outline of $mp^\bullet$, which would be helpful to understand $mp^\bullet$. Notice that $mp(t) = (m, w)$ implies $m \geq w$. Therefore, the function $mp^\bullet$ is certainly a right inverse of mp, because given a tree $[\mathsf{L}(w, \mathit{Node}(m - w, \mathit{Leaf}, \mathit{Leaf}))]$ where $m \geq w$, the maximum path weight of the tree is m and the path weight to the terminal leaf is w. Now we derive the operator $\odot$ as follows.

$$
\begin{array}{ll}
\multicolumn{2}{l}{(m_1, w_1) \odot (m_2, w_2)} \\
= & \{ \text{ Lemma 6.20 } \} \\
 & mp(mp^\bullet(m_1, w_1) +\!\!\!+ \; mp^\bullet(m_2, w_2)) \\
= & \{ \text{ Definition of } mp^\bullet \ \} \\
 & mp([\mathsf{L}(w_1, \mathit{Node}(m_1 - w_1, \mathit{Leaf}, \mathit{Leaf})), \mathsf{L}(w_2, \mathit{Node}(m_2 - w_2, \mathit{Leaf}, \mathit{Leaf}))]) \\
= & \{ \text{ Definition of } mp \ \} \\
 & (m_1 \uparrow (w_1 + m_2), w_1 + w_2)
\end{array}
$$

Lemma 6.20 guarantees the associativity of the derived operator $\odot$. In summary, we obtain the following parallel program.

$$
\begin{array}{l}
\mathit{maxPath} \circ \mathit{z2t} = \pi_1 \circ mp \\
\quad \begin{array}{lcl}
mp([\,]) & = & (0, 0) \\
mp([\mathsf{L}(n, t)]) & = & (n + \mathit{maxPath}(t), n) \\
mp([\mathsf{R}(n, t)]) & = & (n + \mathit{maxPath}(t), n) \\
mp(z_1 +\!\!\!+ \; z_2) & = & mp(z_1) \odot mp(z_2)
\end{array} \\
\quad (m_1, w_1) \odot (m_2, w_2) = (m_1 \uparrow (w_1 + m_2), w_1 + w_2)
\end{array}
$$

Leftmost Odd Number

The next example is a small query to find the leftmost odd number in a tree. In fact, this problem can be solved by flattening the tree into a list and considering a divide-and-conquer algorithm on the list. Here we will derive a parallel program without such clever observation.

Function $\mathit{leftOdd} : \mathit{Tree}_{\mathbb{R}} \to (\mathbb{R} \cup \mathsf{1})$ below returns the leftmost odd number in the input tree if one exists; otherwise, it returns special value $()$ that stands for emptiness.

$$
\begin{array}{lcl}
\mathit{leftOdd}(\mathit{Leaf}) & = & () \\
\mathit{leftOdd}(\mathit{Node}(n, l, r)) & = & \textbf{case } \mathit{leftOdd}(l) \textbf{ of} \\
 & & \quad () \to \textbf{if } \mathit{odd}(n) \textbf{ then } n \textbf{ else } \mathit{leftOdd}(r) \\
 & & \quad v \to v
\end{array}
$$

In the downward computation of *leftOdd*, we need to determine whether a nearly-leaf odd number is the leftmost one or not. For this purpose, we add additional information to the result: $\mathsf{L}(v)$ and $\mathsf{R}(v)$ respectively correspond to an odd number v at the left and the right of the terminal leaf.

$$\begin{array}{lcl}
\mathit{leftOdd}_{\downarrow}([]) & = & () \\
\mathit{leftOdd}_{\downarrow}(x +\!\!+ [\mathsf{L}(n,l)]) & = & \mathbf{case}\ \mathit{leftOdd}_{\downarrow}(x)\ \mathbf{of} \\
 & & \quad \mathsf{L}(v) \rightarrow \mathsf{L}(v) \\
 & & \quad a \rightarrow \mathbf{case}\ \mathit{leftOdd}(l)\ \mathbf{of} \\
 & & \qquad\qquad () \rightarrow \mathbf{if}\ \mathit{odd}(n)\ \mathbf{then}\ \mathsf{L}(n)\ \mathbf{else}\ a \\
 & & \qquad\qquad v \rightarrow \mathsf{L}(v) \\
\mathit{leftOdd}_{\downarrow}(x +\!\!+ [\mathsf{R}(n,r)]) & = & \mathbf{case}\ \mathit{leftOdd}_{\downarrow}(x)\ \mathbf{of} \\
 & & \quad \mathsf{L}(v) \rightarrow \mathsf{L}(v) \\
 & & \quad a \rightarrow \mathbf{if}\ \mathit{odd}(n)\ \mathbf{then}\ \mathsf{R}(n) \\
 & & \qquad\quad \mathbf{else\ case}\ \mathit{leftOdd}(r)\ \mathbf{of}\ () \rightarrow a \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad v \rightarrow \mathsf{R}(v)
\end{array}$$

Function $\mathit{leftOdd}_{\downarrow}$ is a path-based computation of *leftOdd*: define ψ by $\psi() = ()$ and $\psi(\mathsf{C}(v)) = v$ where C is either L or R; then $\psi \circ \mathit{leftOdd}_{\downarrow} = \mathit{leftOdd} \circ \mathit{z2t}$ holds. Next, we would like to derive its upward definition.

$$\begin{array}{lcl}
\mathit{leftOdd}_{\uparrow}([]) & = & () \\
\mathit{leftOdd}_{\uparrow}([\mathsf{L}(n,l)] +\!\!+ x) & = & \mathbf{case}\ \mathit{leftOdd}(l)\ \mathbf{of} \\
 & & \quad () \rightarrow \mathbf{if}\ \mathit{odd}(n)\ \mathbf{then}\ \mathsf{L}(n)\ \mathbf{else}\ \mathit{leftOdd}_{\uparrow}(x) \\
 & & \quad v \rightarrow \mathsf{L}(v) \\
\mathit{leftOdd}_{\uparrow}([\mathsf{R}(n,r)] +\!\!+ x) & = & \mathbf{case}\ \mathit{leftOdd}_{\uparrow}(x)\ \mathbf{of} \\
 & & \quad () \rightarrow \mathbf{if}\ \mathit{odd}(n)\ \mathbf{then}\ \mathsf{R}(n) \\
 & & \qquad\quad \mathbf{else\ case}\ \mathit{leftOdd}(r)\ \mathbf{of}\ () \rightarrow () \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad v \rightarrow \mathsf{R}(v) \\
 & & \quad a \rightarrow a
\end{array}$$

In summary, $\mathit{leftOdd}_{\downarrow} = \mathit{leftOdd}_{\uparrow}$ (say *lo*) is both downward and upward, and Theorem 6.19 proves that *leftOdd* is decomposable.

Now let us derive a decomposition $(\phi, \odot, \psi)$ of *leftOdd*. Here, ψ is the one that we have given. Obtaining ϕ is straightforward as the same as the previous cases. From Lemma 6.20, it is sufficient to work out a right inverse of *lo* for deriving $\odot$. It is not difficult, and the following function $\mathit{lo}^{\bullet}$ is a right inverse of *lo*.

$$\begin{array}{lcl}
\mathit{lo}^{\bullet}() & = & [] \\
\mathit{lo}^{\bullet}(\mathsf{L}(v)) & = & [\mathsf{L}(v, \mathit{Leaf})] \\
\mathit{lo}^{\bullet}(\mathsf{R}(v)) & = & [\mathsf{R}(v, \mathit{Leaf})]
\end{array}$$

Then, we can derive a decomposition of *leftOdd* after a small amount of calculation. We would like to omit the calculation, because it is slightly boring though straightforward. The parallel program obtained is the following.

$$
\begin{array}{l}
\mathit{leftOdd} \circ \mathit{z2t} = \psi \circ \mathit{lo} \\
\quad \begin{array}{lcl}
\psi() & = & () \\
\psi(\mathsf{L}(v)) & = & v \\
\psi(\mathsf{R}(v)) & = & v \\
\mathit{lo}([]) & = & () \\
\mathit{lo}([\mathsf{L}(n,t)]) & = & \textbf{case } \mathit{leftOdd}(t) \textbf{ of } () \rightarrow \textbf{if } \mathit{odd}(n) \textbf{ then } \mathsf{L}(n) \textbf{ else } () \\
 & & \qquad\qquad\qquad\qquad\quad v \rightarrow \mathsf{L}(v) \\
\mathit{lo}([\mathsf{R}(n,t)]) & = & \textbf{if } \mathit{odd}(n) \textbf{ then } \mathsf{R}(n) \textbf{ else case } \mathit{leftOdd}(t) \textbf{ of } () \rightarrow () \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad v \rightarrow \mathsf{R}(v) \\
\mathit{lo}(z_1 \mathbin{+\!\!\!+} z_2) & = & \textbf{case } (\mathit{lo}(z_1), \mathit{lo}(z_2)) \textbf{ of } (\mathsf{L}(v), _) \rightarrow \mathsf{L}(v) \\
 & & \qquad\qquad\qquad\qquad\qquad\quad (\mathsf{R}(v), ()) \rightarrow \mathsf{R}(v) \\
 & & \qquad\qquad\qquad\qquad\qquad\quad (_, r) \rightarrow r
\end{array}
\end{array}
$$

The key is distinguishing two kinds of odd numbers: those that are to the left of the terminal leaf and those that are to the right. Writing downward/upward programs is helpful for noticing such case analyses necessary for parallel computations.

Height

As the final example, let us consider computing the height of a tree.

$$
\begin{array}{lcl}
\mathit{height}(\mathit{Leaf}) & = & 1 \\
\mathit{height}(\mathit{Node}(_, l, r)) & = & 1 + (\mathit{height}(l) \uparrow \mathit{height}(r))
\end{array}
$$

This problem is similar to the maximum path weight problem, and we can specify downward and upward definitions in a similar way.

$$
\begin{array}{lcl}
\mathit{height}_{\downarrow}([]) & = & (1,1) \\
\mathit{height}_{\downarrow}(x \mathbin{+\!\!\!+} [\mathsf{L}(_, l)]) & = & \textbf{let } (h,d) = \mathit{height}_{\downarrow}(x) \\
 & & \textbf{in } (h \uparrow (d + \mathit{height}(l)), d+1) \\
\mathit{height}_{\downarrow}(x \mathbin{+\!\!\!+} [\mathsf{R}(_, r)]) & = & \textbf{let } (h,d) = \mathit{height}_{\downarrow}(x) \\
 & & \textbf{in } (h \uparrow (d + \mathit{height}(r)), d+1) \\
\mathit{height}_{\uparrow}([]) & = & (1,1) \\
\mathit{height}_{\uparrow}([\mathsf{L}(_, l)] \mathbin{+\!\!\!+} x) & = & \textbf{let } (h,d) = \mathit{height}_{\uparrow}(x) \\
 & & \textbf{in } (1 + (h \uparrow \mathit{height}(l)), d+1) \\
\mathit{height}_{\uparrow}([\mathsf{R}(_, r)] \mathbin{+\!\!\!+} x) & = & \textbf{let } (h,d) = \mathit{height}_{\uparrow}(x) \\
 & & \textbf{in } (1 + (h \uparrow \mathit{height}(r)), d+1)
\end{array}
$$

The function $\mathit{height}_{\downarrow} = \mathit{height}_{\uparrow}$ (say ht) computes the height of a tree in its first result, and its second result retains the depth of the terminal leaf. The function ht is a path-based computation of *height*, because $\pi_1 \circ \mathit{ht} = \mathit{height} \circ \mathit{z2t}$ holds.

Now we would like to parallelize it. The only nontrivial part is the part to obtain an efficient associative operator from its right inverse. In this case, different from

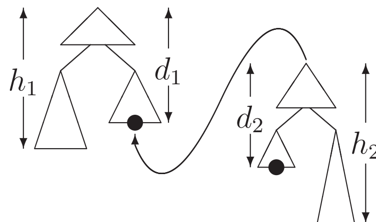

Figure 6.9. An outline of $ht^\bullet\ (h_1, d_1) +\!\!+\ ht^\bullet\ (h_2, d_2)$: the curved arrow denotes the plugging operation, which corresponds to the concatenation of two zippers.

the previous examples, we should define a right inverse $ht^\bullet$ in a recursive manner because $ht^\bullet(h, d)$ should yield a tree of height h. Therefore, it seems difficult to simplify the definition of $\odot$, even though a naive definition of $\odot$, namely $a \odot b = ht(ht^\bullet(a) +\!\!+ ht^\bullet(b))$, is inefficient. In truth, that simplification is not difficult. Look at Figure 6.9, which shows an outline of the tree $ht^\bullet(h_1, d_1) +\!\!+ ht^\bullet(h_2, d_2)$. The left and right trees respectively correspond to $ht^\bullet(h_1, d_1)$ and $ht^\bullet(h_2, d_2)$, and the curved arrow corresponds to the concatenation operation on zippers. Now it is easy to see that the height of this tree is $h_1 \uparrow (d_1 + h_2)$ and the depth of the terminal leaf is $d_1 + d_2$. In short, the following provides a definition of $\odot$.

$$(h_1, d_1) \odot (h_2, d_2) = (h_1 \uparrow (d_1 + h_2), d_1 + d_2)$$

Then, we get the following parallel program for *height*.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{height} \circ \mathit{z2t} = \pi_1 \circ \mathit{ht}} \\
\mathit{ht}([\,]) & = & (1, 1) \\
\mathit{ht}([\mathsf{L}(n, t)]) & = & (1 + \mathit{height}(t), 2) \\
\mathit{ht}([\mathsf{R}(n, t)]) & = & (1 + \mathit{height}(t), 2) \\
\mathit{ht}(z_1 +\!\!+ z_2) & = & \mathbf{let}\ (h_1, d_1) = \mathit{ht}(z_1) \\
 & & \qquad (h_2, d_2) = \mathit{ht}(z_2) \\
 & & \mathbf{in}\ (h_1 \uparrow (d_1 + h_2), d_1 + d_2)
\end{array}$$

We have considered how to merge the results of independent substructures by using an abstract image in Figure 6.9. The most significant thing is that Theorem 6.19 guarantees the correctness of the merging operation obtained from the image. Theorem 6.19 proves that the results of ht, namely the height of the tree and the depth of the terminal leaf, are sufficient for merging the results of two parts; thus, we can derive a correct merging operation no matter what shape of trees we image for $ht^\bullet$.

6.2.5 Experiments

To confirm the scalability of the derived parallel programs, we made some experiments. The environment for the experiments is the same as that in Section 6.1.4.

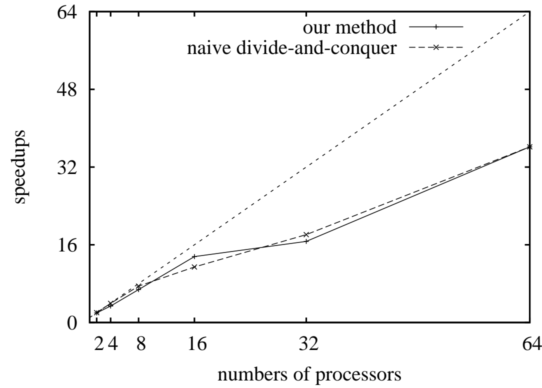

Figure 6.10. Speedup ratios against a sequential implementation for a complete binary tree: the sequential implementation ends up in 0.85 seconds.

