Generation of Efficient Algorithms for Maximum Marking Problems

(和訳: 最大マーク付け問題の効率的解法の自動生成)

Isao Sasano
Acknowledgments

First I would like to thank to Prof. Masato Takeichi for giving useful comments to this work and letting me do this work freely. Second I would like to thank to Zhenjiang Hu for giving useful comments to this work and having several discussions. Third I would like to thank to Mizuhito Ogawa for letting me know interesting papers about graph algorithms and having several meetings on this work, and to Akihiko Takano for giving useful comments and encouraging me.

This work owes much to the thoughtful and helpful discussion comments made by Hideya Iwasaki, and other Tokyo CACA members. Thanks to Jeremy Gibbons for his valuable comments on an earlier version of our paper accepted to ICFP 2000, to Oege de Moor for providing us several interesting optimization problems, to Richard Bird for kindly sending us his draft about deriving linear time algorithm for maximum marking problems using relational calculus, to Shinichi Morishita for introducing us the problems of data mining.
# Contents

## 1 Introduction
- 1.1 Optimization Problems ........................................... 1
- 1.2 Program Derivation .............................................. 2
- 1.3 Maximum Marking Problems ...................................... 4
- 1.4 Existing Approaches ............................................. 5
- 1.5 Program Generation ............................................. 6
- 1.6 Our proposing method ........................................... 7
- 1.7 A Tour ............................................................. 8
  - 1.7.1 Specification ................................................ 8
  - 1.7.2 Derivation .................................................. 10
  - 1.7.3 Remarks .................................................... 12
- 1.8 Organization of this thesis ...................................... 13

## 2 Preliminaries ..................................................... 14
- 2.1 Recursive Data Types ........................................... 14
- 2.2 Catamorphism .................................................... 15
- 2.3 Mutumorphisms .................................................. 16
- 2.4 Notation ........................................................... 16

## 3 Maximum Marking Problems ..................................... 18
- 3.1 A Formal Definition .............................................. 18
  - 3.1.1 Marking .................................................... 19
  - 3.1.2 Weight ..................................................... 20
  - 3.1.3 Maximum Weightsum Problem ............................. 21
- 3.2 Examples .......................................................... 21
  - 3.2.1 Tree Knapsack Problem .................................... 22
  - 3.2.2 r-MSS Problem ............................................ 22
  - 3.2.3 Party Planning Problem ................................... 22
3.2.4 Coloring Problem ........................................... 23
3.3 Existing Solution — Borie’s Approach ......................... 23

4 Derivation of Efficient Algorithms ................................. 25
  4.1 A Sufficient Condition about Property ......................... 25
  4.2 A Sufficient Condition about Weight Function ............... 31
     4.2.1 Homomorphic Weight Function ............................ 32
     4.2.2 Generalized Optimization Lemma .......................... 32
  4.3 Decomposition Lemma ........................................... 33
     4.3.1 Property with Boolean Functions ......................... 34
     4.3.2 Property with Finite-range Functions ..................... 34
  4.4 The Main Theorem ............................................. 35
  4.5 Extension of property ......................................... 36
     4.5.1 Finite Accumulative Property .............................. 37
     4.5.2 Theorem .................................................. 38
  4.6 Calculation Strategy ........................................... 40

5 Examples .................................................................. 47
  5.1 Paragraph Formatting Problem .................................. 47
  5.2 Security Van Problem ............................................. 49
  5.3 Knapsack Problem ............................................... 51
     5.3.1 Specification ............................................... 51
     5.3.2 Property by Finite Mutumorphisms ....................... 52
     5.3.3 Property by Finite Accumulative Property ............. 52
     5.3.4 Weight function .......................................... 53
     5.3.5 Derivation by Theorem 4 ................................. 53
     5.3.6 Derivation by Theorem 3 ................................. 53
  5.4 Knapsack Problem on List ....................................... 54
     5.4.1 Specification ............................................... 54
     5.4.2 Derivation .................................................. 54
  5.5 Tree Knapsack Problem .......................................... 55
     5.5.1 Specification ............................................... 55
     5.5.2 Derivation .................................................. 57
  5.6 0-1 Multiple Knapsack Problem ................................ 57
     5.6.1 Specification ............................................... 57
     5.6.2 Derivation .................................................. 58
  5.7 Multidimensional Knapsack Problem ............................ 58
     5.7.1 Specification ............................................... 59
Abstract

In existing work on graph algorithms, it is known that a linear time algorithm can be derived mechanically from a logical formula for a class of optimization problems. But this has a serious problem that the derived algorithm has huge constant factor. In this work, we redefine this problem on recursive data structures as a maximum marking problem and propose method for deriving a linear time algorithm for that. In this method, specification is given using recursive functions instead of logical formula, which results in a practical linear time algorithm. This method is mechanical and in fact, based on this deriving method, we make a system which automatically generates a practical linear time algorithm from specification for a maximum marking problem.
Chapter 1

Introduction

This thesis proposes an automatic method for deriving linear time algorithms for a class of optimization problems, which we call maximum marking problems. In this chapter firstly we describe what kind of optimization problems maximum marking problems include. Secondly we show why derivation is desired and what kind of methods one can use for deriving efficient algorithms generally. Thirdly we discuss about program generation and mention that our method is appropriate for program generation. Fourthly we describe the contribution of this thesis. Finally we show give an overview of our generation method through an example and then close this chapter by giving the organization of this thesis.

1.1 Optimization Problems

Many problems can be specified in the way ”maximize (or minimize) certain value under some condition”. Such problems are generally called optimization problems. For example, the knapsack problem [MT90] is an example of optimization problems. Other examples include problems in data mining, e.g., optimal range problems [FMMT96a]. Efficient algorithms for optimization problems are needed, so developing efficient algorithms for optimization problems is important task.

Here consider the three examples: the tree knapsack problem, r-MSS problem, and the party planning problem. Developing efficient algorithms for these problems is not so trivial.

Tree Knapsack Problem [dM95] This problem is an extension of the or-
ordinary knapsack problem [MT90] to one on tree. Input of the ordinary knapsack problem is a set of items each of which has weight and value. Output is a feasible selection of items whose value sum is maximum in all the feasible item selections. A selection is feasible when sum of weight of selected items does not exceed the given capacity $C$. Tree knapsack problem is an extended version of the ordinary knapsack problem so that items are arranged in a tree structure and selected items are connected, that is, a set of items induces a connected subgraph of the input given tree. We assume weight of items are integers.

**r-MSS Problem [BRS99]** This problem is an extension of the maximum segment sum problem, which is a famous problem in program derivation field [Bir89, Gri90]. This problem appears in a problem of data mining, optimum range association rules problem [BRS99]. Input is a sequence of numbers. Output is a feasible selection of elements whose sum is maximum in all the feasible element selections. A selection is feasible when selected elements form up to $r$ connected subsequences. When $r = 1$, it is the ordinary maximum segment sum problem.

**Party Planning Problem [CLRS01]** Professor McKenzie is consulting for the president of the A.-B Corporation, a company that has a hierarchical structure. That is, the supervisory relations form a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating that is a positive or negative real number. The president wants to have a company party. To make the party fun for all attendees, the president does not want both an employee and his or her direct supervisor to attend. The problem is to design a linear algorithm making the guest list. The goal is to maximize the sum of the conviviality ratings of the guests.