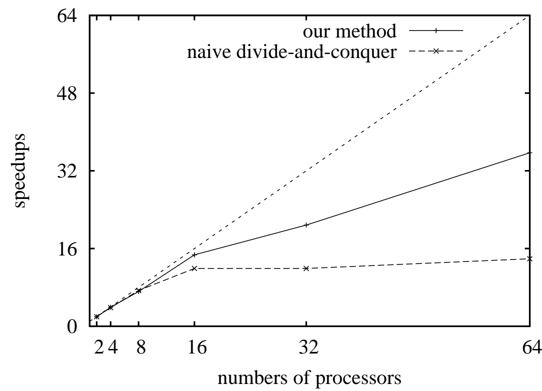

Figure 6.11. Speedup ratios against a sequential implementation for a randomly generated tree: the sequential implementation ends up in 0.98 seconds.

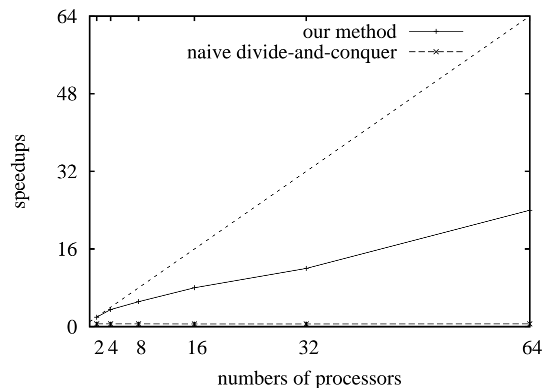

Figure 6.12. Speedup ratios against a sequential implementation for a monadic tree: the sequential implementation ends up in 0.56 seconds.

Given the sequential definition and the derived parallel program of each problem, we prepared two implementations of divide-and-conquer parallel algorithm. One is the naive divide-and-conquer parallel algorithm: using the sequential definition, we compute the value of some independent subtrees in parallel, and after that, compute the rest part sequentially. The other is the SHUNT contraction algorithm with the m-bridge technique, which is implemented in C++ parallel programming library SkeTo [MIEH06]. To reduce the overheads of parallelization, we use the sequential definition as much as possible. We compute the result of subtrees allocated to a processor by using the sequential definition.

We prepared three trees of $2^{26} - 1$ nodes containing integers: a complete binary tree, a randomly generated tree, and a monadic tree. We measured execution times, from which times for initial data distributions are excluded, for the three examples: maximum path weight, leftmost odd number, and height. The results of these three were very similar, and we only report the result of the maximum path weight problem.

Figures 6.10, 6.11, and 6.12 show the results of our experiments, in which we plot speedup ratios against sequential implementations. For a complete binary tree, both implementations show good scalability. It is worth noting that the parallel-tree-contraction-based method is as fast as the naive divide-and-conquer implementation. The m-bridge technique yields the same division as the naive division on complete binary trees, and thus, we can make full use of the sequential definition in the parallel-tree-contraction-based method; in short, both implementation does completely the same computation. For a randomly generated tree and a monadic tree, the naive divide-and-conquer implementation shows poor scalability. It is because computation of a relatively large subtree forms a bottleneck. The parallel-tree-contraction-based method shows good scalability even for these trees. However, scalability is relatively bad for the monadic tree. In this case, few computations are performed by using sequential definition because few subtrees are allocated to a processor. Then, the overheads of parallelization come to the surface and affect efficiency.

6.3 Parallel Programming on Non-Binary Trees

So far, we have considered binary trees, on which parallel tree contraction are useful for developing efficient parallel algorithms. However, contrary to rich studies on binary trees, few studies consider non-binary trees. Some studies just mentioned that non-binary trees can be encoded as binary trees. However, such encoding is often troublesome because it breaks the original structure. For example, consider that we want to compute the height of a non-binary tree, and we encode the tree as a binary tree. Then, the height of the binary tree is not that of the original tree anymore. Moreover, we have several kinds of binary-tree encoding for a single tree, and we need to select an appropriate binary-tree encoding to develop an efficient parallel

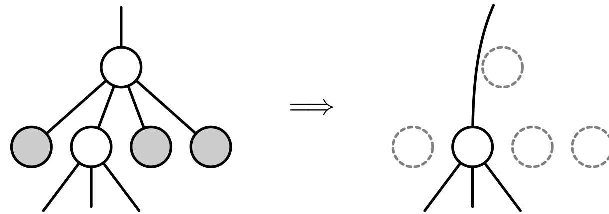

Figure 6.13. An M-SHUNT operation (marked nodes are colored)

algorithm on it. In short, the binary-tree encoding makes problems complicated.

In this section, we extend results on binary trees shown in the previous section. We consider trees of bounded degrees, propose a new parallel tree contraction algorithm for them, and prove "the third list-homomorphism theorem" on them.

6.3.1 Parallel Tree Contraction Algorithm for Non-Binary Trees

Here, we propose a new parallel tree contraction algorithm that works well even for non-binary trees. Our algorithm is a generalization of the SHUNT contraction algorithm [ADKP89]. An important feature of our algorithm is that it requires no binary-tree encoding. Therefore, it is easy to develop parallel algorithms based on our parallel tree contraction algorithm. Our algorithm is cost optimal if the maximum degree of the underlying tree is O(1). In addition, we show sufficient conditions when computations can be parallelized based on our algorithm. Our sufficient conditions are generalizations of those known on binary trees [GR89, ADKP89].

We assume that a tree has no unary node; otherwise, we remove such a node by inserting a dummy leaf.

In the case of binary trees, a leaf has a unique sibling, and thus, we can define the SHUNT operation to be applied to a leaf. In the case of non-binary trees, however, a single leaf does not necessarily specify its sibling nor a SHUNT operation. We introduce *marks* on leaves to specify a SHUNT operation.

Definition 6.21 (SHUNT operation with marks)**.** A SHUNT operation with marks (we will call it *M-SHUNT*) is an operation applied to an internal node whose children are one unmarked node and the other marked leaves. An M-SHUNT operation removes the internal node and all its marked children and connects the unmarked child to the parent of the internal node. □

Figure 6.13 shows the behavior of an M-SHUNT operation. Note that an M-SHUNT operation to a binary internal node is isomorphic to the usual SHUNT operation. We consider that each removal of a node takes O(1) time; thus, an M-SHUNT operation for a k-ary node takes O(k) time.

The following procedure is our parallel tree contraction algorithm for non-binary trees.

Procedure 6.22 (SHUNT contraction for non-binary trees)**.**

(1) Number all leaves from left to right.
(2) Mark all odd-numbered leaves that have an unmarked right sibling, and apply M-SHUNT operations to all the possible nodes.
(3) Mark all odd-numbered unmarked leaves, and apply M-SHUNT operations to all the possible nodes.
(4) Erase the numbers of the marked leaves, and halve those of the unmarked leaves.
(5) Go to (2) until the tree consists of only one node. □

It is worth noting that Procedure 6.22 is equivalent to Procedure 6.5 when the input is a binary tree. In this sense, we can state that Procedure 6.22 is a generalization of Procedure 6.5. Actually, Procedure 6.22 inherits good characteristics from Procedure 6.5.

Lemma 6.23. Procedure 6.22 raises no conflicting applications of M-SHUNT operations.

Proof. Note that unmarked leaves are numbered from left to right throughout the procedure.

Let v_1 be the parent of an internal node v_2.

First we prove by contradiction that simultaneous M-SHUNT operations to v_1 and v_2 never occur in the step (2). Let l_1 be the rightmost newly marked leaf of v_1 and l_2 be the leftmost newly marked leaf of v_2. By the assumption that M-SHUNT operations are applicable to both v_1 and v_2, v_1 has exactly one unmarked child that is v_2, and v_2 has the only unmarked child on the right of l_2. Because l_1 is newly marked and v_2 is the only unmarked child of v_1, v_2 is a right sibling of l_1, and thus the number of l_1 is less than that of l_2. Since even-numbered leaves remain unmarked, an unmarked even-numbered leaf (say l_3) should exist between l_1 and l_2. Here, l_3 is not a child of v_1 because v_1 has the only unmarked child v_2; l_3 is not a descendant of v_2 because v_2 should have the unmarked child on the right of l_2 but not on the left of l_2. Therefore, such a leaf l_3 must not exist and a contradiction occurs.

The case for the step (3) is similar. Let l_1 be the leftmost newly marked leaf of v_1, l_2 be the rightmost newly marked leaf of v_2, and l_3 be an even-numbered leaf between l_1 and l_2. Notice that since v_2 is an unmarked child of v_1 and l_1 is not marked in the previous step, l_1 is a right sibling of v_2 and the number of l_1 is greater than that of l_2. Obviously l_3 is not a child of v_1. If l_3 is a child of v_2, then l_3 is an unmarked right sibling of l_2 due to the order of l_1 and l_2, but in such a case l_2 should be marked in the previous step (2), which is a contradiction. □

Theorem 6.24. Procedure 6.22 runs in $O(kn/p+k\log p)$ time on an EREW PRAM with p processors, where n is the size of the tree and k is the maximum degree of the tree.

Proof. Correctness on the EREW PRAM follows from Lemma 6.23.

The step (1) can be implemented by the Euler-tour technique with the list ranking procedure and is done in $O(kn/p + k \log p)$ time. Since an M-SHUNT operation costs $O(k)$ time, the cost of the theorem is achieved if the steps (2)–(5) take $O(n/p + \log p)$ steps. Since the number of unmarked leaves decreases into the half through a sequence of the steps (2)–(5), the cost is achieved as the case of Procedure 6.5. □

Note that the complexity is cost optimal when p is $O(n/\log n)$ and k is $O(1)$.

An important fact is that Procedure 6.22 requires no binary-tree encoding; thus, it is helpful for developing parallel algorithms. The following theorem shows a sufficient condition to achieve parallel computations following the Procedure 6.22, which is a generalization of Theorem 6.7.

Theorem 6.25. Assume that there are a set of values S and two sets of indexed functions F and G such that the following conditions hold.

- Any element of F and G can be evaluated in $O(1)$ time.
- For all $f_i \in F$ and $a_1, a_2, \ldots, a_{l-1}, a_{l+1}, \ldots, a_k \in S$, where k is the arity of f_i, there exists a function $g_j \in G$ such that $f_i(a_1, \ldots, a_{l-1}, x, a_{l+1}, \ldots, a_k) = g_j(x)$ holds and the index j can be computed from $a_1, \ldots, a_{l-1}, a_{l+1}, \ldots, a_k$, l, and i in $O(k)$ time.
- For all $g_i, g_j \in G$, there exists a function $g_m \in G$ such that $g_i(g_j(x)) = g_m(x)$ holds and the index m can be computed in $O(1)$ time from i and j.

Then, an algebraic computation defined by (S, F) can be computed in $O(n/p + \log p)$ time on an EREW PRAM with p processors, where n is the size of the expression.

Proof. We can assume that G contains the identity function without loss of generality. As a preprocess, associate the index of the identity function to each internal node. After that, run Procedure 6.22 with performing computation described bellow when an M-SHUNT operation is applied to a node. Let $f_j \in F$ and p respectively be the operator and the index stored at the node. If all children of the node are leaves whose values are $a_1, \ldots, a_k$, store $g_p(f_j(a_1, \ldots, a_k))$ to the leaf left after the M-SHUNT operation. If l-th child of the node is an internal node that stores an index q and values of other children are $a_1, \ldots, a_{l-1}, a_{l+1}, \ldots, a_k$, update the index q by r such that $g_r(x) = g_p(f_j(a_1, \ldots, a_{l-1}, g_q(x), a_{l+1}, \ldots, a_k))$ holds.

It is not difficult to see that the procedure above yields the result of the algebraic computation; besides, because arity of each function in F is at most constant, it runs in $O(n/p + \log p)$ time. □

A direct consequence of Theorem 6.25 is a parallel algorithm to evaluate algebraic computations whose carrier is finite, which is a generalization of the known result where operators are binary [GR89].

Corollary 6.26. If the size of the set S is O(1) and each element of F can be evaluated in O(1) time, any algebraic computation defined by (S, F) can be computed in $O(n/p + \log p)$ time on an EREW PRAM with p processors, where n is the size of the expression.

Proof. Let each functions in G required in Theorem 6.25 be a transition table from S to S. Since the size of S is O(1), the size of a table is O(1) and a composition of two tables can be evaluated in O(1) time. □

Corollary 6.26 yields, for example, an efficient parallel evaluation algorithm for arithmetic expressions consisting of $+$, $-$, $\times$, $/$, and conditional operators, on a Galois field.

It is worth noting that the set of functions G in Theorem 6.25 corresponds to the set of continuations raised from one-hole contexts of trees, as the same as the case of Theorem 6.7. In other words, computations on one-hole contexts are key to parallel computations even on non-binary trees. An important consequence of this observation is that the m-bridge technique is applicable for trees of bounded degrees. As the same as the case of binary trees, Lemma 6.10 indicates that each bridge corresponds to a one-hole context; moreover, from Lemmas 6.11 and 6.12, the m-bridge technique yields good load-balancing when the maximum degree is O(1). Therefore, the premise of Theorem 6.25 implies efficient divide-and-conquer parallel algorithms on trees of bounded degrees.

6.3.2 The Third List-Homomorphism Theorem on Polynomial Data Structures

Next, we generalize the programming part. We would like to generalize the notions of path-based computations, upward computations, downward computations, and decompositions. Based on them, we prove the third list-homomorphism theorem on polynomial data structures. Since polynomial data structures can be seen as trees of bounded degrees, the tree contraction algorithm in the previous subsection leads to efficient parallel computations.

To introduce zippers for polynomial data structures, we follow McBride [McB01] who showed a systematic derivation of zippers based on derivatives on functors. The derivative of a polynomial functor F, denoted by $\partial\mathsf{F}$, is a polynomial functor defined as follows, where $\bullet$ denotes a distinguishable element.

$$\begin{aligned} \partial(!A) &= !\emptyset \\ \partial(\mathsf{I}) &= !\{\bullet\} \\ \partial(\mathsf{F} \times \mathsf{G}) &= \mathsf{F} \times \partial\mathsf{G} + \partial\mathsf{F} \times \mathsf{G} \\ \partial(\mathsf{F} + \mathsf{G}) &= \partial\mathsf{F} + \partial\mathsf{G} \end{aligned}$$

The functor $\partial\mathsf{F}$ corresponds to a one-hole context structure of F. The zipper structure of $\mu\mathsf{F}$, denoted by Z_F, is recognized as $Z_\mathsf{F} = (\partial\mathsf{F}(\mu\mathsf{F}))^*$.

To convert zippers to trees, we use an operator $(\triangleleft_{\mathsf{F}}) : (Z_{\mathsf{F}} \times \mu\mathsf{F}) \to \mu\mathsf{F}$, which is defined by

$$z \triangleleft_{\mathsf{F}} t = \mathsf{foldr}_{\lessdot_{\mathsf{F}},t}(z)$$

where $(\lessdot_{\mathsf{F}}) : (\partial\mathsf{F}(\mu\mathsf{F}) \times \mu\mathsf{F}) \to \mu\mathsf{F}$ is the plugging-in operator [McB01] defined as follows.

$$\begin{array}{lcl}
a \lessdot_{!A} t &=& a \\
\bullet \lessdot_{\mathsf{I}} t &=& t \\
\mathsf{L}(a,b) \lessdot_{\mathsf{F}\times\mathsf{G}} t &=& (a, b \lessdot_{\mathsf{G}} t) \\
\mathsf{R}(a,b) \lessdot_{\mathsf{F}\times\mathsf{G}} t &=& (a \lessdot_{\mathsf{F}} t, b) \\
\mathsf{L}(a) \lessdot_{\mathsf{F}+\mathsf{G}} t &=& \mathsf{L}(a \lessdot_{\mathsf{F}} t) \\
\mathsf{R}(b) \lessdot_{\mathsf{F}+\mathsf{G}} t &=& \mathsf{R}(b \lessdot_{\mathsf{G}} t)
\end{array}$$

We will omit the subscript of $\triangleleft_{\mathsf{F}}$ when it is apparent form its context.

Notice that the operator $\triangleleft$ takes an additional tree to convert a zipper to a tree, because a zipper corresponds to a one-hole context. We would like to minimize the differences between zippers and trees, and thus, we force the difference between zippers and trees to be leaves. A set of leaves of $\mu\mathsf{F}$, denoted by $\mathit{leaves}_{\mu\mathsf{F}}$, is formalized as follows.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{leaves}_{\mu\mathsf{F}} = [\![\mathsf{F}]\!]_{\mathit{leaves}}} \\
[\![!A]\!]_{\mathit{leaves}} &=& A \\
[\![\mathsf{I}]\!]_{\mathit{leaves}} &=& \emptyset \\
[\![\mathsf{F} \times \mathsf{G}]\!]_{\mathit{leaves}} &=& [\![\mathsf{F}]\!]_{\mathit{leaves}} \times [\![\mathsf{G}]\!]_{\mathit{leaves}} \\
[\![\mathsf{F} + \mathsf{G}]\!]_{\mathit{leaves}} &=& [\![\mathsf{F}]\!]_{\mathit{leaves}} + [\![\mathsf{G}]\!]_{\mathit{leaves}}
\end{array}$$

Downward and upward computations are formalized as follows.

Definition 6.27 (path-based computation)**.** A function $h' : Z_{\mathsf{F}} \to B$ is said to be a *path-based computation* of a function $h : \mu\mathsf{F} \to A$ if there exists an operator $\ominus : (B \times \mu\mathsf{F}) \to A$ such that the following equation holds for all $t \in \mathit{leaves}_{\mu\mathsf{F}}$.

$$h(z \triangleleft t) \;=\; h'(z) \ominus t \qquad \square$$

Definition 6.28 (downward computation)**.** A function $h' : Z_{\mathsf{F}} \to B$, which is a path-based computation of a function $h : \mu\mathsf{F} \to A$, is said to be *downward* if there exist an operator $(\otimes) : (B \times \partial\mathsf{F}A) \to B$ and a value $e \in B$ such that the following equation holds.

$$h' \;=\; \mathsf{foldl}_{\otimes,e} \circ \mathsf{map}_{\partial\mathsf{F}h} \qquad \square$$

Definition 6.29 (upward computation)**.** A function $h' : Z_{\mathsf{F}} \to B$, which is a path-based computation of a function $h : \mu\mathsf{F} \to A$, is said to be *upward* if there exist an operator $(\oplus) : (\partial\mathsf{F}A \times B) \to B$ and a value $e \in B$ such that the following equation holds.

$$h' \;=\; \mathsf{foldr}_{\oplus,e} \circ \mathsf{map}_{\partial\mathsf{F}h} \qquad \square$$

As the same as the case of node-valued binary trees, we consider recursive division on one-hole contexts to characterize scalable divide-and-conquer parallel programs.

Definition 6.30 (decomposition)**.** A *decomposition* of a function $h : \mu\mathsf{F} \to A$ is a tuple $(\phi, \odot, \ominus)$ that consists of a function $\phi : \partial\mathsf{F}A \to B$, an associative operator $\odot : (B \times B) \to B$, and an operator $\ominus : (B \times \mu\mathsf{F}) \to A$ such that the following equation holds for all $t \in \mathit{leaves}_{\mu\mathsf{F}}$.

$$h(z \lhd t) = \mathsf{hom}_{\odot, \phi \circ \partial\mathsf{F}h}(z) \ominus t$$ □

Theorem 6.31. If $(\phi, \odot, \ominus)$ is a decomposition of a function h and all of ϕ, $\odot$, and $\ominus$ are constant-time computations, then h can be evaluated for a tree of n nodes in $\mathrm{O}(n/p + \log p)$ time on an EREW PRAM with p processors.

Proof. It is not difficult to confirm that h satisfies the premise of Theorem 6.25. □

It is worth remarking that since decomposition of a function implies the premise of Theorem 6.25, decomposable function can be evaluated in a divide-and-conquer manner based on the m-bridge technique.

Since a decomposition is characterize by a list homomorphism, we can utilize parallelization methods on lists for free. For instance, "the third list-homomorphism theorem" on trees is a direct consequence of that on lists.

Lemma 6.32. Assume that a function h', which is a path-based computation of h satisfying $h(z \lhd t) = h'(z) \ominus t$ holds for all $t \in \mathit{leaves}_{\mu\mathsf{F}}$, is both downward and upward; then, there exists a decomposition $(\phi, \odot, \ominus)$ of h such that $\phi(\partial\mathsf{F}h(a)) = h'([a])$ and $a \odot b = h'(h'^{\circ}(a) \mathbin{+\!\!+} h'^{\circ}(b))$ hold. □

Proof. Since h' is both leftward and rightward, $\odot$ is associative and $h' = \mathsf{hom}_{\odot, \phi \circ \partial\mathsf{F}h}$ holds from Lemma 6.3. Therefore, $(\phi, \odot, \ominus)$ forms a decomposition of h. □

Theorem 6.33 (the third list-homomorphism theorem on polynomial data structures)**.** A function $h : \mu\mathsf{F} \to A$ is decomposable if and only if there exist three operators $(\oplus) : (\partial\mathsf{F}A \times B) \to B$, $(\otimes) : (B \times \partial\mathsf{F}A) \to B$, and $(\ominus) : (B \times \mu\mathsf{F}) \to A$, such that the following equations hold for all $t \in \mathit{leaves}_{\mu\mathsf{F}}$.

$$\begin{aligned} h(z \lhd t) &= \mathsf{foldr}_{\oplus, e}(\mathsf{map}_{\partial\mathsf{F}h}(z)) \ominus t \\ &= \mathsf{foldl}_{\otimes, e}(\mathsf{map}_{\partial\mathsf{F}h}(z)) \ominus t \end{aligned}$$

Proof. The "if" part is a direct consequence of Lemma 6.32, and the "only if" part is straightforward. □

As the final remark, we show that any bottom-up computations are decomposable, if we neglect efficiency.

Lemma 6.34. Assume that a function $h : \mu\mathsf{F} \to A$ satisfies the following properties.

$$h(t) = h(t') \Rightarrow h(z \lhd t) = h(z \lhd t')$$
$$\mathsf{map}_{\partial\mathsf{F}h}(x) = \mathsf{map}_{\partial\mathsf{F}h}(y) \Rightarrow h(x \lhd t) = h(y \lhd t)$$

Then, h is decomposable.

Proof. Let $\odot$ be $+\!\!+$, and define ϕ and $\ominus$ by $\phi = \partial \mathsf{F} h^{\bullet}$ and $z \ominus t = h(z \triangleleft t)$. Then, $\odot$ is associative; besides, $\mathsf{map}_{\partial \mathsf{F} h} \circ \mathsf{hom}_{\odot, \phi \circ \partial \mathsf{F} h} = \mathsf{map}_{\partial \mathsf{F} h \circ \phi \circ \partial \mathsf{F} h} = \mathsf{map}_{\partial \mathsf{F} h}$ holds. Therefore, $h(z \triangleleft t) = \mathsf{hom}_{\odot, \phi \circ \partial \mathsf{F} h}(z) \ominus t$ holds. □

The result shown in Lemma 6.34 is useless in practice, because derived program does almost nothing on zippers. Thus it is necessary to specify the computation on zippers, or at least, the type of result of computation on zippers. The third list-homomorphism theorem enables us to obtain parallel computations having specified types, and thus, we can exclude such useless cases.

6.4 Summary and Discussions

In this section, we have developed a framework for systematic construction of cost optimal divide-and-conquer parallel programs on polynomial data structures. We have introduced parallel tree contraction algorithms that are useful for developing cost-optimal parallel algorithms, shown sufficient conditions to apply the algorithms, and proved "the third list-homomorphism theorems" that enable us to develop programs satisfying the conditions in a systematic manner. Our theorem is exactly a generalization of the original theorem on lists, and our results build on list homomorphisms. Therefore, existing automatic parallelization methods will be applicable.

We have focused on the third list-homomorphism theorem [Gib96]. While the third list-homomorphism theorem is a folk theorem in the calculational programming community, its effectiveness has not been discussed well. We have shown derivations of parallel programs based on right inverses, and confirmed that the third list-homomorphism theorem is certainly useful for developing parallel programs. While the use of right inverses is not our new invention, because Gibbons [Gib96] also showed a right-inverse-based derivation, intensive study about the use of right inverses in calculations is our contribution.

The third list-homomorphism theorem requires two sequential programs, while usual parallelization methods generate a parallel program from a sequential program. Even though this requirement seems a shortcoming, it is a strong point in truth. In general, there is little hope that we can obtain a parallel program from a sequential program, because parallel programs will be more complicated than sequential programs. In other words, extra information is necessary to develop parallel programs from sequential ones. What the third list-homomorphism theorems provide is a systematic way to reveal such extra information.

After confirmed usefulness of the third list-homomorphism theorem, we have tried to generalize the results on lists to trees. Such generalization has been discussed in the framework of *skeletal parallel programming*, in which parallel programs are developed by combining ready-made parallel computation patterns called skeletons. Skillicorn and Gibbons [Ski96, GCS94] introduced parallel tree skeletons and proposed their implementation based on parallel tree contraction. Matsuzaki et al. [MHT03, MHT06, Mat07b] did intensive studies on this topic, and im-

plemented parallel tree skeletons in skeleton-based parallel programming library SkeTo [MIEH06]. Although parallel skeletons are useful for parallel programming, their drawback is the requirement for operators that we plug skeletons with. Because parallel tree skeletons are implemented based on tree-contraction algorithms, operators used by skeletons should satisfy certain conditions, such as the premise of Theorem 6.7, and it is a duty of users to guarantee the conditions. Our "the third list-homomorphism theorem" on trees enables us to develop operators satisfying the conditions in a systematic way. It is worth remarking that the conditions required for parallel tree skeletons [MHT03, Mat07b] are equivalent to the condition required for decompositions, while formalizations differ a bit.

On developing "the third list-homomorphism theorem" on trees, we focused on paths so as to utilize the known theories on lists. Then, we noticed the relationship between paths and one-hole contexts, and found that Huet's zippers [Hue97] led to understandable programs that dealt with one-hole contexts. The notion of one-hole contexts establishes a link between parallel tree contraction and the third list-homomorphism theorem, because it is also the key to parallel tree contraction algorithms.

As seen, our parallel programming framework relies on parallel tree contraction algorithms. Parallel tree contraction, which was first proposed by Miller and Reif [MR85], is known to be a useful framework to develop cost-optimal parallel programs on trees, and many computations have been implemented on it [CV88, GR89, ADKP89, Ski96, GCS94, MHT03, Mat07b]. While there are several efficient parallel tree contraction algorithms on binary trees [CV88, GR89, ADKP89, MW97], few studies consider parallel tree contraction algorithms on non-binary trees without binary-tree encoding. While the parallel tree contraction algorithm by Miller and Reif [MR85] works for non-binary trees, it requires concurrent-read/concurrent-write PRAM. In [Rei93], a cost-optimal tree contraction algorithm on EREW PRAM is shown, which is an extension of that by Miller and Reif. Our new tree contraction algorithm is simpler than the algorithm and suitable for practical use.

Because few practical tree contraction algorithms are known for non-binary trees, existing parallel tree skeletons only deal with binary trees. Matsuzaki et al. [MHKT05, KME07, Mat07b] proposed parallel tree skeletons on non-binary trees, which are implemented based on a binary-tree encoding. They required a condition on operators for skeletons, called extended distributivity, as a requirement of a successful parallel implementation on the binary-tree encoding. The premise of Theorem 6.25 is simpler and more understandable than theirs.

We have shown that our results can be scaled up to polynomial data structures, which correspond to trees of bounded degrees. *Regular data structures* are a generalization of polynomial data structures, and include trees of unbounded degrees. Since McBride [McB01] showed a systematic derivation of zipper structures for regular data structures, it is not a problem to formalize path-based computations, downward computations, and upward computations on regular data structures based on their zipper structures. However, it is difficult to construct cost-optimal parallel tree

contraction algorithm without binary-tree encoding on regular data structures. In this chapter, we have required that the operation to merge results of siblings can be done in O(1) time, and this requirement is not realistic on regular data structures. Therefore, it is necessary to parallelize this merging operation. In summary, it is an interesting future work to give "the third list-homomorphism theorem" on regular data structures.

We have considered list homomorphisms not only on lists but also trees, and confirmed that list homomorphisms provide a good abstraction for efficient parallel programs. However, when we consider parallelization of a large program, we would like to parallelize the whole of the program, because a small sequential part in the program will be a bottleneck when a large number of processors are available. Therefore, expressiveness is a very important issue for parallel programming frameworks. From this viewpoint, effectiveness of list homomorphisms for large and practical examples is still disputable.

Chapter 7

Automatic Parallelization on the Third List-Homomorphism Theorem

In Chapter 6, we have developed theories for parallel programming. Associativity enables us to divide a structure at its middle, and the third list-homomorphism theorem helps to obtain associative operators. We have shown some calculations to derive parallel programs; however, our derivations have been handiwork so far.

The goal of this chapter is to establish automatic parallelization methods based on the third list-homomorphism theorem. We design a programming language to describe sequential programs that are objects of parallelization, and introduce two automatic parallelization methods. One [MMM+07, Mor07] is based on program inversion. Following Lemma 6.3, we derive an associative operator by generating a right inverse of a function. The other [MMHT08b] is based on generate-and-testing method. We generate candidates of parallel programs, and find a program that is certainly a list homomorphism. In both methods, automatic theorem proving techniques play an important role. We use *quantifier elimination* [CJ98] techniques to achieve automatic theorem proving.