How can we develop efficient algorithms for these problems? Using our proposing method, one can obtain efficient linear algorithms for these problems by a mechanical way and can obtain them fully automatically by using our system for MMP.

### 1.2 Program Derivation

Consider the examples we showed in the previous section. When efficient algorithms are asked for, usually people first think about how to solve them
and then write efficient programs by hand. This is ordinary way which is generally taken. But this ordinary method is not satisfactory because correctness of algorithms is not guaranteed. Developing a correct algorithm is important because if an algorithm is not correct, then anyone doesn’t want to use it. If an algorithm is used for an important purpose, then correctness is crucial. So, guaranteeing correctness of algorithm is needed.

But generally speaking, writing an correct and efficient program by hand directly is difficult task. In many cases, after writing a program, one have to repair syntax error or semantic error. In some cases, removing bugs from a program takes longer time than writing the program. Furthermore, specification of a problem may be changed in the future. If it is changed, then modifying suitable part of the program is difficult if an other programmer than wrote the original program has to modify it. Even the programmer who wrote the original one feels difficulty in it if it has passed long time since writing the original one.

One way to remedy those problems is to use program derivation: deriving an efficient program from specification [PP96]. The task is to obtain an efficient program from specification. To guarantee that the obtained program satisfies specification, it is only required that each derivation step should be correct. Using program derivation, correctness and efficiency are obtained at the same time. Furthermore, when the problem is changed slightly, then one only has to change the specification slightly and do a similar derivation in the case it is possible.

Specification is written in some language such as logical formula, function, relation, and so on. In this thesis, we use specification written in function. Derivation process consists of several transformation steps. A transformation step is a transformation that transforms a function to another function, which preserves the meaning. A feature of functional specification is that specification itself can be executed. Programmer first writes a program which may be inefficient but is guaranteed correct, which can be considered as a specification. Here, efficiency of a function means the number of reduction steps for evaluating the function when implementing the function in some functional language such as Haskell [PJH99, Bir98], ML [MTHM97, Pau96] or Scheme [ADH+98, AwJS96]. Program derivation is performed by applying certain transformation rule to the function in each stage and finally an efficient function will be obtained. Generally program derivation is not done mechanically since there are several possible selections in each stage.
1.3 Maximum Marking Problems

We concentrate on a class of optimization problems, which we call ”maximum marking problems” (MMP for short). The MMP can be specified as follows: Given a data structure $x$, the task is to find a way to mark some elements in $x$ such that the marked data structure of $x$ satisfies a certain property $p$ and has the maximum value with respect to certain weight function $w$. This means that no other marking of $x$ satisfying $p$ can produce a larger value with respect to $w$.

By restricting the class of optimization problems to MMP, it becomes possible to derive mechanically a linear time algorithm for solving it from simple specifications. Though of course it is not expected to derive a linear one for every MMP problem since MMP includes NP-hard problems, we can derive a linear one mechanically if property $p$ and weight function $w$ satisfies certain condition. Our method is mechanical, so we can implement it as a system which automatically generates linear time algorithms for MMP.

MMP not only is suitable for automatic generation but also is general enough to express wide range of optimization problems. So we can say that MMP is an appropriate class of problems in that sense.

Originally MMP was considered in graph algorithms in less general form: to find a subset of the vertex set which satisfies certain property and has maximum sum of weight of vertices [BLW87]. This is a sub-class of MMP, which we call ”maximum weightsum problems” [SHTO00, SHTO01a], where one can only allowed to use the $wsum$, which computes sum of selected elements, as the weight function and to use two kinds of marks, True and False, which indicate select and non-select respectively. This sub-class includes many graph problems and a sufficient condition of property $p$ for linearity was given by [BLW87].

By MMP, we can express various kinds of problems which include the problems we show in Section 1.1. Though those examples seems a bit different, they can be formulated in the uniform way: Given a data $x$, find a feasible selection of elements whose weight is maximum in all the feasible selection. By expressing feasibility using a predicate $p$ and giving weight using a weight function $w$, we can write the specification as follows:

$$\uparrow w / \circ \text{filter } p \circ \text{gen } [\text{True, False}]$$

Here $\text{gen}$ is used for generating all the way of selecting elements with marks True and False. Selection is expressed by marking selected elements and un-
marking non-selected elements, where \textit{True} corresponds to "mark" and \textit{False} corresponds to "unmark". The operator $\uparrow_f$ is called the selection operator \cite{Bir87} and is defined by

$$a \uparrow_f b = \begin{cases} b, & \text{if } f a \leq f b \\ a, & \text{otherwise} \end{cases}$$

In this definition, the value of $a \uparrow_f b$ is $b$ when $f a = f b$. The operator $/$ is called the reduce operator \cite{Bir87}, which takes an associative binary operator and a list, defined as follows:

$$\oplus/[a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

The differences between the three problems are in the different definitions of the \textit{property description} of $p$.

The properties for the tree knapsack problem, $r$-MSS problem, the party planning problem, respectively denoted by $p_{tk}$, $p_{rmss}$, and $p_{pp}$ are as follows:

- $p_{tk}$: sum of weight of selected items does not exceed the given capacity $C$ and selected items are connected in the input tree.
- $p_{rmss}$: selected elements form up to $r$ connected subsequences in the input sequence.
- $p_{pp}$: any two selected elements does not have direct parent-child relationship.

We shall refer to these kinds of problems, the main subject of this thesis, as \textit{maximum marking problems}. In general, a maximum marking problem is to find a way of feasible marking the input data that maximize the value of the given weight function $w$, where feasibility is given by certain property $p$.

### 1.4 Existing Approaches

The maximum marking problems are interesting in that they encompass a very large class of optimization problems \cite{BLW87, BPT92}. Solving maximum marking problems, however, requires much insightful analysis. There are basically two kinds of approaches.
• **The Algebraic Approach.**

Using the algebraic laws of programs [Bir89, BdM96], one may try to calculate efficient solutions to the problems by program transformation. For instance, Bird derived a linear algorithm to solve the maximum segment sum problem [Bir89], which is a maximum marking problem on lists. Bird and de Moor [BdM96] demonstrated the derivation of a greedy linear functional algorithm for the party planning problem based on the relational calculus.

However, the success of derivation usually depends on a powerful calculation theorem and requires careful and insightful justification to meet the conditions of the theorem. For many cases, such justification is difficult for (even experienced) functional programmers to mimic to solve other similar problems.

• **The Construction-from-predicates Approach.**

Though little known in functional programming community, it has been known for decades [AP89, Wim87, BLW87, BPT92] that if maximum weightsum problems are specified by regular predicates [BPT92], they are solvable in linear time on decomposable graphs and that linear-time algorithms for them can be derived mechanically from the specifications of the graph problems.

Though more systematic and constructive than the algebraic approach, this approach yields algorithms that suffer from a prohibitively large table (see Section 3.3 and Chapter 7 for details). The algorithm for solving the party planning problem, for instance, would need to construct a table with more than $2^{2^{142}}$ entries [BPT92]. The algorithms thus cannot be put to practical use.

### 1.5 Program Generation

Program generation has seen an important role in a wide range of software development processes. A successful program generation system requires not only a powerful language supporting coding of program generation, but also a set of effective transformation rules for the generation of programs. An example, which convincingly shows the importance of the design of effective transformation rules, is the well-known *fold-build* rule [GLP93] for fus-
ing composition of functions in Glasgow Haskell Compiler (GHC). It is this general, concise and cheap calculation rule that makes it possible for GHC to practically generate from large-scale programs efficient programs without unnecessary intermediate data structures. Generally, the effective rules for program generation should meet several requirements.

- First, they should be general enough to be applied to a program pattern, by which a useful class of problems can be concisely specified.
- Second, they should be abstract enough to capture a big step of the program generating process rather than being a set of small rewriting rules.
- Third, they can be efficiently implemented by program generation systems.

In this thesis we shall propose such a rule for generating efficient programs from the following program pattern

$$mmp \ w \ p \ ms = \uparrow_w / \circ \ \text{filter} \ p \ \circ \ \text{gen} \ ms,$$

which is the specification of MMP. Our proposing method is a mechanical method that derives a linear time algorithm for MMP from the above program pattern. So our method is appropriate for automatic program generation from specification, and actually we implement our method as an automatic program generation system for MMP in Chapter 6.

1.6 Our proposing method

In this thesis we propose a new approach to deriving practical linear-time algorithms for maximum marking problems over data structures such as lists, trees, and decomposable graphs. The key points of our approach are to express the property $p$ by recursive boolean functions over the structure $x$ rather than a usual logical predicate and to apply program transformation techniques to reduce the constant factor, thereby exploiting the advantages of the algebraic and construction-from-predicates approaches.

Our main contributions can be summarized as follows.

7
• We propose an optimization theorem that gives a generic and practical linear-time algorithms for solving maximum marking problems (Chapter 4). By using the optimization theorem, we can derive practical linear-time algorithms for various kinds of problems which include real-world problems such as knapsack problems, optimal range problems [FMMT96a] in the data mining, and many problems in Bird et al.’s textbook [BdM96], as partly shown in Chapter 5.

• Our method is simple, general, and flexible. We demonstrate this in Chapter 5 by deriving linear-time algorithms for various kinds of interesting and nontrivial maximum marking problems. The Haskell codes for solving some problems in this thesis are available at http://www.ipl.t.u-tokyo.ac.jp/~sasano/mws.html.

• We are the first to successfully apply the algebraic approach to solving the huge-table problem appearing in the derivation of linear-time algorithms on decomposable graphs [AP89, Wim87, BLW87, BPT92], a problem not solved by table compression [BLW87] or by dynamic table management [APT00]. This should be a significant step in making the theoretically appealing linear-time graph algorithms practically useful.

• We show that our proposing method can be implemented as an automatic program generation system for MMP (Chapter 6). Our method can be implemented using the existing transformation systems like MAG [dMS98], and efficient programs can be obtained in a fully automatic way. We also implement a flexible system called Yicho, in which we can specify transformation strategies.

1.7 A Tour

Here we briefly explain our idea by going through the list version of the party planning problem.

1.7.1 Specification

The list version of the party planning problem, which will be called the maximum independent-sublist sum problem (mis for short), is to compute \( \text{vs} \), the set of elements from a non-empty list \( xs \), such that no two elements in
mis :: [Elem] -> [MElem]
mis xs = let opts = mis' xs
    in getdata (foldr1 (bmax second)
        [ (c,w,cand)
          | (c,w,cand) <- opts,
          c==2 || c==3])

mis' :: [Elem] -> [(Class,Weight,[MElem])]
mis' [x] = [(2,x,[x,True]), (3,0,[(x,False)])]
mis' (x:xs) =
    let opts = mis' xs
    in eachmax 
        [ (table (marked mx) c,
           (if marked mx then weight mx else 0) + w,
           mx:cand)
          | mx <- [mark x, unmark x],
          (c,w,cand) <- opts]

bmax f a b = if f a > f b then a else b

eachmax xs = foldl f [] xs
    where f [] (c,w,cand) = [(c,w,cand)]
         f ((c,w,cand) : opts) (c',w',cand') =
             if c==c' then
                 if w'>w' then (c,w,cand) : opts
                 else opts ++ [(c',w',cand')]  
             else (c,w,cand):f opts (c',w',cand')

Figure 1.1: A linear-time Haskell program for the mis problem.
vs are adjacent in xs. Clearly, it is one of the maximum marking problems on lists, which can be specified by

\[
\text{mis} :: [\alpha] \rightarrow [\alpha] \\
\text{mis} \; xs = \uparrow_{ws} / [vs | vs <- \text{subs} \; xs, p_{mis} (xs; vs)]
\]

where \text{subs} enumerates all sublists (not necessarily contiguous) of a list.

What is left is to define \( p_{mis} \), the specific component of the problem. Because \( vs \) is a sublist of \( xs \), we can specify \( p_{mis} \) in two steps: first marking the elements in \( xs \) which belong to \( vs \), and then on the marked \( xs \) defining the property. Thus,

\[
p_{mis} (xs; vs) = p \left( \text{marking} \; xs \; vs \right).
\]

Here \( p \) checks that all pairs of marked elements are not adjacent in the marked \( xs \).

\[
p :: [\alpha] \rightarrow \text{Bool} \\
p \; [x] = \text{True} \\
p \; (x : xs) = \begin{cases} 
\text{if} \ \text{marked} \; x \\
\text{then} \ \text{not} \ \left( \text{marked} \ (\text{hd} \; xs) \right) \land p \; xs \\
\text{else} \ p \; xs
\end{cases}
\]

\[
\text{hd} :: [\alpha] \rightarrow \alpha \\
\text{hd} \; [x] = x \\
\text{hd} \; (x : xs) = x
\]

For later transformation, we define \( \text{hd} \) in the same way as \( p \) over the two cases singleton list and cons list. So much for our specification of the problem. It is worth noting that our specification is as natural as that in [BPT92] using the monadic second-order logic (See Section 3.3). The critical difference is in specifying \( p \) by using recursive functions instead of a logical predicate. This enables us to use the functional program calculation for the later derivation.

### 1.7.2 Derivation

The derivation is based on our optimization theorem in Chapter 4, which says that if the property description \( p \) can be defined in mutumorphisms [Fok89, Jeu93, HITT97], then a linear-time algorithm solving the maximum marking problem with respect to \( p \) can be derived mechanically.
Therefore, the derivation of a linear-time algorithm for the mis problem reduces to be a derivation of mutumorphisms for \( p \). More precisely, we hope to transform \( p \) to the following form:

\[
p [x] = \phi_1 x \\
p (x : xs) = \phi_2 x (p \; xs, p_1 \; xs, \ldots, p_n \; xs)
\]

where \( \phi_i \)'s denote some functions and \( p_i \)'s are auxiliary property descriptions defined in a fashion similar to that in which \( p \) is defined:

\[
p_i :: [\alpha] \rightarrow \text{Bool} \\
p_i [x] = \phi_{i1} x \\
p_i (x : xs) = \phi_{i2} x (p \; xs, p_1 \; xs, \ldots, p_n \; xs).
\]

Consider now the \( p \) for the mis problem. By introducing \( p_1 \) defined by

\[
p_1 :: [\alpha] \rightarrow \text{Bool} \\
p_1 [x] = \text{not} (\text{marked} \; x) \\
p_1 (x : xs) = \text{not} (\text{marked} \; x)
\]

we can transform \( p \) to the following form.

\[
p [x] = \text{True} \\
p (x : xs) = \text{if} \; \text{marked} \; x \; \text{then} \; p_1 \; xs \; \text{\&} \; p \; xs \; \text{else} \; p \; xs
\]