7.1 Brief Introduction of Quantifier Elimination

In this chapter, we will make use of quantifier elimination [CJ98] techniques. Here we briefly introduce them.

Given a quantified formula, for example $\forall x : ax^2 + bx + c > 0$, quantifier elimination techniques calculate an equivalent formula containing no quantifier, such as $(a = b = 0 \wedge c > 0) \vee (a > 0 \wedge b^2 - 4ac < 0)$. Quantifier elimination techniques are useful for automatic theorem proving, because a quantifier-free formula is easy to give a proof or disproof.

This chapter corresponds to combination of [MMM+07] and [MMHT08b].

Here, we only consider elementary theories of real closed fields, that is, each predicate is an equation/inequality of polynomial expressions, and each variable ranges over $\mathbb{R}$. Tarski [Tar51] proved that elementary theories of real closed fields is decidable by providing a quantifier elimination procedure. Collins [Col75] proposed a more efficient procedure, called cylindrical algebraic decomposition, which is actually available in some of existing computer algebra systems.

While cylindrical algebraic decomposition is effective, it is a complicated procedure and very costly. We will consider a simpler case, in which each predicate is inequality of linear expressions. For such cases, efficient procedures are known. We introduce two procedures: Fourier-Motzkin elimination [DE73] and the method by Loos and Weispfenning [LW93].

Fourier-Motzkin elimination procedure is the following. We consider elimination of a universal quantifier, and the case of an existential quantifier is its dual. First, translate the formula into its conjunctive normal form; then, because of the distributivity of universal quantifiers over conjunctions, namely $\forall x : \bigwedge_i e_i \Leftrightarrow \bigwedge_i (\forall x : e_i)$, it is sufficient to consider elimination of a universal quantifier for disjunctions of inequalities. We can eliminate a quantifier as follows, where a_p, a_q, a_r, and a_s are positive numbers.

$$\begin{aligned}
& \forall x : (\bigvee_p a_p x + b_p \leq 0) \vee (\bigvee_q a_q x + b_q < 0) \vee (\bigvee_r a_r x + b_r \geq 0) \vee (\bigvee_s a_s x + b_s > 0) \\
= \; & (\bigvee_{p,r} a_r b_p \leq a_p b_r) \vee (\bigvee_{q,r} a_r b_q \leq a_q b_r) \vee (\bigvee_{p,s} a_s b_p \leq a_p b_s) \vee (\bigvee_{q,s} a_s b_q < a_q b_s)
\end{aligned}$$

Fourier-Motzkin elimination is practically efficient if transformations into the normal forms are not costly. However, in worst cases, it is terribly inefficient. For example, if we need to eliminate alternating universal and existential quantifiers, the size of the formula increases exponentially in every elimination step. Similarly, it is impractical to extend the procedure so that it can deal with nonlinear expressions, because a few case analyses cause an exponential increase of the size of the formula.

Another efficient procedure for linear inequalities is the method proposed by Loos and Weispfenning [Wei88, LW93]. The method is based on the idea to enumerate all test cases such that the formula is true if and if and only if all test cases yields true. The key idea of the method is summarized as the following lemma.

Theorem 7.1 ([Wei88, LW93])**.** Let φ be a quantifier-free linear formula containing no negation, let x be a linear variable in φ, and define sets I and J by $I = \{(a,b) \mid (ax = b) \in \varphi \vee (ax \leq b) \in \varphi\}$ and $J = \{(a,b) \mid (ax < b) \in \varphi \vee (ax \neq b) \in \varphi\}$. Then, $\forall x : \varphi$ is equivalent to $\bigwedge_{s \in S} \varphi[s/x]$, where the set S is defined as follows.

$$\begin{aligned}
S \;=\; & \{\frac{b}{a} \mid (a,b) \in J\} \cup \{\frac{b}{a} \pm 1 \mid (a,b) \in I\} \\
& \cup \{\frac{1}{2}(\frac{b_i}{a_i} + \frac{b_j}{a_j}) \mid (a_i,b_i),(a_j,b_j) \in I \wedge i \neq j\}
\end{aligned}$$

□

$$
\begin{array}{rcll}
prog & ::= & decl \cdots decl & \{ \text{ program } \} \\
decl & ::= & f([a]) = e;\ f([a] +\!\!+ x) = e;\ f(x +\!\!+ [a]) = e; & \{ \text{ function definition } \} \\
e & ::= & c \mid a \mid f(x) \mid e \oslash e \mid \textsf{if}\ p\ \textsf{then}\ e\ \textsf{else}\ e & \{ \text{ expression } \} \\
\oslash & ::= & + \mid - \mid \times \mid \uparrow \mid \downarrow & \{ \text{ arithmetic operator } \} \\
p & ::= & p \wedge p \mid p \vee p \mid \neg p \mid e < e \mid e = e \mid e \leq e \mid e \neq e & \{ \text{ predicate } \}
\end{array}
$$

Figure 7.1. The syntax of the language used for our automatic parallelization, where c denotes a constant real value and a denotes a real-valued variable.

The set S above corresponds to the test cases. Notice that a linear formula represents intervals, and a universal quantifier requires that the union of intervals covers the entire of $\mathbb{R}$. The test cases are generated so that each possibly non-covered interval contains at least one test case. Loos and Weispfenning [LW93] showed improvements that yield smaller set of test cases, by using case analyses and additional symbols, namely infinity and infinitesimal.

One good thing of Theorem 7.1 is that it does not requires any formula of the specific form. Therefore, this method can be used to deal with nonlinear cases, if there are no squares of the same variable. For example, when we will eliminate $\forall x$, we regard all variables except x as a constant value, and apply the procedure shown in Theorem 7.1. The procedure usually requires some information about coefficients, e.g., b/a is nonsense if a is 0; instead of the information, we add case analyses expressed by formulae. Such case analyses are not too costly, which is a good point of Theorem 7.1. Note that the procedure cannot deal with all of nonlinear formulae even if the formulae have no square of the same variable, because square of the same variable may occur as a result of an elimination step.

7.2 Designing a Language for Automatic Parallelization

Well, let us design a language to describe the inputs of automatic parallelization methods. Here, we only consider parallelization of functions that compute values from lists.

As mentioned, the important requirements for the language are effectiveness and expressiveness. For effective parallelization, we would like to make use of the third list-homomorphism theorem and automatic theorem proving techniques. To utilize the third list-homomorphism theorem, we require users to write both of leftward and rightward programs. There are many studies about automatic theorem proving, and we adopt quantifier elimination techniques. Quantifier elimination techniques are good at dealing with numerical computations, which appear at many practical applications of parallel programming, such as data mining and simulation. Therefore,

use of quantifier elimination also improves expressiveness. In addition, quantifier elimination is based on usual inequalities, which is intuitive even for nonspecialists and useful for encoding practical computations.

In summary, Figure 7.1 shows the syntax of our language for describing the input of automatic parallelization systems. The program is a set of recursive functions, each of which consists of leftward and rightward definitions. For defining the function, we can use constant values, variables, function calls, conditional expressions, and arithmetic operations including addition, subtraction, multiplication, minimum, and maximum. The conditions are equalities or inequalities between expressions. Use of divisions is prohibited for avoiding worrying about treatment of division by zero. In theory, there is no problem to add divisions to our language.

7.3 Parallelization via Inversion

Morita et al. [MMM+07, Mor07] proposed a parallelization method based on automatic inversion of programs. The method is based on Lemma 6.3. We first derive a right inverse of the function that is the object of parallelization, and obtain an associative operator from the right inverse. The difficulty is how to obtain a right inverse. Obtaining a right inverse (we will call it *right inversion*) is nontrivial even on our restricted language in Figure 7.1. Moreover, there are few studies for automatic right inversion, while there are many studies for automatic inversion, and we need to construct a new method.

7.3.1 Automatic Derivation of Right Inverses

Before introducing the automatic right inversion method by Morita et al., let us specify problems for providing a right inversion method. In our language, conditional expressions and recursions are the main difficulty for right inversion. To obtain an inverse of a conditional expression, we need to provide an appropriate way to decide which branch should be chosen in an inverse computation. In usual, such decisions require a lot of information of the program. However, it is difficult to extract accurate information from recursive functions.

The key idea of the method is to give up to deal with these difficulties. First of all, we only consider right inverse functions that are not recursive. In other words, we only consider right inverse functions that result in lists of length at most constant. This assumption is also effective to obtain efficient parallel programs, because, as seen in Lemma 6.3, the derived parallel program should do computations on the list generated from the right inverse. However, because of this assumption, some functions are excluded from the domain of the method. For example, it cannot derive a parallel program of the function *length* that returns the length of the list inputted. As similar to recursions, we do not try to deal with conditionals. Instead of choosing appropriate branches for conditionals, we just derive inverses of all branches, and

after that, we remove unnecessary branches.

Now let us introduce the automatic right inversion method. The method consists of three steps: inversion, verification, and improvements. We explain these three steps with an example, the function *mis* that computes maximum initial-segment sum. As seen in Section 6.1.3, the function *mis* can be parallelized by tupling it with the summation function *sum*, and thus, our objective is to derive a right inverse of the following function *ms*.

$$\begin{array}{lcl} ms([\,]) & = & 0 \\ ms([a] +\!\!\!+\, x) & = & \textbf{let}\ (i,s) = ms(x)\ \textbf{in}\ (0 \uparrow (a+i),\ a+s) \end{array}$$

The first and second components of *ms* respectively retain the maximum initial-segment sum and the summation of the list.

Inversion

We first assume that the right inverse of *ms*, denoted by $ms^\bullet$, results in a list containing two elements. There is no serious reason for this assumption—we can try to obtain another right inverse, which may results in a list of three elements. Note that if this assumption is not appropriate, the derived right inverse is incorrect. Later we will confirm its correctness in the verification step.

From the definition of right inverses and the assumption, we have the following formula.

$$ms(x) = (i,s) \Rightarrow (ms^\bullet(i,s) = [a,b] \Rightarrow ms([a,b]) = (i,s))$$

We calculate as follows.

$$\begin{array}{ll}
\multicolumn{2}{l}{ms([a,b]) = (i,s)} \\
\Leftrightarrow & \{ \text{ definition of } ms \ \} \\
& (0 \uparrow a \uparrow (a+b), a+b) = (i,s) \\
\Leftrightarrow & \{ \text{ equality on tuples } \} \\
& 0 \uparrow a \uparrow (a+b) = i \wedge a+b = s \\
\Leftrightarrow & \{ \text{ hoisting conditionals } \} \\
& (0 \geq a \wedge 0 \geq a+b \wedge 0 = i \wedge a+b = s) \vee \\
& (a \geq 0 \wedge a \geq a+b \wedge a = i \wedge a+b = s) \vee \\
& (a+b \geq 0 \wedge a+b \geq a \wedge a+b = i \wedge a+b = s) \\
\Leftrightarrow & \{ \text{ solving the equations } \} \\
& (0 \geq a \wedge 0 \geq s \wedge i = 0 \wedge a+b = s) \vee \\
& (i \geq 0 \wedge i \geq s \wedge a = i \wedge b = s-i) \vee \\
& (s \geq 0 \wedge b \geq 0 \wedge i = s \wedge a+b = s) \\
\Leftarrow & \{ \text{ fixing values of some of variables (we use zero in this case) } \} \\
& (0 \geq s \wedge i = 0 \wedge a = 0 \wedge b = s) \vee \\
& (i \geq 0 \wedge i \geq s \wedge a = i \wedge b = s-i) \vee \\
& (s \geq 0 \wedge i = s \wedge a = s \wedge b = 0)
\end{array}$$

This calculation have derived a sufficient condition that $[a, b]$ obtained from (i, s) satisfies the necessary condition of the output of right inverses, $ms([a, b]) = (i, s)$. Therefore, the following function is a candidate of a right inverse.

$$\begin{aligned} ms^\bullet(i,s) = &\ \mathsf{if}\ 0 \geq s \wedge i = 0\ \mathsf{then}\ [0, s] \\ &\ \mathsf{else\ if}\ i \geq 0 \wedge i \geq s\ \mathsf{then}\ [i, s-i] \\ &\ \mathsf{else\ if}\ s \geq 0 \wedge i = s\ \mathsf{then}\ [s, 0] \end{aligned}$$

Notice that the function $ms^\bullet(i, s)$ is a partial function (besides, this definition is not in the language in Figure 7.1). As the previous calculation proves, the function $ms^\bullet(i, s)$ works as a right inverse of ms for inputs that are in its domain. However, because of its partiality, it may not a right inverse of ms. We will verify its correctness at the next step.

It is worth remarking that the "hoisting conditionals" step is the key step of the calculation. In that step, we eliminate the maximum operations, which is essentially conditional expressions, by pushing it as a precondition of equations; then, since the rest parts are written in usual arithmetic operations such as $+$ and $\times$, it is easy to solve them.

Verification

Next, we would like to verify that the derived candidate is a right inverse. It is sufficient to confirm that the domain of the candidate covers the range of the original function. Here, we use the following lemma [Mor07].

Lemma 7.2 ([Mor07])**.** Assume that a function f is defined as follows:

$$\begin{aligned} f([]) &= e \\ f([a] +\!\!+\, x) &= a \oslash f(x) \end{aligned}$$

Then, given a predicate p, $x \in \mathrm{ran}(f) \Rightarrow p(x)$ holds if $p(e) \wedge (p(r) \Rightarrow p(a \oslash r))$ holds.

Proof. It is straightforward from induction on lists. □

Let us confirm the correctness of $ms^\bullet$ we have derived. What we would like to prove is the inequality $\mathrm{ran}(ms) \subseteq \mathrm{dom}(ms^\bullet)$, and here, $\mathrm{dom}(ms^\bullet)$ is $\{(i, s) \mid (0 \geq s \wedge i = 0) \vee (i \geq 0 \wedge i \geq s) \vee (s \geq 0 \wedge i = s)\}$. First, it is apparent that the following formula holds.

$$ms([]) = (0, 0) \in \{(i, s) \mid (0 \geq s \wedge i = 0) \vee (i \geq 0 \wedge i \geq s) \vee (s \geq 0 \wedge i = s)\}$$

Next, from the definition of ms, what we would like to prove is the following formula.

$$\begin{aligned} &\forall i, s, a : ((0 \geq s \wedge i = 0) \vee (i \geq 0 \wedge i \geq s) \vee (s \geq 0 \wedge i = s)) \Rightarrow \\ &\quad ((0 \uparrow (a + i), a + s) \in \{(i, s) \mid (0 \geq s \wedge i = 0) \vee (i \geq 0 \wedge i \geq s) \vee (s \geq 0 \wedge i = s)\}) \end{aligned}$$

This can be seen as a first order formula on linear inequalities. Thus, the quantifier elimination techniques enable us to prove or disprove it. In this case, this formula

certainly holds. Therefore, $\text{ran}(ms) \subseteq \text{dom}(ms^\bullet)$ holds from Lemma 7.2, which implies the correctness of $ms^\bullet$.