Applying our optimization theorem for solving maximum marking problems now soon yields a linear-time algorithm like that in Figure 1.1, where the algorithm is coded in Haskell. The function \( \text{bmax} \) in Figure 1.1 corresponds to the selection operator \( \uparrow \). The main function \( \text{mis} \) takes a list as its argument and returns the input list with the selected elements marked with \( \text{True} \). For example, \( \text{mis} \; [1..4] \) returns

\[
[(1,\text{False}),(2,\text{True}),(3,\text{False}),(4,\text{True})].
\]

Note that our initial specification only returns \([2,4]\). The correctness of the algorithm follows from our optimization theorem (see Chapter 4). The linear property for \( \text{mis} \) comes from the following observation:

- The argument to \( \text{eachmax} \) has at most 8 elements, and so does the second argument of \( f \) used to define \( \text{eachmax} \). So \( \text{eachmax} \) costs \( O(1) \) time. Therefore, the auxiliary function \( \text{mis}' \) is a linear-time program.
The *opts* in the body of *mis* has at most 4 elements, so it costs constant time to produce the final result after computing *mis’ xs*.

The function *mis’* computes one optimal solution for each class, where classes correspond to elements of range of *h* defined as follows.

\[ h :: [\alpha] \rightarrow (\text{Bool}, \text{Bool}) \]
\[ h x = (p x, p_1 x) \]

Class 0 corresponds to \((False, False)\), Class 1 to \((False, True)\), Class 2 to \((True, False)\), and Class 3 to \((True, True)\). These classes can be interpreted as follows.

- Class 0 means that \(p\) does not hold and \(p_1\) does not hold. This means that the head of the list is marked and the set of marked elements in the list is not independent.

- Class 1 means that \(p\) does not hold and \(p_1\) holds. This means that the head of the list is not marked and the set of marked elements in the list is not independent.

- Class 2 means that \(p\) holds and \(p_1\) does not hold. This means that the head of the list is marked and the set of marked elements in the list is independent.

- Class 3 means that \(p\) holds and \(p_1\) holds. This means that the head of the list is not marked and the set of marked elements in the list is independent.

From the function \(h\), we can automatically derive the definition of *table*. See Chapter 4 for details.

### 1.7.3 Remarks

Two remarks are worth making. First, the optimization theorem, which will be discussed in detail in Chapter 4, plays a significant role in our derivation. To apply this theorem, the only thing one have to do is to find the property description in mutumorphic form.

Second, the property description in mutumorphic form can be derived from a recursive property description by using the calculational strategy we
present in Chapter 4. This derivation utilizes tupling and fusion transformations, which are nothing very special and for which a wealth of calculation techniques have been developed [Jeu93, BdM96]. Our derivation is thus surprisingly simple and powerful.

1.8 Organization of this thesis

The organization of rest part of this thesis is as follows. In Chapter 2, we briefly describe the basic concepts of program calculation such as fusion and tupling, and explain the notation used in this thesis. In Chapter 3, we formally define maximum marking problems. In Chapter 4, we propose the optimization theorem and our calculation framework. In Chapter 6 we show the system which generates a practical linear time algorithms for maximum marking problem, and show its effectiveness by generating linear time algorithms for several examples. In Chapter 7, we discuss related work. In Chapter 8, we make our concluding remarks.
Chapter 2

Preliminaries

In this chapter we briefly review the notational conventions and some basic concepts of program calculation [Bir87, MFP91, BdM96] used in this thesis.

2.1 Recursive Data Types

To simplify the presentation and proof of the optimization theorem in Chapter 4, we restrict ourselves to considering polynomial data types. And to avoid categorical notations, we describe polynomial data types in the following form:

\[ D \alpha = C_1 (\alpha, D_1, \ldots, D_{n_1}) | C_2 (\alpha, D_1, \ldots, D_{n_2}) | \cdots | C_k (\alpha, D_1, \ldots, D_{n_k}) \]

Here \( D_i \)'s denote \( D \alpha \), and \( C_i \)'s are called data constructors applying to an element of type \( \alpha \) and a bounded number of recursive components. Though seemingly restricted, these polynomial data types are powerful enough to cover our commonly used data types, such as lists, binary trees, rooted trees [BLW87], and series-parallel graphs [TNS82]. Moreover, other data types like the rose trees, a kind of regular data type defined by

\[ RTree \alpha = \text{Node } \alpha [ RTree \alpha ] \]

can be encoded into one of these polynomial data types. This will be demonstrated in Chapter 4.
For each data constructor $C_i$, we define $F_i$ by

$$F_i f (e; x_1, \ldots, x_n) = (e; f x_1, \ldots, f x_n).$$

### 2.2 Catamorphism

**Catamorphisms**, one of the most important concepts in program calculation [MFP91, SF93, BdM96], form a class of important recursive functions over a given data type. They are the functions that promote through the data constructors.

For example, for the type of lists, given $e$ and $\oplus$, there exists a unique catamorphism $cata$ satisfying the following equations:

$$cata [] = e$$
$$cata (x : xs) = x \oplus (cata xs)$$

In essence, this solution is a relabeling: it replaces every occurrence of $[]$ with $e$ and every occurrence of $:$ with $\oplus$ in the cons list. Because of the uniqueness property of catamorphisms (i.e., for this example $e$ and $\oplus$ uniquely determines a catamorphism over cons lists), we usually denote this catamorphism as $cata = ([e; (\oplus)])$.

**Definition 1 (Catamorphism)** A catamorphism over a recursive data type $D$ is characterized by

$$f = ([\phi_1, \ldots, \phi_k])_D \equiv f \circ C_i = \phi_i \circ F_i f \ (i = 1, \ldots, k)$$

If it is clear from the context, we usually omit the subscript $D$ in $([\phi_1, \ldots, \phi_k])_D$. □

Catamorphisms play an important role in program transformation (program calculation) because they satisfy a number of nice calculational properties in which the fusion theorem is of greatest importance:

**Theorem 1 (Fusion)**

$$f \circ ([\phi_1, \ldots, \phi_k])_D = ([\psi_1, \ldots, \psi_k])_D$$

provided that for every $i$ with $1 \leq i \leq k$

$$f \circ \phi_i = \psi_i \circ F_i f.$$ □
The fusion theorem gives the condition that has to be satisfied in order to promote (fuse) a function into a catamorphism to obtain a new catamorphism. It actually provides a constructive but powerful mechanism for deriving a “bigger” catamorphism from a program in a compositional style, a typical style for functional programming. When applying the fusion theorem, we may do generalization, which is an operation substituting a new function for the target part of the fusion transformation.

2.3 Mutumorphisms

Mutumorphisms, generalizations of catamorphisms to mutually defined functions, are defined as follows [Fok89, Fok92, HIT97].

Definition 2 (Mutumorphisms) Functions \( f_1, f_2, \ldots, f_n \) are said to be mutumorphisms on a recursive data type \( D \) if each function \( f_i \) is defined mutually by

\[
f_i \circ C_j = \phi_{ij} \circ F_j (f_1 \triangle f_2 \triangle \ldots \triangle f_n)
\]

for \( j \in \{1, 2, \ldots, k\} \).

Note that \( f_1 \triangle f_2 \triangle \ldots \triangle f_n \) represents a function defined as follows:

\[
(f_1 \triangle f_2 \triangle \ldots \triangle f_n) x = (f_1 x, f_2 x, \ldots, f_n x)
\]

It is known that mutumorphisms can be turned into a single catamorphism by the tupling transformation [Fok89, Fok92, HIT97].

Theorem 2 (Mutu Tupling) If \( f_1, f_2, \ldots, f_n \) are mutumorphisms like those in Definition 2, then

\[
f_1 \triangle f_2 \triangle \ldots \triangle f_n = ([\phi_1, \phi_2, \ldots, \phi_k])_D
\]

where \( \phi_i = \phi_{1i} \triangle \ldots \triangle \phi_{ni} \) for \( i = 1, \ldots, k \).

2.4 Notation

Throughout this thesis, we use notation like the functional programming language Haskell [PJH99]. Here, we explain about the list comprehension and the definition of function \( \text{foldl} \) used in, e.g., Figure 4.1.
• List comprehension

List comprehension is a notation for describing a list, which is based on the notation of set used in mathematics. List comprehension takes the following form:

\[ (\{\text{Expression}\} | \{\text{Qualifier}\}, \ldots, \{\text{Qualifier}\}) \]

\(\{\text{Qualifier}\}\) is an expression of boolean or a generator. Generator takes the following form:

\(\{\text{Variable}\} ← \{\text{List}\}\)
\(\{(\text{Variable}), (\text{Variable})\} ← \{\text{List of tuples}\}\)
\(\{(\text{Variable}), (\text{Variable}), (\text{Variable})\} ← \{\text{List of triples}\}\)

\[\cdots\]

For example, \(\{ x \mid x ← [1, 2, 3, 4], x \times x ≤ 10 \}\) is an expression using list comprehension, whose value is a list \([1, 2, 3]\).

• The function foldl

foldl is one of the folding function on lists, which is defined as follows:

\[ \text{foldl} : (\alpha → \beta → \alpha) → \alpha → [\beta] → \alpha \]
\[ \text{foldl} f a [] = a \]
\[ \text{foldl} f a (x : xs) = \text{foldl} f (f a x) xs \]
Chapter 3

Maximum Marking Problems

In this chapter we define the maximum marking problems and clarify the condition that weight should satisfy. At the last of this chapter we clarify the limitations of the existing solutions.

3.1 A Formal Definition

A maximum marking problem can be rephrased as follows. Given a data structure \( x \), the task is to find a way to mark some elements in \( x \) such that the marked data structure of \( x \) satisfies a certain property \( p \) and has the maximum value with respect to certain weight function \( w \). This means that no other marking of \( x \) satisfying \( p \) can produce a larger value with respect to \( w \). A straightforward solution, whose complexity is exponential in the number of elements in \( x \), is as follows:

\[
mmp w p ms = \uparrow_w \circ \text{filter} \circ \text{gen ms}
\]

We use \( \text{gen ms} \) to generate all possible markings of input data \( x \) using a set (list) of marks \( ms \), and from those which satisfy the property \( p \) we use \( \uparrow_w \) to select one whose value with respect to the weight function \( w \) is maximum. Here we assume that there is at least one possible markings of input data \( x \) satisfying the property \( p \).
3.1.1 Marking

Here we describe about marking. We use the following function $mark$ as marking function:

$$mark : [Mark] \rightarrow \alpha \rightarrow [\alpha^*]$$

$$mark \ (ms \ x) = [(x; m) \mid m \leftarrow ms]$$

This function $mark$ takes as its first argument a list $ms$ of all the marks used for marking and as its second argument an element $x$, and returns a list of all the possible marking of the element $x$. $Mark$ is the type of marks. In the simplest case, $Mark = \text{Bool}$. This means that there are two possibilities: mark or unmark. We mean that if the element $x$ is marked with $\text{True}$ then it is marked, and if marked with $\text{False}$ then it is unmarked. In another case, for example, $Mark = \{\text{Red}, \text{Green}, \text{Yellow}\}$. This is used for the coloring problem in Chapter 4. When $Mark$ consists of more than two marks, then we call the problem ”maximum multi-marking problem” in particular.

We use the following function $kind$ to get the mark from a marked element, which is defined as follows:

$$kind : \alpha^* \rightarrow Mark$$

$$kind \ (x, m) = m.$$

We explain some of our notation for marking. For a data type of $\alpha$, we use $\alpha^*$ to extend $\alpha$ with marking information. It can be defined more concretely by

$$\alpha^* = (\alpha, \text{Mark})$$

where a boolean value indicates whether or not the element of type $\alpha$ is marked. Accordingly, we use $a^*, b^*, \ldots, x^*$ to denote variables of the type $\alpha^*$ or the type $D \alpha^*$ (i.e., $D \ (\alpha^*)$).

The function $gen$, exhaustively enumerating all the possible ways of mark-
ing every element, can be recursively defined by

\[
\text{gen :: } [\text{Mark}] \rightarrow D \alpha \rightarrow [D \alpha^*]
\]

\[
\text{gen ms = } ([\eta_1, \ldots, \eta_k])
\]

where

\[
\eta_i (e, xs_1^*, \ldots, xs_n^*) =
\]

\[
C_i (e^*, x_1^*, \ldots, x_n^*) \mid
e^* \leftarrow \text{mark ms } e,
x_1^* \leftarrow xs_1^*,
x_2^* \leftarrow xs_2^*,
\ldots
x_n^* \leftarrow xs_n^* \quad (i = 1 \ldots k)
\]

So much for the specification of \textit{mmp}, with which we can define various maximum marking problems. For example, the \textit{mis} problem in Section 1.7 can be specified by

\[
\text{mis} = \uparrow \text{wsum} / \circ \text{filter } p \circ \text{gen } [\text{True, False}]
\]

where \(p\) is the description of the “independent” property specified in Section 1.7, and the function \textit{wsum} computes the sum of weights of marked elements in a data structure:

\[
\text{wsum :: } D \alpha^* \rightarrow \text{Weight}
\]

\[
\text{wsum } = ([\phi_1, \ldots, \phi_k])
\]

where

\[
\phi_i (e^*, w_1, \ldots, w_n) =
\]

\[
(\text{if } \text{marked } e^* \text{ then weight } e^* \text{ else } 0) +
\]

\[
w_1 + \cdots + w_n
\]

where \textit{marked} is a function which takes as its argument an element \(e^*\) and checks whether or not \(e^*\) is marked.

\[
\text{marked :: } \alpha^* \rightarrow \text{Bool}
\]

\[
\text{marked } (e, m) = m.
\]

This function is only used for the case that \textit{Mark} = \textit{Bool}.

### 3.1.2 Weight

Here we clarify the condition that \textit{weight} should satisfy. In maximum marking problem, what we want is a marked data that has maximum value with
respect to weight function $w$:

$$w : D \alpha^* \to Weight.$$  

As we describe in Chapter 1, a maximum marked data is taken by the following function:

$$\uparrow_w / :: [D \alpha^*] \to D \alpha^*$$

where $\uparrow_f$ is defined by

$$a \uparrow_f b = b, \quad \text{if } f a \leq f b$$

$$= a, \quad \text{otherwise}.$$  

So, the type $Weight$ has to have the relation $\leq$. In order to guarantee that the obtained result by the function $\uparrow_w$ has the maximum value with respect to the weight function $w$, we require that $\leq$ should be total order on $Weight$. So, for example, weight can be integer, real number, natural number, and so on, with the usual total order $\leq$ on each set.