The key is the use of quantifier elimination. By the virtue of the design of our language, all of the formulae we would like to prove/disprove are first order formulae on polynomial inequalities. Thus, quantifier elimination works as a mighty method.

When we fail in the verification, the only way we can do is to consider candidates of right inverses that yield longer lists than those we have considered. For example, assume that we had failed to verify $ms^\bullet$. Then, since we have considered a function that yields lists of length two, we might be able to obtain a right inverse of ms by considering functions that yield lists of length three. However, such an iterative procedure will not terminate for functions whose outputs strongly depend on the length of its inputs, such as *length*.

Improving Efficiency

So far, we have shown a way to obtain a correct right inverse. However, obtained right inverses are inefficient in general, because it may have a lot of conditional branches. We have adopted a naive strategy for dealing with branches, and much of them may be unnecessary.

Recall the definition of $ms^\bullet$, which has three branches: the case of $0 \geq s \wedge i = 0$, the case of $i \geq 0 \wedge i \geq s$, and the case of $s \geq 0 \wedge i = s$. We can observe the second branch is the most generic one in the sense that both of $(0 \geq s \wedge i = 0) \Rightarrow (i \geq 0 \wedge i \geq s)$ and $(s \geq 0 \wedge i = s) \Rightarrow (i \geq 0 \wedge i \geq s)$ hold. Thus, the second branch suffices to give a correct right inverse, and we obtain the following right inverse, which is more efficient than the previous one.

$$ms^\bullet(i, s) = \mathsf{if}\ i \geq 0 \wedge i \geq s\ \mathsf{then}\ [i, s - i]$$

The right inverse above is exactly the one that we have considered in Section 6.1.3.

Now let us formalize this improvement. Given a set of formula $\{\varphi_1, \varphi_2, \ldots, \varphi_n\}$, where each φ_i stands for the precondition of the ith branch, ith branch is unnecessary if the following property hold.

$$\varphi_i \Rightarrow (\bigvee_{1 \leq j \leq n \wedge i \neq j} \varphi_j)$$

Again, this formula can be proved or disproved by quantifier elimination techniques on our language.

7.3.2 Strong Points and Drawbacks

One of the most good characteristics of this method is that it is semantic approach in the sense the derivation scarcely depends on the syntactic representation of the function. Thus, it is unnecessary for users to be careful for writing programs. Another good characteristic is that this method is suitable for semi-automatic, namely

interactive, derivation. Even when users think of a candidate of right inverse, it is not trivial to prove its correctness. As seen, automatic theorem proving techniques are effective to verify that a candidate is a right inverse; besides, we have also shown automatic improvement methods.

One of the most critical drawbacks of this method is lack of capability of dealing with functions whose right inverses do not yield lists of length at most constant, which exclude functions like *length*. This restriction makes the coverage of this method small. Another issue is the efficiency of derived operators. As mentioned, naive derivations yield inefficient right inverses; besides, there is no guarantee that the inefficiency is completely removed by the improvement method proposed. Moreover, as mentioned in Section 6.1.3, even if an efficient right inverse is obtained, it is not easy to obtain an efficient associative operator from Lemma 6.3.

7.4 Parallelization via Candidate Generation and Associativity Testing

Next we would like to introduce another method for automatic parallelization, which consists of two steps. One is a generation of candidates of associative operators, and the other is the verification of associativity of candidates. More formally, given a function f, these two steps are summarized as follows.

1. Enumerate operators $\odot$ such that $f([a] +\!\!+ x) = f([a]) \odot f(x)$ holds.
2. Prove that a candidate $\odot$ satisfies $f(x +\!\!+ y) = f(x) \odot f(y)$.

This method is hopeless at a glance, because of the following two reasons. First, there will be too many candidates $\odot$ that satisfy the requirement $f([a] +\!\!+ x) = f([a]) \odot f(x)$. Second, there might exist no operator $\odot$ that satisfies $f(x +\!\!+ y) = f(x) \odot f(y)$. These two observations imply that naive implementation will be useless. Here is the place that the third list-homomorphism theorem takes its role. Based on the third list-homomorphism theorem, we can guarantee the existence of an operator $\odot$ satisfying $f(x +\!\!+ y) = f(x) \odot f(y)$; then, we would able to find the operator only by considering a rather small number of candidates.

In the following, we explain the method by using the function *mis* as an example. For simplicity, we do not consider the empty list as an initial segment, and thus, the input for the parallelization procedure is the following.

$$\begin{array}{lcl} mis([a]) & = & a \\ mis([a] +\!\!+ x) & = & a \uparrow (a + mis(x)) \\ mis(x +\!\!+ [a]) & = & mis(x) \uparrow (sum(x) + a) \\ sum([a]) & = & a \\ sum([a] +\!\!+ x) & = & a + sum(x) \\ sum(x +\!\!+ [a]) & = & sum(x) + a \end{array}$$

The input is written in our language in Figure 7.1, and it consists of a set of functions (mis and sum in this example) that are written in both leftward and rightward manners.

In the following, we will write $(f_1 \triangle f_2 \triangle \cdots \triangle f_k)$ to denote a function such that $(f_1 \triangle f_2 \triangle \cdots \triangle f_k)(x) = (f_1(x), f_2(x), \ldots, f_k(x))$ holds.

7.4.1 Generating Candidates and Testing Associativity

Preprocessing

Given a set of functions $F = \{f_1, f_2, \ldots, f_n\}$, we would like to obtain an operator $\odot$ such that $(f_1 \triangle f_2 \triangle \cdots \triangle f_n)(x +\!\!+ y) = (f_1 \triangle f_2 \triangle \cdots \triangle f_n)(x) \odot (f_1 \triangle f_2 \triangle \cdots \triangle f_n)(y)$ holds. Our strategy is to parallelize subsets of F from smaller ones to larger ones, so that we can reduce the number of candidates considered. For this purpose, we compute a sequence $[F_1, F_2, \ldots, F_m]$ such that the following four properties hold: (i) $F_i \subseteq F$; (ii) $F_m = F$; (iii) $i < j \Rightarrow F_i \not\supseteq F_j$; (iv) for each $F_i = \{f_{i1}, f_{i2}, \ldots, f_{ik}\}$, $(f_{i1} \triangle f_{i2} \triangle \cdots \triangle f_{ik})$ is both leftward and rightward. Note that the fourth property implies that each F_i is parallelizable. For example, from the set of functions $\{mis, sum\}$, we generate a sequence $[\{sum\}, \{mis, sum\}]$.

After the preprocessing, we perform the following two steps, namely candidate generation and associativity testing, for each F_i from $i = 1$ to $i = m$. Throughout the process, each function have a flag that stands for whether the function has been parallelized, and each parallelized function f_i has an expression $e^{\star}_{f_i}$ as the result of parallelization.

Candidate Generation

For each function f_i in the given set $\{f_1, f_2, \ldots, f_k\}$, let $e_{f_i} = e^{\star}_{f_i}$ if f_i has been parallelized before, and otherwise, generate an expression e_{f_i} that contains no variables a and y and satisfies the following equation, where $L_{f_1}, \ldots, L_{f_k}, R_{f_1}, \ldots, R_{f_k}$ are fresh variables.

$$f_i([a] +\!\!+ y) = e_{f_i}[f_1([a])/L_{f_1}, \ldots, f_k([a])/L_{f_k}, f_1(y)/R_{f_1}, \ldots, f_k(y)/R_{f_k}]$$

Then, a set of expressions $\{e_{f_1}, e_{f_2}, \ldots, e_{f_k}\}$ is the candidate that is processed by the next step, the associativity testing.

As an example, let us consider the case where the input of this step is $\{mis, sum\}$. In this situation, sum has been parallelized and an expression $e^{\star}_{sum} = L_{sum} + R_{sum}$ has been obtained. The function mis has not been parallelized yet, and we may generate $e_{mis} = L_{mis} \uparrow (L_{sum} + R_{mis})$ for mis, because $mis([a] +\!\!+ x) = mis([a]) \uparrow (sum([a]) + mis(x))$ holds. Then, $\{e^{\star}_{sum}, e_{mis}\}$ is a candidate and will be processed at the next step.

Associativity Testing

Given a candidate $\{e_{f_1}, e_{f_2}, \ldots, e_{f_k}\}$, try to prove the following formula.

$$\begin{array}{l} \forall e_{f_i} \in \{e_{f_1}, e_{f_2}, \ldots, e_{f_k}\} : \\ \quad f_i(x +\!\!+ y) = e_{f_i}[f_1(x)/L_{f_1}, \ldots, f_k(x)/L_{f_k}, f_1(y)/R_{f_1}, \ldots, f_k(y)/R_{f_k}] \end{array}$$

It is unnecessary to prove the formula about e_{f_i} when f_i has been parallelized.

If we successfully proved the formula, then let $e^\star_{f_i} = e_{f_i}$ for each f_i; in other words, f_i is parallelized. Otherwise, return to the candidate generation step. The parallelization ends in failure when the formula does not hold for all considerable candidates.

The formula certainly holds for a candidate $\{e^\star_{sum}, e_{mis}\}$ where $e^\star_{sum} = L_{sum}+R_{sum}$ and $e_{mis} = L_{mis} \uparrow (L_{sum} + R_{mis})$. Thus, $e^\star_{mis} = L_{mis} \uparrow (L_{sum} + R_{mis})$ and mis is successfully parallelized.

Output

Given a set of functions $\{f_1, \ldots, f_n\}$ and corresponding expressions $e^\star_{f_1}, \ldots, e^\star_{f_n}$, output the following operator $\odot$.

$$(L_{f_1}, \ldots, L_{f_k}) \odot (R_{f_1}, \ldots, R_{f_k}) = (e^\star_{f_1}, \ldots, e^\star_{f_k})$$

The associativity testing proves that the operator satisfies the following equation.

$$(f_1 \vartriangle f_2 \vartriangle \cdots \vartriangle f_n)(x +\!\!+ y) = (f_1 \vartriangle f_2 \vartriangle \cdots \vartriangle f_n)(x) \odot (f_1 \vartriangle f_2 \vartriangle \cdots \vartriangle f_n)(y)$$

For the case of mis, the following operator is outputted.

$$(L_{sum}, L_{mis}) \odot (R_{sum}, R_{mis}) = (L_{sum} + R_{sum}, L_{mis} \uparrow (L_{sum} + R_{mis}))$$

This operator is certainly the operator that we have derived several times.

Characteristics

One of the characteristics of this method is that it is based on candidate generation and associativity testing. Therefore, the way to manage these two steps is the issue to perform effective parallelization.

For generating candidates, it seems appropriate to generate only expressions of low computational cost, because costly computations are unnecessary for parallel computations. Another interesting strategy to generate candidates is use of other automatic parallelization methods: we use the result of an automatic parallelization method as a candidate. Then, this method can be recognized as a method to compose automatic parallelization methods and verify their results.

For associativity testing, use of automatic theorem proving techniques is a natural way. There are a lot of studies about automatic theorem proving, and we can choose appropriate one according to our objectives.

Figure 7.2. The dependency graph of the functions *mis* and *sum*.

7.4.2 Implementation

Following the previous subsection, we implemented the method as an automatic parallelization system. The system is written in Haskell [Pey03].

Preprocessing

As a preprocessing, we would like to obtain the subset of the inputted functions such that the subset is parallelizable. For this purpose, we consider a graph whose vertexes and edges are respectively functions and caller-callee dependencies between functions, and apply strongly connected component decomposition and topological sorting to the graph.

Figure 7.2 shows the graph corresponding to the set of functions $\{mis, sum\}$. Since *mis* calls *sum* in its rightward definition, there is an edge from *sum* to *mis*. Then, strongly connected component decomposition yields two strongly connected components, $\{sum\}$ and $\{mis\}$, and topological sorting on strongly connected components yields an ordering $[\{sum\}, \{mis\}]$. Therefore, we first parallelize *sum*, and after that, we parallelize $\{mis\}$ by using the information of $\{sum\}$.

Candidate Generation

In our implementation, the system generates candidates by replacing subexpressions of by base-case definition. Let us see the candidate generation strategy through an example. Since both $mis([a])$ and $sum([a])$ yields a, replacing an expression a by $mis([a])$ or $sum([a])$ does not change the value of expressions. Hence, from the equation that defines the step case of *mis*, namely $mis([a] +\!\!+ x) = a \uparrow (a + mis(x))$, we can obtain the following nine equations by considering three cases for each occurrence of a: replacing it by $mis([a])$, replacing it by $sum([a])$, and leaving it unchanged.

$$
\begin{aligned}
&mis([a] +\!\!+ x) = a \uparrow (a + mis(x))\\
&mis([a] +\!\!+ x) = a \uparrow (mis([a]) + mis(x))\\
&mis([a] +\!\!+ x) = a \uparrow (sum([a]) + mis(x))\\
&mis([a] +\!\!+ x) = mis([a]) \uparrow (a + mis(x))\\
&mis([a] +\!\!+ x) = sum([a]) \uparrow (a + mis(x))\\
&mis([a] +\!\!+ x) = mis([a]) \uparrow (mis([a]) + mis(x))\\
&mis([a] +\!\!+ x) = mis([a]) \uparrow (sum([a]) + mis(x))\\
&mis([a] +\!\!+ x) = sum([a]) \uparrow (mis([a]) + mis(x))\\
&mis([a] +\!\!+ x) = sum([a]) \uparrow (sum([a]) + mis(x))
\end{aligned}
$$

Since equations that contain a as their direct subexpressions are useless as candidates, we throw away them; then four expressions are left. The candidates are

expressions that is obtained from the right hand side expressions of the four equations by replacing occurrences of function calls by corresponding variables: replacing $mis([a])$ by L_{mis}, $sum([a])$ by L_{sum}, $mis(x)$ by R_{mis}, and $sum(x)$ by R_{sum}. In summary, we obtain the following four candidates for mis.

$$\begin{array}{c} L_{mis} \uparrow (L_{mis} + R_{mis}) \\ L_{mis} \uparrow (L_{sum} + R_{mis}) \\ L_{sum} \uparrow (L_{mis} + R_{mis}) \\ L_{sum} \uparrow (L_{sum} + R_{mis}) \end{array}$$

Associativity Testing

For associativity testing, we use induction on lists and quantifier elimination techniques.

Let us consider the case where the candidate is $\{e_{mis}, e^{\star}_{sum}\}$, which we have considered in the previous section. We would like to prove the following equation.

$$mis(x +\!\!+ y) = mis(x) \uparrow (sum(x) + mis(y))$$

The equation holds when the length of x is 1, which is a direct consequence of the candidate generation step. When the length of x is longer than 1, namely $x = [a] +\!\!+ z$, we calculate as follows.

$$\begin{array}{l} mis([a] +\!\!+ z +\!\!+ y) = mis([a] +\!\!+ z) \uparrow (sum([a] +\!\!+ z) + mis(y)) \\ \quad \Leftrightarrow \quad \{ \text{ definition of } mis \ \} \\ \qquad a \uparrow (a + mis(z +\!\!+ y)) = mis([a] +\!\!+ z) \uparrow (sum([a] +\!\!+ z) + mis(y)) \\ \quad \Leftrightarrow \quad \{ \text{ induction hypothesis } \} \\ \qquad a \uparrow (a + (mis(z) \uparrow (sum(z) + mis(y)))) \\ \qquad\quad = mis([a] +\!\!+ z) \uparrow (sum([a] +\!\!+ z) + mis(y)) \\ \quad \Leftrightarrow \quad \{ \text{ definitions of } mis \text{ and } sum \ \} \\ \qquad a \uparrow (a + (mis(z) \uparrow (sum(z) + mis(y)))) \\ \qquad\quad = (a \uparrow (a + mis(z))) \uparrow ((a + sum(z)) + mis(y)) \end{array}$$

In summary, we would like to prove the last equation holds for all a, z, and y. To prove this is not easy, because we need to infer results of recursive functions. Instead of proving the last equation, we assume that each function call will yield arbitrary values and try to prove the following equation.

$$\begin{array}{l} \forall a, L_{mis}, L_{sum}, R_{mis} : \\ \quad a \uparrow (a + (L_{mis} \uparrow (L_{sum} + R_{mis}))) = (a \uparrow (a + L_{mis})) \uparrow ((a + L_{sum}) + R_{mis}) \end{array}$$

We can prove/disprove this equation by quantifier elimination techniques.

We implemented two quantifier elimination techniques: Fourier-Motzkin elimination [DE73] and the method by Loos and Weispfenning [LW93]. Because Fourier-Motzkin elimination is terribly inefficient for nonlinear expressions, the implementation of Fourier-Motzkin elimination specializes in the linear case. It is efficient for

```
struct my_tuple_t {
  int mis, sum;
  my_tuple_t(int mis, int sum) : mis(mis), sum(sum){}
  my_tuple_t(){}
};

struct func_t : public sketo::functions::base<my_tuple_t(int)> {
  my_tuple_t operator()(const int a) const {
    const int mis = a;
    const int sum = a;
    return my_tuple_t(mis, sum);
  }
} func;

struct odot_t : public sketo::functions::base
    <my_tuple_t(my_tuple_t,my_tuple_t)> {
  my_tuple_t operator()(const my_tuple_t &x, const my_tuple_t &y) const {
    const int mis = std::max(x.mis,(x.sum+y.mis));
    const int sum = (x.sum+y.sum);
    return my_tuple_t(mis, sum);
  }
} odot;
```

Figure 7.3. The output that the parallelizer generates for *mis*.

this case because we only eliminate universal quantifiers and elimination of universal quantifiers does not breaks disjunctive normal forms. In addition, simplification of formula is effective in practice [DS97], and we implement the following three simplifications: flattering expressions, common subexpressions elimination, and evaluation of subexpressions whose values are fixed.

We implemented the associativity testing step as a parallel program, because the associativity testing for each candidate is independent. It is a strong point of the method that it is suitable for parallel implementation.

Output

The output of the system is C++ codes that is available form parallel skeleton library SkeTo [MIEH06] of developers' version.

Figure 7.3 is the output for *mis*. The output consists of a function `func_t` for singleton lists, an associative operator `odot_t` for merging results of sublists, and a structure `my_tuple_t` for communications between processors.

Strong Points and Drawbacks

One of the good characteristics of this implementation is efficiency of generated parallel programs: each generated program is efficient in the sense the merging

Table 7.1. The number of candidates generated for each problem.

	length	*max*	*mis*	*mip*	*mss*	*atoi*
number of candidates	2	1	5	730	24	3

Table 7.2. The computational times for parallelization (unit: second)

	length	*max*	*mis*	*mss*	*mip*	*atoi*
Fourier-Motzkin elimination	0.01	0.01	0.01	0.04	N/a	N/a
method by Loos and Weispfenning	0.01	0.01	0.01	5.25	N/a	0.01

operator is less costly than sequential ones because of our candidate generation strategy. Thus, it is unnecessary to develop neat efficiency improvement methods for this implementation.

The drawback of this implementation is that it is fully syntactic: Success in parallelization highly depends on the syntactic representation of programs. Moreover, it may be necessary for parallelization to introduce useless terms in the input program, such as $+0$ and $\times 1$. We will have a more detailed discussion about this issue in the next section.

7.4.3 Experiments

To confirm effectiveness of the method, we did some experiments. The computational environment of our experiments is the following: Intel Core2 Duo 1.8 GHz CPU, 1.5 GB memory, Windows XP SP3, and GHC 6.8.2. We considered six examples: function *length* that computes the length of the list, function *max* that yields the maximum element in the list, function *mis* that computes the maximum initial-segment sum of the list, function *mip* that computes the maximum initial-segment product of the list, function *mss* that computes the maximum segment sum of the list, and function *atoi* that translates the sequence of characters into a number.

Table 7.1 shows the numbers of candidates generated by our systems. Except for the case of *mip*, not many candidates are considered for each problem. The reason why many candidates are generated for *mip* is that it is necessary to parallelize plural functions at once. Numbers of candidates become larger in such cases, because we need to consider combinations of candidates of each function.

Table 7.2 shows computational times of our two parallelization systems, which respectively use Fourier-Motzkin elimination and the method by Loos and Weispfenning as their quantifier elimination procedures. For most of the examples, the parallelizers generate parallel codes immediately. In the case of *mss*, the Fourier-Motzkin-elimination-based system runs apparently faster than the Loos-Weispfenning-based system, because the former uses the specialized efficient implementation of quantifier elimination. While it, the Fourier-Motzkin-elimination-based system cannot deal with *atoi*, because the definition of *atoi* contains nonlinear expressions. This fact indicates that it is very important to design an appropriate language so that

we can cope with our objective by efficient implementations.

Our parallelizers could not parallelize *mip*. The reason is the lack of powerful quantifier elimination method. In theory, *mip* can be dealt in generation-and-testing method by using cylindrical algebraic decomposition; however, because candidates considered for *mip* is large and cylindrical algebraic decomposition is a costly method, this approach may not be practical for the case.

7.5 Comparison of two Automatic Parallelization Methods

In this section, we would like to give a comparison of two automatic parallelization methods, namely the inversion-based method and the generation-and-testing-based method. In brief, they have complementary strong and weak points, and combining them could be an interesting future work.

7.5.1 Comparing Efficiency

First, we would like to compare efficiency of these methods.

Apparently, the most costly steps are the steps of quantifier elimination, and thus, efficiency of the quantifier elimination procedures affects efficiency of parallelization procedures a lot for both methods. Thus, this issue raises little difference of two methods.

Next, consider the number of quantifier elimination steps required in a parallelization. In the inversion-based method, the number mainly depends on the number of conditional branches in the source program and the length of lists that the right inverse will yield. In usual, considering longer lists are required when many functions are necessary for providing leftward and rightward definitions. Therefore, the inversion-based method is inefficient when the input program consists of many functions with many conditional expressions. In the generation-and-testing-based method, the number of quantifier elimination steps highly depends on the number of candidates generated. Therefore, as mentioned, the generation-and-testing-based method is not effective when plural functions should be parallelized at once, namely when the source program consists of mutually recursive functions.

7.5.2 Comparing Feasibility

The generation-and-testing method is sensitive to syntactic representation of functions. We often fail to generate appropriate candidates unless we consider other programs equivalent to the original program, which would be obtained by using associativity, distributivity, introduction of units, and more problem-specific information.

As an example, recall that we considered the "non-empty" maximum initial-segment sum problem for demonstrating the method. Now consider the usual maximum initial-segment sum problem, in which the empty list is also considered as an initial segment.

$$\begin{array}{lcl} mis([a]) & = & 0 \uparrow a \\ mis([a] +\!\!+ x) & = & 0 \uparrow (a + mis(x)) \end{array}$$

For this program, we cannot generate any candidates from our candidate generation strategy. In fact, the following a bit peculiar program can be dealt with.

$$\begin{array}{lcl} mis([a]) & = & 0 \uparrow a \\ mis([a] +\!\!+ x) & = & (0 \uparrow a) \uparrow (a + mis(x)) \end{array}$$

This program is equivalent to the previous one, because $mis(x) \geq 0$ holds for any list x; besides, we can generate an appropriate candidate for this program.

Another difficulty in generation-and-testing method arises when we would like to deal with recursive functions whose values relate each other. Consider the following peculiar summation program.

$$\begin{array}{lcl} sum1([a]) & = & a \\ sum1([a] +\!\!+ x) & = & a + sum2(x) \\ sum2([a]) & = & a \\ sum2([a] +\!\!+ x) & = & a + sum1(x) \end{array}$$

The program computes the summation in a mutual recursive manner. Then, for verifying associativity of a candidate, we need to prove the following equation.

$$\forall a, x, y : \ a + sum1(x) + sum2(y) = a + sum2(x) + sum2(y)$$

In truth, the equation holds because $sum1(x) = sum2(x)$ holds for any list x; however, in our implementation, we approximate the equation by the following recursion-free equation so as to apply quantifier elimination techniques.

$$\forall a, L_{sum1}, L_{sum2}, R_{sum2} : \ a + L_{sum1} + R_{sum2} = a + L_{sum2} + R_{sum2}$$

Then, the equation does not hold, and we fail in parallelization of the program.

The inversion-based method does not have such weak points. It is rarely affected with syntactic variations or constraint on function values. However, as mentioned, it cannot deal with length-related functions such as *length* and *atoi*. Moreover, we can hardly specify which function is infeasible by the inversion-based method. Thus, methods to resolve this restriction are required.

7.5.3 Relationship between two Methods

Next, we would like to discuss a relationship between the two methods. We would like to start our discussion from the function *length*, which is defined as follows.

$$\begin{array}{lcl} length([a]) & = & 1 \\ length([a] +\!\!+ x) & = & 1 + length(x) \end{array}$$

We cannot parallelize the program by the inversion-based method. In fact, it is unnecessary to derive an associative operator corresponding to *length* for parallelization of *length*. Notice that $\mathit{length} = \mathit{sum} \circ \mathsf{map}_{\mathsf{const}_1}$ holds and $\mathsf{map}_{\mathsf{const}_1}$ is trivially parallelizable. Therefore, it is *sum* that we would like to derive an associative operator for, and deriving an associative operator for *sum* is trivial.

Now the issue is how to decompose *length* to $\mathit{sum} \circ \mathsf{map}_{\mathsf{const}_1}$. Note that a list homomorphism can be decomposed into foldr with map, namely $\mathsf{hom}_{\odot,f} = \mathsf{foldr}_{\odot,\iota_\odot} \circ \mathsf{map}_f$ where $\iota_\odot$ is the unit of $\odot$; thus, consider decomposing *length* to foldr with map. The parameter of map can be determined from the result of *length* for singleton lists, because $\mathsf{foldr}_{\odot,\iota_\odot}([a]) = a$ holds. Then, since all elements in the input list become 1 by $\mathsf{map}_{\mathsf{const}_1}$, the 1 in the definition of the step case can be replaced by the list element a. In summary, we obtain the decomposition $\mathit{length} = \mathsf{foldr}_{+,0} \circ \mathsf{map}_{\mathit{length} \circ \mathit{wrap}}$, which is what we want to.

While this decomposition and replacement method seems clever, such replacements generally break parallelizability. For a function f that is a list homomorphism, a function g satisfying $f = g \circ \mathsf{map}_{f \circ \mathit{wrap}}$ may not be a list homomorphism. Therefore, for utilizing the replacement method, we should confirm that the obtained candidate (such as g above) is certainly a list homomorphism. However, we found that we could frequently obtain an associative operator while confirming that the candidate is a list homomorphism. The generation-and-testing method is based on this observation. If there exists an effective and efficient method to check whether a candidate is a list homomorphism or not, it may be effective to apply inversion-based method for the candidate that is a list homomorphism.

7.6 Summary and Discussions

In this section, we have developed automatic parallelization method based on the third list-homomorphism theorem. We have introduced two methods. One is based on inversion, in which we derive right inverses of functions and obtain associative operators. The other is based on generation-and-testing, in which we enumerate candidates of associative operators and try to prove their associativity. For both methods, the key to efficient parallel programs is the use of automatic theorem proving techniques in addition to the third list-homomorphism theorem. Automatic theorem proving techniques are useful not only for verification but also efficiency improvements. Here we have concentrated on quantifier elimination techniques and designed a programming language that is suitable for the use of quantifier elimination. Then, we can derive parallel programs for nontrivial problems in practical costs. Since our parallelization methods rely on existing studies about automatic theorem proving and automatic inversion, research progress on these topics will improve effectiveness of our methods.

There are many studies about automatic parallelization; however, most of them struggled to find known parallelizable computational patterns from programs [RF93,

SKN96, Pot98, GGL06]. While importance of reduction operations is well recognized [DCCS06], most of the existing studies only consider specific cases of reduction operations such as addition and multiplications, and fewer studies considered generating associative operators that enhance parallel computations.

To our knowledge, there are two approaches to automatic parallelization of reductions, or especially automatic derivation of list homomorphisms. One is based on the third list-homomorphism theorem, and the other is based on the closure property for function compositions.

In this chapter, we introduced automatic parallelization methods based on the third list-homomorphism theorem. The strong point of the theorem is that users can provide useful information to systems by writing both leftward and rightward programs. As discussed in Chapter 6, additional information should be necessary for parallelization in general. The use of the third list-homomorphism theorem enables us to specify additional information naturally.

Geser and Gorlatch [GG99] also proposed an automatic parallelization method based on the third list-homomorphism theorem, which also adopts generation-and-testing strategy. Their system first generates a function that is a generalization of both of the leftward and rightward programs, and checks associativity of the operator raised by the function. We have shown that even a more sloppy and naive strategy works well. For dealing with more difficult problems, their strategy will be effective to reduce the number of candidates.

While the third list-homomorphism theorem is effective for automatic parallelization, writing two programs is a bother. It is an important research topic to clarify effectiveness of the additional information supplied from the third list-homomorphism theorem and to what extent we can obtain a parallel program from a sequential program.

It is a folk fact that we can perform efficient parallel computation of a loop if compositions of functions corresponding an iteration of the loop can be computed efficiently [Cal92, CTH98]. Based on this fact, some automatic parallelization methods are proposed. Callahan [Cal92] showed automatic parallelization of differential computations. Fisher and Ghuloum [FG94] did a solid study for function-composition-based automatic parallelization. Xu et al. [XKH04] showed that this strategy can be used in ease if users tell algebraic properties of primitive operators to the system, such as distributivity and associativity. These results are attractive because we can obtain a parallel program from only one sequential program. However, as the study of Xu et al. indicates, it may be necessary to supply some additional information to system for effective parallelization.

As a final remark of this chapter, we would like to give a brief discussion about how to parallelize functions that do not exactly match with list homomorphisms.