### 3.1.3 Maximum Weightsum Problem

Consider the following simple case of maximum marking problem:

$$mmp wsum p [True, False].$$

Here, $wsum$ is a weight function that computes the sum of elements of marked $True$. We call this kind of problem ”maximum weightsum problem”, which is well studied in [SHTO00]. In the following, we use the function $mws$ to represent the above form:

$$mws p = mmp wsum p [True, False].$$

### 3.2 Examples

Here we show several examples of maximum marking problems, including the problems we showed in Chapter 1 and an example of maximum multi-marking problem called the coloring problem [SHT01].
3.2.1 Tree Knapsack Problem

The input of this problem is a binary tree, which is defined as follows:

\[ \text{Tree } \alpha ::= \text{Leaf } \alpha \mid \text{Bin } (\alpha, \text{Tree } \alpha, \text{Tree } \alpha) \]

The property which should be satisfied can be described by

\[ p t = \text{weightsum } t \leq C \land t \]

where \text{weightsum} takes as its argument a marked tree \( t \) and simply returns the sum of weight of selected items, and \( tc \) also takes as its argument a marked tree \( t \) and checks whether marked elements are connected together. Selection is represented by two marks, \text{True} and \text{False}. Weight function \( w \) should computes the sum of value of selected items. By using these functions, the tree knapsack problem can be specified by

\[ \uparrow w / \circ \text{filter } p \circ \text{gen} [\text{True}, \text{False}] \]

Detail description of these functions are in Chapter 5.

3.2.2 \text{r-MSS} Problem

Input of \text{r-MSS} problem is a sequence of numbers. We represent a sequence by a list. Property which should be satisfied is that selected elements form up to \( r \) connected subsequences in the input sequence. By letting the property \( p \), he \text{r-MSS} problem can be specified as follows:

\[ \uparrow w / \circ \text{filter } p \circ \text{gen} [\text{True}, \text{False}] \]

Here, the weight function \( w \) only computes the sum of selected elements. Detail description of these functions are in Chapter 5.

3.2.3 Party Planning Problem

The input of the party planning problem is a general tree, so we can use the following recursive (regular) data structure.

\[ \text{Org } \alpha ::= \text{Leader } \alpha \mid \text{Org } \alpha \]

This does not belong to recursive data types we defined in Section 2.1, so data type conversion is needed (See section 4). The property which should be
satisfied is that any two selected elements does not have direct parent-child relationship. By letting the property $p$, the party planning problem can be specified as follows:

$$\uparrow w / \circ \text{filter } p \circ \text{gen } [\text{True}, \text{False}].$$

Here, the weight function $w$ only computes the sum of selected elements. Detail description of these functions are in Chapter 4.

3.2.4 Coloring Problem

Here we show an example of maximum multi-marking problems, the coloring problem. Suppose there are three marks: Red, Blue, and Yellow. The problem is to find a way of marking all the elements such that each sort of mark does not appear continuously, and that the sum of the elements marked in Red minus the sum of the elements marked in Blue is maximum.

This problem can be specified as follows:

$$\text{coloring} = \uparrow_w / \circ \text{filter indep} \circ \text{gen } [\text{Red}, \text{Green}, \text{Yellow}]$$
$$\text{indep } xs = \text{indep'} xs 0$$
$$\text{indep'} [] \text{ color } = \text{True}$$
$$\text{indep'} (x : xs) \text{ color } = \text{kind } x \neq \text{color } \land \text{indep'} xs (\text{kind } x)$$

$$w = + / \circ \text{map } f$$
$$\text{where } f \ e^* = \text{case } \text{kind } e^* \text{ of }$$
$$\text{Red } \rightarrow \text{weight } e^*$$
$$\text{Blue } \rightarrow -(\text{weight } e^*)$$
$$\text{Yellow } \rightarrow 0.$$

Of course, this definition is terribly inefficient taking exponential time, though it is straightforward. In the following chapters, we show that they can be automatically transformed to an efficient linear one.

3.3 Existing Solution — Borie’s Approach

The maximum marking problems have many instances in graph algorithms. Bern et al. showed that many graph problems — such as the minimum vertex cover problem, the maximum independent set problem, the maximum
matching problem, and the traveling salesman problem — can be described by \textit{mws}, and, more interestingly, that they can be solved in linear time on decomposable graphs [BLW87]. Basing their work on Bern et al.’s and on Courcelle’s [Cou90b], Borie et al. proposed a method for automatically constructing linear-time algorithms that solve the maximum-weightsum problems [BPT92].

The main idea is to restrict the property \( p \) to be described in terms of a small canonical set of primitive predicates (such as the incident predicate \( \text{Inc}(v,e) \)) or a combination of them by logical operators (\( \land, \lor, \) and \( \neg \)) and (either first-order or second-order) quantifiers (\( \forall \) and \( \exists \)). For example, the property \( p \) for the maximum independent set problem is described by

\[
\begin{align*}
p &= \forall v_1 \forall v_2 \neg \text{Adj}(v_1, v_2) \\
\text{Adj}(v_1, v_2) &= \exists e_1 (\text{Inc}(v_1, e_1) \land \text{Inc}(v_2, e_1)) \\
&\land \neg (v_1 = v_2).
\end{align*}
\]

With this restriction, it is possible to automatically construct a linear-time algorithm for \textit{mws} \( p \) from the predicative structure of \( p \).

Though attractive in theory, a linear-time algorithm needs to create a huge table, and this prevents these algorithms from actually being used [BLW87, BPT92, APT00]. The table created for the maximum independent set problem, for instance, contains more than \( 2^{(2^{142})} \) entries. The functional approach we propose can reduce this number to 8 (See Figure 1.1).
Chapter 4

Derivation of Efficient Algorithms

This chapter focuses on a formal study of our functional approach to solving the maximum marking problems, in which approach our main theorem, the optimization theorem, plays a significant role. It not only clarifies a sufficient condition for the existence of linear-time algorithms, but also gives a calculation rule for the construction of such algorithms. As will be seen later, the key points of our calculation are the functional (rather than predicative) structure of property descriptions, and the good use made of program transformation such as fusion and tupling [HITT97, Fok92, Fok89]. We begin here by giving in Lemma 1 a sufficient condition under which efficient linear-time algorithms are guaranteed to exist. Then we use mutumorphisms to formalize in Lemma 3 the class of problems we can solve in linear time. Finally, we summarize the two lemmas in our optimization theorem.

4.1 A Sufficient Condition about Property

It is obvious that not all optimization problems specified by

\[
\text{spec} :: D \alpha \to D \alpha^* \\
\text{spec} = \text{mws } p
\]

can be solved in linear time. Our first result gives a sufficient condition for \( p \) in the form of the composition of a function and a catamorphism.
Lemma 1 \textbf{(Optimization Lemma)} Let $\text{spec}$ be defined by

\begin{align*}
\text{spec} &::= D \alpha \rightarrow D \alpha^*
\text{spec} &= \text{mws} \left( \text{accept} \circ (\psi_1, \ldots, \psi_k) \right).
\end{align*}

If the domain of the predicate $\text{accept}$ is finite, then $\text{spec}$ can be solved in linear time. \hfill \qed

To prove the lemma, we define an optimization function $\text{opt}$ in Figure 4.1. We will show that $\text{opt}$ solves $\text{spec}$ correctly and that is computable in linear time. In Figure 4.1, $\text{getdata}$ is a function which takes as its argument a triple and returns the third element.

\begin{align*}
\text{getdata} (c, w, r^*) &= r^*
\end{align*}

\textbf{Correctness}

Here we show how the right-hand side of

\begin{align*}
\text{spec} x &= \text{opt accept} \phi_1 \ldots \phi_k x
\end{align*}

Figure 4.1: Optimization function $\text{opt}$. 
is transformed into the left-hand side. In the transformation rules we use the auxiliary functions $\psi'_i (i = 1, \ldots, k)$ defined by

$$
\psi'_i (e, cand_1, \ldots, cand_n) =
\left[ (\phi_i (e^*, c_1, \ldots, c_n)),
\begin{array}{l}
\text{(if marked } e^* \text{ then weight } e^* \text{ else } 0) \\
+ w_1 + \ldots + w_n,
\end{array}
\right.
C_i (e^*, r_1, \ldots, r_n) | \\
e^* \leftarrow \text{mark } e, \text{unmark } e,
(c_1, w_1, r_1) \leftarrow cand_1, \ldots,
(c_n, w_n, r_n) \leftarrow cand_n.
$$
The first transformation rule is simply an unfolding of `opt`.

The second transformation rule is

\[ ([\psi_1, \ldots, \psi_k]) = \text{eachmax} \circ ([\psi_1', \ldots, \psi_k']). \]
The relation between $\psi_i$ and $\psi'_i$ is as follows:

$$\psi_i = \text{eachmax} \circ \psi'_i$$

The function $([\psi_1, \ldots, \psi_k])$ applies the function $\text{eachmax}$ to the list of candidates at each stage. The function $\text{eachmax} \circ ([\psi'_1, \ldots, \psi'_k])$, however, first creates all the markings of input data and corresponding weights and classes. Then it applies the function $\text{eachmax}$. These two functions compute the same candidate solutions because the function $\text{eachmax}$ takes a list which contains candidate solutions and returns a list which consists of the rightmost optimal solution for each class in the input list, preserving the order, and the function $\text{eachmax}$ is idempotent (i.e., $\text{eachmax} \circ \text{eachmax} = \text{eachmax}$).

The third transformation rule

$$\text{filter} (\text{accept} \circ \text{fst}) \circ \text{eachmax} = \text{eachmax} \circ \text{filter} (\text{accept} \circ \text{fst})$$

means the commutativity between $\text{filter}$ and $\text{eachmax}$ functions, where $\text{filter}$ is the abbreviation of

$$\lambda p. \lambda xs. [x | x ← xs, p x].$$

The function $\text{filter} (\text{accept} \circ \text{fst}) \circ \text{eachmax}$ first applies the function $\text{eachmax}$ and then filters. The function $\text{eachmax} \circ \text{filter} (\text{accept} \circ \text{fst})$ first filters and then applies the function $\text{eachmax}$. These two functions compute the same result because the predicate $(\text{accept} \circ \text{fst})$ is concerned only with the classes and because the functions $\text{filter}$ and $\text{eachmax}$ preserve the order.

The fourth transformation rule is

$$\uparrow_{\text{snd}} / \circ \text{eachmax} = \uparrow_{\text{snd}} /.$$  

This equation holds because $\uparrow_{\text{snd}} /$ returns the rightmost optimal solution, and $\text{eachmax}$ returns a list which consists of the rightmost optimal solution for each class in the input list, preserving the order.

The fifth transformation rule is

$$c = ([\phi_1, \ldots, \phi_k]) r^*.$$  

This means that the class to which $r^*$ belongs is $c$, which can be shown by induction.
The sixth transformation rule is
\[ \text{getdata} \circ \uparrow \text{snd} = \uparrow \text{wsum} \circ \text{map getdata}. \]
This holds because the second element is the weightsum of the third element, which can be shown by induction.

The seventh transformation rule
\[ ([\eta_1, \ldots, \eta_k]) = \text{map getdata} \circ ([\psi_1', \ldots, \psi_k']) \]
follows from Theorem 1.

The eighth and ninth transformation rules are simply the foldings of \text{mws} and \text{spec}.

\[ \square \]

**Linearity**

Here we show that the function \text{opt} is linear.

\[
\text{opt accept } \phi_1 \ldots \phi_k \ x = \\
\text{getdata} \ (\uparrow \text{snd} / \\
\ [(c, w, r^*) | (c, w, r^*) \leftarrow ([\psi_1, \ldots, \psi_k])_D \ x, \ \text{accept } c])
\]

The important observation is that the number of elements in the list \([\psi_1, \ldots, \psi_k])_D \ x is bounded by the number of classes\(^1\) (i.e., the number of elements in the domain of \text{accept} or the range of the catamorphism) because each \text{eachmax} returns a list whose length is bounded by the number of classes. Therefore, if \([\psi_1, \ldots, \psi_k])_D \ x can be computed in linear time, so can \text{opt}.

To prove that \([\psi_1, \ldots, \psi_k])_D is a linear-time algorithm, it is suffice to show that \(\psi_i e \text{ cand}_1 \ldots \text{ cand}_n\) can be computed in constant time. Taking the fact that \(\phi_i\) can be computed in \(O(1)\) time because the size of its argument is independent of the input, we can see that the construction of each triple \((c, w, r^*)\) (i.e., each element consumed later by \text{eachmax}) can be computed in \(O(1)\) time because \(c\) can be obtained in \(O(1)\) time by \(\phi_i\), \(w\) by performing the + operation \(n_i\) times, and \(r^*\) simply by combining \(e^*, r_1^*, \ldots, r_n^*\). Here \(n_i\) is bounded by

\[ N = \max \ \{n_i \mid 1 \leq i \leq k\}. \]

\(^1\)From the condition that the domain of \text{accept} is finite, we know the range of the catamorphism \([\phi_1, \ldots, \phi_k]) is finite. As seen in Section 1.7, this catamorphism projects the input to this finite range, whose elements are called \text{classes} [BLW87].
Moreover, exactly \( (2 \times |cand_1| \times \ldots \times |cand_n|) \) triples of \((c, w, r)\) are generated, and this number is bounded by the constant \(2^C N\), where \(C\) is the number of classes. Therefore, the complexity of \( \psi_i e cand_1 \ldots cand_n \) is \(O(1)\). Though the size of the bound increases exponentially with \(N\), in many well-used data types, such as lists and binary trees, \(N\) is small (perhaps one or two). \(\square\)

**Remarks**

This lemma was inspired by Bern et al.’s work on decomposable graphs [BLW87], where a similar condition on decomposable graphs was given. We generalize the idea from decomposable graphs\(^2\) to generic recursive data types. Moreover, we give a concrete linear time algorithm using the optimization function \(opt\) in Figure 4.1.

In the optimization lemma, we give the optimization function \(opt\) to compute an optimal solution. It is worth noting that by slightly changing the definition of the optimization function \(opt\), we can also do recognition and enumeration in linear time. Recognition means recognizing whether or not a solution satisfying the property description exists, and enumeration means counting the number of solutions satisfying the property description [BPT92].