Prefix and postfix computations, namely scanl and scanr, are important classes of computations in parallel programming [Ble89, LD94]. It is well known both $\mathsf{scanl}_{\oplus,e}$ and $\mathsf{scanr}_{\oplus,e}$ are computable in parallel if $\oplus$ is associative. Since we have developed methods to derive associative operators, the methods are also applicable for deriving

parallel programs for prefix/postfix computations, once we can identify them.

In existent studies [FG94, CTH98, HTI99], several people proposed methods to identify prefix/postfix computations and decompose them from reduction computations. Their methods are similar to the method used in generation-and-testing parallelization for performing incremental parallelization. In the paper of Morita et al. [MMM+07], another approach to identifying prefix/postfix computations was proposed. They prepare a languages to force user to write prefix and postfix computations explicitly, which are identified as the patterns $\mathsf{map}_f \circ \mathit{inits}$ and $\mathsf{map}_f \circ \mathit{tails}$, respectively. Since $\mathsf{map}_{\mathsf{hom}_{\odot,g}} \circ \mathit{inits} = \mathsf{scanl}_{\odot,\iota_\odot} \circ \mathsf{map}_g$ and $\mathsf{map}_{\mathsf{hom}_{\odot,g}} \circ \mathit{tails} = \mathsf{scanr}_{\odot,\iota_\odot} \circ \mathsf{map}_g$ hold, where $\iota_\odot$ is the unit of $\odot$, the parallelization problems of the patterns are reduced into the parallelization problem of f; thus, parallelization methods for reductions are directly applicable.

As seen in Sections 6.2 and 6.3, list homomorphisms can characterize parallel computations on not only lists but also trees. Therefore, we will hopefully be able to develop automatic parallelization methods for functions traversing trees based on parallelization methods on lists. However, we have not developed any results so far, and it is a topic of further research.

Chapter 8

Conclusion

8.1 Summary of the Thesis

In this thesis, we studied a calculational approach to automatic construction of efficient algorithms. We mainly considered construction of two kinds of algorithm patterns: dynamic programming algorithms and divide-and-conquer algorithms. For both patterns, we prepared calculational laws for automatic construction of algorithms of the patterns, designed languages for automatic implementation of the calculational laws, and proposed systems that automatically derive efficient programs.

As the former part, we discussed derivation of efficient algorithms for combinatorial optimization problems. We demonstrated that it was important for algorithm development to specify structures of problems, such as structures of generating candidates, structures of orders to optimize (or objective functions), and structures of constraints that solutions should satisfy. Based on them, we proposed calculational laws that enabled us to derive and confirm monotonicity conditions in ease. Our calculational laws are effective for deriving dynamic programming algorithms, as we confirmed it by deriving efficient algorithms for shortest path problems and their variants. To bring our calculational laws to practical uses, we designed a domain-specific language for optimal path querying. We demonstrated that the domain-specific language enabled us to perform efficient optimal path querying for a large set of problems.

As the latter part, we discussed parallelization problems. We focused on the third list-homomorphism theorem, which is useful to reveal additional information that is effective to construct divide-and-conquer parallel programs. We confirmed that theories on lists can be generalized to trees by considering divide-and-conquer algorithms on paths, and introduced the third list-homomorphism theorem on trees. The theorem is generic in the sense that it can deal with all polynomial data structures. We demonstrated that the third list-homomorphism theorem is useful for developing efficient parallel programs not only on lists but also on trees by several examples. In addition, we proposed automatic implementations of parallelization. Our imple-

mentations are based on a language that is designed so as to utilize not only the third list-homomorphism theorem but also quantifier elimination techniques, which are useful for verifying correctness and improving efficiency of derived programs.

The key was to specify latent information that was effective for constructing efficient algorithms. In the case of combinatorial optimization problems, we focused on incrementality condition, which formed an interface to efficient algorithms, and based on the interface, we designed a domain-specific language for optimal path querying. In the case of parallelization, we focused on a set of functions that implied existence of an associative operator, and then, even such a naive procedure as brute-force searching could derive parallel programs. Another key was design of languages. Syntactic restrictions on languages clarify structures of problems and supply useful information to program transformation systems.

8.2 Future Works

Our study is a small step to automatic construction of algorithms. We only considered two algorithm patterns, and our automatic algorithm construction methods could deal with only restricted class of problems. Many problems are left unsolved, even if we exclude technical ones, and there are many interesting directions for further studies.

An important and interesting topic is to study automatic algorithm construction of another algorithm pattern. Especially, greedy algorithms are one of the most interesting algorithm patterns, because they are very efficient but hard to prove correctness. Approximation algorithms and randomized algorithms are also interesting patterns, because they are very useful in practice.

Another is to provide a unified language on which we can automatically derive several kinds of algorithm patterns. In this thesis, we independently designed a language for each case to automate derivation of efficient algorithms. However, the languages designed are similar to each other: a set of catamorphisms that consist of conditional expressions of restricted forms. Catamorphisms enable us to utilize induction, which is very useful to verify properties. Conditional expressions reinforce expressiveness of languages, while conditional expressions are generally difficult to deal with. In short, automatic algorithm development may consist of certain patterns, which would be captured by a language for it.

It is also interesting to study combination of plural automatic algorithm construction frameworks. We considered neither further improvement nor automatic derivation of efficient implementation for derived algorithm patterns. Automatic algorithm construction frameworks will be suitable for combining each other, because we can predict derived programs.

Last but not the least, it should be promising to study intensive use of automatic theorem proving techniques in automatic algorithm construction. As seen, derivation of efficient algorithms usually requires to verify certain properties, such

as monotonicity and associativity, and it is one of the hardest steps in automatic algorithm construction. Automatic theorem proving techniques will form ready-made program analysis tools and help us to verify such properties.

We are hoping that a lot of algorithms will be derived automatically.

シェクスピイアも、ゲエテも、李太白も、近松門左衛門も滅びるであろう。しかし芸術は民衆の中に必ず種子を残している。わたしは大正十二年に「たとい玉は砕けても、瓦は砕けない」と云うことを書いた。この確信は今日でも未だに少しも揺るがずにいる。

又

打ち下ろすハンマアのリズムを聞け。あのリズムの存する限り、芸術は永遠に滅びないであろう。（昭和改元の第一日）

又

わたしは勿論失敗だった。が、わたしを造り出したものは必ず又誰かを作り出すであろう。一本の木の枯れることは極めて区々たる問題に過ぎない。無数の種子を宿している、大きい地面が存在する限りは。（同上）

（芥川龍之介「侏儒の言葉（遺稿）」[1]より）

[1]青空文庫: www.aozora.gr.jp.

Bibliography

[ADKP89] Karl R. Abrahamson, N. Dadoun, David G. Kirkpatrick, and Teresa M. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.

[ALS91] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991.

[BBJ+02] Christopher L. Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, and Madhav V. Marathe. Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router. In *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2002.

[BBJ+07] Christopher L. Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner. Label constrained shortest path algorithms: An experimental evaluation using transportation networks. Technical report, Virginia Tech (USA), Arizona State University (USA), and Karlsruhe University (Germany), 2007.

[BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BdM92] Richard S. Bird and Oege de Moor. Between dynamic programming and greedy: Data compression. Programming Research Group, 11 Keble Road, Oxford OX1 3QD, England, 1992.

[BdM93a] Richard S. Bird and Oege de Moor. From dynamic programming to greedy algorithms. In *Formal Program Development - IFIP TC2/WG 2.1 State-of-the-Art Report*, volume 755 of *Lecture Notes in Computer Science*, pages 43–61. Springer, 1993.

[BdM93b] Richard S. Bird and Oege de Moor. Solving optimisation problems with catamorphisms. In *Mathematics of Program Construction, 2nd*

International Conference, Oxford, U.K., June 29 - July 3, MPC 1992, Proceedings, volume 669 of *Lecture Notes in Computer Science*, pages 45–69, Springer 1993.

[BdM96] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1996.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Ben86] Jon Bentley. *Programming Pearls*. ACM, 1986.

[BGW76] Robert Balzer, Neil M. Goldman, and David S. Wile. On the transformational implementation approach to programming. In *Proceedings of the 2nd International Conference on Software Engineering, 13-15 October 1976, San Francisco, California*, pages 337–344. IEEE Computer Society, 1976.

[Bir84] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir87] Richard S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer, 1987. *Technical Monograph PRG*-56.

[Bir89] Richard S. Bird. Algebraic identities for program calculation. *Computer Journal*, 32(2):122–126, 1989.

[Bir01] Richard S. Bird. Maximum marking problems. *Journal of Functional Programming*, 11(4):411–424, 2001.

[Bir06] Richard S. Bird. Loopless functional algorithms. In *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings*, volume 4014 of *Lecture Notes in Computer Science*, pages 90–114. Springer, 2006.

[BJ04] Gerth Stølting Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science*, 92:3–15, 2004.

[BJJM99] Roland Carl Backhouse, Patrik Jansson, Johan Jeuring, and Lambert G. L. T. Meertens. Generic programming: An introduction. In *Advanced Functional Programming, Third International School, Braga, Portugal, September 12-19, 1998, Revised Lectures*, volume 1608 of *Lecture Notes in Computer Science*. Springer, 1999.

[BJM00] Christopher L. Barrett, Riko Jacob, and Madhav V. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[Ble89] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[Bod96] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[BPT92] Richard B. Borie, R. Gary Parker, and Craig A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7(5&6):555–581, 1992.

[BvdEvG94] Roland Carl Backhouse, J. P. H. W. van den Eijnde, and A. J. M. van Gasteren. Calculating path algorithms. *Science of Computer Programming*, 22(1–2):3–19, 1994.

[Cal92] David Callahan. Recognizing and parallelizing bounded recurrences. In *Languages and Compilers for Parallel Computing, Fourth International Workshop, Santa Clara, California, USA, August 7-9, 1991, Proceedings*, volume 589 of *Lecture Notes in Computer Science*, pages 169–185. Springer, 1992.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM'93*, pages 119–132. ACM, 1993.

[Chv83] Vaclay Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.

[CJ98] Bob F. Caviness and Jeremy. R. Johnson, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer, 1998.

[CJdP07] Barbara Chapman, Gabriele Jost, and Ruud Van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[Col75] George E. Collins. Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages, 2nd GI Conference, Kaiserslautern, May 20-23, 1975*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer, 1975.

[Col94] Murray Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. In *Parallel Computing: Trends and Applications, PARCO 1993, Grenoble, France*, pages 489–492. Elsevier, 1994.

[Col95] Murray Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5:191–203, 1995.

[CP89] Jiazhen Cai and Robert Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, 1989.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to algorithms, Second edition.* MIT Press, 2001.

[CTH98] Wei-Ngan Chin, Akihiko Takano, and Zhenjiang Hu. Parallelization via context preservation. In *Proceedings of the 1998 International Conference on Computer Languages, ICCL'98, May 14-16, 1998, Chicago, IL, USA*, pages 153–162. IEEE Computer Society, 1998.

[Cur96] Sharon A. Curtis. *A Relational Approach to Optimization Problems.* PhD thesis, Oxford University Computing Laboratory, 1996.

[Cur97] Sharon A. Curtis. Dynamic programming: a different perspective. In *Algorithmic Languages and Calculi, IFIP TC2 WG2.1 International Workshop on Algorithmic Languages and Calculi, 17-22 February 1997, Alsace, France*, volume 95 of *IFIP Conference Proceedings*, pages 1–23. Chapman & Hall, 1997.

[Cur03] Sharon A. Curtis. The classification of greedy algorithms. *Science of Computer Programming*, 49(1–3):125–157, 2003.

[CV88] Richard Cole and Uzi Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.

[CW72] Thomas E. Cheatham, Jr and Ben Wegbreit. A laboratory for the study of automating programming. *SIGSAM Bulletin*, (21):8–26, 1972.

[CZ07] Edward P. F. Chan and Jie Zhang. A fast unified optimal route query evaluation algorithm. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, pages 371–380. ACM, 2007.

[DCCS06] Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence Snyder. Global-view abstractions for user-defined reductions and scans. In *Proceedings of the ACM SIGPLAN Symposium on*

Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006, pages 40–47. ACM, 2006.

[DE73] George B. Dantzig and B. Curtis Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory, Series A*, 14(3):288–297, 1973.

[DK92] Arthur L. Delcher and Simon Kasif. Efficient parallel term matching and anti-unification. *Journal of Automated Reasoning*, 9(3):391–406, 1992.

[dM92] Oege de Moor. *Categories, Relations and Dynamic Programming.* PhD thesis, Oxford University Computing Laboratory, 1992. *Technical Monograph PRG*-98.

[dM95] Oege de Moor. A generic program for sequential decision processes. In *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings*, volume 982 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 1995.

[dMG99] Oege de Moor and Jeremy Gibbons. Bridging the algorithm gap: A linear-time functional program for paragraph formatting. *Science of Computer Programming*, 35(1):3–27, 1999.

[dMG00] Oege de Moor and Jeremy Gibbons. Invited talk: Pointwise relational programming. In *Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 371–390. Springer, 2000.

[dMLW03] Oege de Moor, David Lacey, and Eric Van Wyk. Universal regular path queries. *Higher-Order and Symbolic Computation*, 16(1–2):15–35, 2003.

[dMS01] Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1–2):135–162, 2001.

[DS88] Martin Desrochers and François Soumis. A generalized permanent labeling algorithm for the shortest path problem with time windows. *INFOR*, 26:191–212, 1988.

[DS97] Andreas Dolzmann and Thomas Sturm. Simplification of quantifier-free formulae over ordered fields. *Journal of Symboric Computation*, 24(2):209–231, 1997.

[Dur02] Juan Eduardo Durán. Transformational derivation of greedy network algorithms from descriptive specifications. In *Mathematics of Program Construction, 6th International Conference, MPC 2002, Dagstuhl Castle, Germany, July 8-10, 2002, Proceedings*, volume 2386 of *Lecture Notes in Computer Science*, pages 40–67. Springer, 2002.

[Edm71] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:127–136, 1971.

[Epp98] David Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.

[Erw97] Martin Erwig. Functional programming with graphs. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming, ICFP'97, Amsterdam, The Netherlands, June 9-11, 1997*, pages 52–65. ACM, 1997.

[Erw00] Martin Erwig. Random access to abstract data types. In *Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings*, pages 135–149. Springer, 2000.

[Erw01] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[FFG02] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *Journal of the ACM*, 49(6):716–752, 2002.

[FFG06] Sergio Flesca, Filippo Furfaro, and Sergio Greco. Weighted path queries on semistructured databases. *Information and Computation*, 204(5):679–696, 2006.

[FG94] Allan L. Fisher and Anwar M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, June 20-24, 1994*, pages 135–146, 1994.

[Fok89] Maarten M. Fokkinga. Tupling and mutumorphisms. *Squiggolist*, 1(4), 1989.

[Fok92] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[Fra81] András Frank. A weighted matroid intersection algorithm. *Journal of Algorithms*, 2(4):328–336, 1981.

[GCS94] Jeremy Gibbons, Wentong Cai, and David B. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, 23(1):1–18, 1994.

[Ger75] Susan L. Gerhart. Correctness-preserving program transformations. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, California, January 1975*, pages 54–66. ACM, 1975.

[GG99] Alfons Geser and Sergei Gorlatch. Parallelizing functional programs by generalization. *Journal of Functional Programming*, 9(6):649–673, 1999.

[GGL06] Armin Größlinger, Martin Griebl, and Christian Lengauer. Quantifier elimination in automatic loop parallelization. *Journal of Symbolic Computation*, 41(11):1206–1221, 2006.

[Gib95] Jeremy Gibbons. An initial-algebra approach to directed acyclic graphs. In *Mathematics of Program Construction, MPC 1995, Kloster Irsee, Germany, July 17-21, 1995, Proceedings*, volume 947 of *Lecture Notes in Computer Science*, pages 282–303. Springer, 1995.

[Gib96] Jeremy Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657–665, 1996.

[Gib07] Jeremy Gibbons. Metamorphisms: Streaming representation-changers. *Science of Computer Programming*, 65(2):108–139, 2007.

[GMS04] Robert Giegerich, Carsten Meyer, and Peter Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.

[Gor96] Sergei Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In *Programming Languages: Implementations, Logics, and Programs, 8th International Symposium, PLILP'96, Aachen, Germany, September 24-27, 1996, Proceedings*, volume 1140 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 1996.

[GR89] Alan Gibbons and Wojciech Rytter. Optimal parallel algorithm for dynamic expression evaluation and context-free recognition. *Information and Computation*, 81(1):32–45, 1989.

[Hel89] Paul Helman. A common schema for dynamic programming and branch and bound algorithms. *Journal of the ACM*, 36(1):97–128, 1989.

[HIT97] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.

[HITT97] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming, ICFP'97, Amsterdam, The Netherlands, June 9-11, 1997*, pages 164–175. ACM, 1997.

[HMS93] Paul Helman, Bernard M. E. Moret, and Henry D. Shapiro. An exact characterization of greedy structures. *SIAM Journal on Discrete Mathematics*, 6(2):274–283, 1993.

[HTI99] Zhenjiang Hu, Masato Takeichi, and Hideya Iwasaki. Diffusion: Calculating efficient parallel programs. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, January 22-23, 1999*, pages 85–94. ACM, 1999.

[Hue97] Gérard P. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[Iba73] Toshihide Ibaraki. Solvable classes of discrete dynamic programming. *Journal of Mathematical Analysis and Applications*, 43(3):642–693, 1973.

[Iba74] Toshihide Ibaraki. Classes of discrete optimization problems and their decision problems. *Journal of Computer and System Science*, 8(1):84–116, 1974.

[Iba78] Toshihide Ibaraki. Branch-and-bound procedure and state-space representation of combinatorial optimization problems. *Information and Control*, 36(1):1–27, 1978.

[ITNH08] Dai Ikarashi, Yoshinori Tanabe, Koki Nishizawa, and Masami Hagiya. Modal μ-calculus on min-plus algebra $\mathbb{N}_\infty$. In *The 10th JSSST Workshop on Programming and Programming Languages, PPL2007, proceedings*, pages 216–230, 2008.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.

[Joh77] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[Jok66] H. C. Joksch. The shortest route problem with constraints. *Journal of Mathematical Analysis and Applications*, 14(2):191–197, 1966.

[Jon96] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.

[KH67] Richard M. Karp and Michael Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):693–718, 1967.

[Kit08] Nanao Kita. 最適経路探索アルゴリズムの自動導出に関する研究 (Automatic derivation of optimal path querying algorithms). Bachelor's thesis, Faculty of Engineering, University of Tokyo, 2008. (In Japanese).

[KK01] Turgay Korkmaz and Marwan Krunz. Multi-constrained optimal path selection. In *Proceedings IEEE INFOCOM 2001, The Conference on Computer Communications, Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies 22-26 April 2001, Anchorage, Alaska, USA*, volume 2, pages 834–843. IEEE, 2001.

[KL81] Bernhard Korte and László Lovász. Mathematical structures underlying greedy algorithms. In *Proceedings of the 1981 International FCT-Conference on Fundamentals of Computation Theory, FCT'81*, pages 205–209. Springer, 1981.

[KL95] David J. King and John Launchbury. Structuring depth-first search algorithms in Haskell. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 344–354. ACM, 1995.

[KME07] Kazuhiko Kakehi, Kiminori Matsuzaki, and Kento Emoto. Efficient parallel tree reductions on distributed memory environments. In *Computational Science - ICCS 2007, 7th International Conference, Beijing, China, May 27 - 30, 2007, Proceedings, Part II*, volume 4488 of *Lecture Notes in Computer Science*, pages 601–608. Springer, 2007.

[KT05] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[KV06] Jevgeni Kabanov and Varmo Vene. Recursion schemes for dynamic programming. In *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings*, volume 4014 of *Lecture Notes in Computer Science*, pages 235–252. Springer, 2006.

[LD94] Sivaramakrishnan Lakshmivarahan and Sudarshan K. Dhall. *Parallel computing using the prefix problem*. Oxford University Press, Inc., 1994.

[Lov76] David B. Loveman. Program improvement by source to source transformation. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, January 1976*, pages 140–152. ACM, 1976.

[LRY+04] Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 219–230. ACM, 2004.

[LS03] Yanhong A. Liu and Scott D. Stoller. Dynamic programming via static incrementalization. *Higher-Order and Symbolic Computation*, 16(1–2):37–62, 2003.

[LW93] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, 1993.

[LY02] Yanhong A. Liu and Fuxiang Yu. Solving regular path queries. In *Mathematics of Program Construction, 6th International Conference, MPC 2002, Dagstuhl Castle, Germany, July 8-10, 2002, Proceedings*, volume 2386 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2002.

[Mar84] Ernesto Q. Vieira Martins. An algorithm for ranking paths that may contain cycles. *European Journal of Operational Research*, 18(1):123–130, 1984.

[Mat07a] Kiminori Matsuzaki. Efficient implementation of tree accumulations on distributed-memory parallel computers. In *Computational Science - ICCS 2007, 7th International Conference, Beijing, China, May 27 - 30, 2007, Proceedings, Part II*, volume 4488 of *Lecture Notes in Computer Science*, pages 609–616. Springer, 2007.

[Mat07b] Kiminori Matsuzaki. *Parallel Programming with Tree Skeletons*. PhD thesis, Graduate School of Information Science and Technology, The University of Tokyo, 2007.

[McB01] Conor McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001.

[Mei92] Erik Meijer. *Calculating Compilers*. PhD thesis, Nijmegen University, 1992.

[MFP91] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

[MHKT05] Kiminori Matsuzaki, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi. Systematic derivation of tree contraction algorithms. *Parallel Processing Letters*, 15(3):321–336, 2005.

[MHT03] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallelization with tree skeletons. In *Euro-Par 2003, Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003, Proceedings*, volume 2790 of *Lecture Notes in Computer Science*, pages 789–798. Springer, 2003.

[MHT06] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Towards automatic parallelization of tree reductions in dynamic programming. In *SPAA 2006: Proceedings of the 18th Annual ACM Symposium on Parallel Algorithms and Architectures, Cambridge, Massachusetts, USA, July 30 - August 2, 2006*, pages 39–48. ACM, 2006.

[MHT07] Akimasa Morihata, Zhenjiang Hu, and Masato Takeichi. プログラム運算によるグラフアルゴリズムの導出 (Deriving graph algorithms based on program calculation). In *the 9th JSSST Workshop on Programming and Programming Languages, PPL2007, proceedings*, pages 201–215, 2007. (In Japanese).

[MIEH06] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming. In *Proceedings of the 1st International Conference on Scalable Information Systems, Infoscale 2006, Hong Kong, May 30-June 1, 2006*, volume 152 of *ACM International Conference Proceeding Series*, page 13. ACM, 2006.

[MKHT06] Akimasa Morihata, Kazuhiko Kakehi, Zhenjiang Hu, and Masato Takeichi. Swapping arguments and results of recursive functions. In *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings*, volume 4014 of *Lecture Notes in Computer Science*, pages 379–396. Springer, 2006.

[MM08] Akimasa Morihata and Kiminori Matsuzaki. A tree contraction algorithm on non-binary trees. *Technical Report METR* 2008-27, Department of Mathematical Informatics, University of Tokyo, June 2008.

[MMHT07] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Calculus of minimals: Deriving dynamic-programming algorithms based on preservation of monotonicity. *Technical Report METR* 2007-61, Department of Mathematical Informatics, University of Tokyo, 2007.

[MMHT08a] Akimasa Morihata, Kiminori Matsuzaki, and Masato Takeichi. Write it recursively: A generic framework for optimal path queries. In *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, Sept. 22-24, 2008, Victoria, BC, Canada*, pages 169–178. ACM, 2008.

[MMHT08b] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 並列プログラムの候補生成と適合性検査による並列化 (Program parallelization by candidate generation and conformity testing). *IPSJ Transaction on Programming.* (In Japanese). Accepted.

[MMHT09] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. The third homomorphism theorem on trees: Downward & upward lead to divide-and-conquer. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, Georgia, USA, January 21-23, 2009*, pages 177–185. ACM, 2009.

[MMM+07] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 146–155. ACM, 2007.

[Mor82] Thomas L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982.

[Mor07] Kazutaka Morita. 右逆関数の自動導出によるプログラムの並列化に関する研究 (Program parallelization based on automatic right inversion). Master's thesis, Graduate School of Information Science and Technology, University of Tokyo, 2007. (In Japanese).

[MPRS99] Ernesto Q. Vieira Martins, Marta Margarida B. Pascoal, Deolinda Maria L. Dias Rasteiro, and Jose Luis E. Santos. The optimal path problem. *Investigação Operacional*, 19:43–69, 1999.

[MR85] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer*

Science, 21-23 October 1985, Portland, Oregon, USA, pages 478–489. IEEE, 1985.

[MR93] Bernhard Möller and Martin Russling. Shorter paths to graph algorithms. In *Mathematics of Program Construction, Second International Conference, Oxford, U.K., June 29 - July 3, 1992, Proceedings*, volume 669 of *Lecture Notes in Computer Science*, pages 250–268. Springer, 1993.

[MW95] Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM Jornal on Computing*, 24(6):1235–1258, 1995.

[MW97] Ernst W. Mayr and Ralph Werchner. Optimal tree constraction and term matching on the hypercube and related networks. *Algorithmica*, 18(3):445–460, 1997.

[OHS03] Mizuhito Ogawa, Zhenjiang Hu, and Isao Sasano. Iterative-free program analysis. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP'03, Uppsala, Sweden, August 25-29, 2003*, pages 111–123. ACM, 2003.

[Pai83] Robert Paige. Transformational programming - applications to algorithms and systems. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, January 1983*, pages 73–87. ACM, 1983.

[Pey03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[Pot98] William M. Pottenger. The role of associativity and commutativity in the detection and transformation of loop-level parallelism. In *ICS'98, Proceedings of the 1998 International Conference on Supercomputing, July 13-17, 1998, Melbourne, Australia*, pages 188–195. ACM, 1998.

[PP96] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[Pun91] Abraham P. Punnen. A linear time algorithm for the maximum capacity path problem. *European Journal of Operational Research*, 53(3):402–404, 1991.

[PY97] Robert Paige and Zhe Yang. High level reading and data structure compilation. In *Conference Record of POPL'97: The 24th ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997, pages 456–469. ACM, 1997.

[Rav99a] Jesús N. Ravelo. *Relations, Graphs and Programs*. PhD thesis, Oxford University Computing Laboratory, 1999. *Technical Monograph PRG-125*.

[Rav99b] Jesús N. Ravelo. Two graph algorithms derived. *Acta Informatica*, 36(6):489–510, 1999.

[Rei93] John H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.

[Rei07] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., 2007.

[RF93] Xavier Redon and Paul Feautrier. Detection of recurrences in sequential programs with loops. In *PARLE'93, Parallel Architectures and Languages Europe, 5th International PARLE Conference, Munich, Germany, June 14-17, 1993, Proceedings*, volume 694 of *Lecture Notes in Computer Science*, pages 132–145. Springer, 1993.

[Rom88] Jean-François Romeuf. Shortest path under rational constraint. *Information Processing Letters*, 28(5):245–248, 1988.

[SdM01] Ganesh Sittampalam and Oege de Moor. Higher-order pattern matching for automatically applying fusion transformations. In *Proceedings of the Second Symposium of Programs as Data Objects, PADO'01*, volume 2053 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2001.

[SGJ94] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, U.K., April 11-13, 1994, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500. Springer, 1994.

[SHT98] Isao Sasano, Zhenjiang Hu, and Masato Takeichi. 構成的手法によるグラフアルゴリズムの導出 (Calculational derivation of graph algorithms). In *Proceedings of the 15th annual conference of Japan Society for Software Science and Technology*, pages 269–272, 1998. (In Japanese).

[SHT00] Isao Sasano, Zhenjiang Hu, and Masato Takeichi. グラフの探索関数の再帰的定義と変換 (A general recursive form of graph traversals and its transformation). *Computer Software*, 17(3):2–19, 2000. (In Japanese).

[SHTO00] Isao Sasano, Zhenjiang Hu, Masato Takeichi, and Mizuhito Ogawa. Make it practical: a generic linear-time algorithm for solving maximum-weightsum problems. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), Montreal, Canada, September 18-21, 2000*, pages 137–149. ACM, 2000.

[SJH06] Hanif D. Sherali, Chawalit Jeenanunta, and Antoine G. Hobeika. The approach-dependent, time-dependent, label-constrained shortest path problem. *Networks*, 48(2):57–67, 2006.

[Ski96] David B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributied Computing*, 39(2):115–125, 1996.

[SKN96] Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *ICS'96, Proceedings of the 1996 International Conference on Supercomputing, May 25-28, 1996, Philadelphia, PA, USA*, pages 18–25. ACM, 1996.

[SLL01] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2001.

[Tar51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951.

[VD05] Daniel Villeneuve and Guy Desaulniers. The shortest path problem with forbidden paths. *European Journal of Operational Research*, 165(1):97–107, 2005.

[Wad90] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[Weg76] Ben Wegbreit. Goal-directed program transformation. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, January 1976*, pages 153–170. ACM, 1976.

[Wei88] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1/2):3–27, 1988.

[Whi35] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57:509–533, 1935.

[XKH04] Dana N. Xu, Siau-Cheng Khoo, and Zhenjiang Hu. Ptype system: A featherweight parallelizability detector. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, volume 3302 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2004.

[YHT05] Tetsuo Yokoyama, Zhenjiang Hu, and Masato Takeichi. Calculation rules for warming-up in fusion transformation. In *the 2005 Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*, pages 399–412, 2005.

[Yok06] Tetsuo Yokoyama. *Deterministic Higher-order Matching for Program Transformation.* PhD thesis, Graduate School of Information Science and Technology, The University of Tokyo, 2006.