### 4.2 A Sufficient Condition about Weight Function

In the above, we only consider the case that marks are \(True\) and \(False\) and the weight function \(w\) is \(wsum\), which simply computes the sum of marked elements. The previous work restricted the weight function \(w\) to be just the sum of weight of marked elements [Bir00, SHTO00, BLW87, BPT92]. As seen in the coloring problem in Chapter 3, we often need to use a more general weight function. Here, we generalize Lemma 1 with respect to marks and weight function. First, we allow any finite number of elements that are different each other as marks. Second, we allow a class of weight functions. We define the following general form, a kind of list homomorphism [Bir87], which we call homomorphic weight function.

\(^{2}\)Decomposable graphs do not allow the addition of a new element when gluing smaller graphs.
4.2.1 Homomorphic Weight Function

Definition 3 (Homomorphic Weight Function) A function \( w \) is a homomorphic weight function if it is defined as follows:

\[
\begin{align*}
  w & :: D \alpha^* \rightarrow \text{Weight} \\
  w & = \text{plus} \circ \text{map } f \\
  \text{where } & \text{ plus } = [\phi_1, \ldots, \phi_k] \\
  \phi_i & (e, x_1, \ldots, x_n) = e \oplus x_1 \oplus \cdots \oplus x_n
\end{align*}
\]

where \( \oplus \) is an associative binary operator which can be computed in \( O(1) \) time, which has an identity element \( e_\oplus \), and which satisfies the condition called distributivity over \( \uparrow_{id} \):

\[
(\uparrow_{id} / xs) \oplus (\uparrow_{id} / ys) = \uparrow_{id} / [x \oplus y \mid x \in xs \land y \in ys].
\]

A homomorphic weight function allows any \( O(1) \) computation \( f \) over each marked element and a more general operation \( \oplus \) rather than just \( + \) for “summing up”. This enables us to deal with the weight function for the coloring problem in Chapter 3.

The function \( \text{map} \) appeared in the definition of homomorphic weight function is defined as follows:

\[
\begin{align*}
  \text{map} & :: (\alpha \rightarrow \beta) \rightarrow D \alpha \rightarrow D \beta \\
  \text{map } f & = ([\phi_1, \ldots, \phi_k] \\
  \text{where } & \phi_i (e, x_1, \ldots, x_n) = C_i (f e, x_1, \ldots, x_n)
\end{align*}
\]

This is not the map function \( \text{map} \) on lists but the map function on \( D \alpha \). This is a little confusing, but we use the same name since we think it is clear from context.

4.2.2 Generalized Optimization Lemma

Here we generalize Lemma 1 with respect to marks and weight function.

Lemma 2 (Generalized Optimization Lemma) Let \( \text{spec} \) be defined by

\[
\begin{align*}
  \text{spec} & :: D \alpha \rightarrow D \alpha^* \\
  \text{spec} & = \text{mmp } w \ (\text{accept } \circ ([\phi_1, \ldots, \phi_k]) \ ms.
\end{align*}
\]

If the domain of the predicate \( \text{accept} \) is finite, \( w \) is a homomorphic weight function, and \( ms \) is a list of finite number of marks, then \( \text{spec} \) can be solved in linear time. \( \square \)
This lemma allows use of any finite number of marks and any homomorphic weight function. To prove the lemma, we define a generalized optimization function $\text{opt}$ in Figure 4.2. The following equation similar to one in the proof of Lemma 1 holds:

$$\text{spec } x = \text{opt} (f, \oplus, t_\oplus) \text{ accept } \phi_1 \ldots \phi_k \text{ ms } x$$

where $\oplus$ and $f$ appear in the definition of $w$:

$$w ::= D \alpha^* \rightarrow \text{Weight}$$

$$w = \text{plus} \circ \text{map } f$$

where $\text{plus} = ([\phi_1, \ldots, \phi_k])$

$$\phi_i (e, x_1, \ldots, x_n) = e \oplus x_1 \oplus \cdots \oplus x_n$$

We omit the proof since correctness and linearity can be shown in the same way as in proof of Lemma 1. In the following, we use the name $\text{opt}$ for indicating the function $\text{opt}$ in Figure 4.2.

### 4.3 Decomposition Lemma

To apply the optimization lemma, we need to represent the property description $p$ as $\text{accept} \circ ([\phi_1, \ldots, \phi_k])$, where the domain of $\text{accept}$ is finite.
4.3.1 Property with Boolean Functions

The following lemma gives a method for transforming the property description $p$ into the above form when $p$ is defined using mutumorphisms.

**Lemma 3 (Decomposition Lemma)** If the property description $p_0 :: D \alpha^* \rightarrow \text{Bool}$ can be defined as mutumorphisms with other property descriptions $p_i :: D \alpha^* \rightarrow \text{Bool}$ for $i = 1, \ldots, n$, then $p_0$ can be decomposed into

$$p_0 = \text{accept} \circ ([\phi_1, \ldots, \phi_k]),$$

where the domain of $\text{accept}$ is finite.

**Proof.** To prove the lemma, we suppose that mutumorphisms $p_0, p_1, \ldots, p_n$ are defined as follows. For each $i \in \{0, 1, \ldots, n\}$ and each $j \in \{0, 1, \ldots, k\}$,

$$p_i \circ C_j = \phi_{ij} \circ \mathbf{F}_j (p_0 \triangle p_1 \triangle \cdots \triangle p_n).$$

By the mutu tupling theorem, we have

$$p_0 \triangle p_1 \triangle \cdots \triangle p_n = ([\phi_1, \ldots, \phi_k])_D \quad (4.1)$$

where

$$\phi_i = \phi_{0i} \triangle \phi_{1i} \triangle \cdots \triangle \phi_{ni} \quad (i = 1, \ldots, k).$$

By defining

$$\text{accept} (x_0, x_1, \ldots, x_n) = x_0,$$

we obtain

$$p_0 = \text{accept} \circ ([\phi_1, \ldots, \phi_k])_D.$$

Next we show that the domain of $\text{accept}$, i.e., the range of $([\phi_1, \ldots, \phi_k])_D$, is finite. Functions $p_0, p_1, \ldots, p_n$ have ranges which consist of two elements, True and False. So, from equation (4.1), the number of elements in the range of $([\phi_1, \ldots, \phi_k])_D$ is $2^{n+1}$, which is a finite number. This means that the domain of $\text{accept}$ is finite. □

4.3.2 Property with Finite-range Functions

It is easy to see that property can be described using mutumorphisms which consist of functions whose range is finite.
4.4 The Main Theorem

Combining the above two lemmas, we obtain the optimization theorem.

**Theorem 3 (Optimization Theorem)**

The maximum marking problem specified by

\[
\text{spec} :: D \alpha \rightarrow D \alpha^*
\]

\[
\text{spec} = \text{mmp} w p_0 ms
\]

can be solved in linear time if the property description \( p_0 :: D \alpha^* \rightarrow \text{Bool} \) can be defined as mutumorphisms with other property descriptions \( p_i :: D \alpha^* \rightarrow \text{Bool} \) for \( i = 1, \ldots, n \), \( w \) is a homomorphic weight function, and \( ms \) is a list of finite number of marks.

This theorem provides a friendly interface for the derivation of efficient linear-time algorithms solving the maximum marking problems. When the property description is defined as mutumorphisms which consist of \( n \) property descriptions, a derived catamorphism will have range which has \( 2^n \) elements. In many cases, property description \( p \) can be easily described as mutumorphisms which consist of a small number of property descriptions (two to four for examples in this thesis). For example, the party planning problem is described with mutumorphisms which consist of only two property descriptions. This means that the linear-time algorithms obtained are practical.

To see how the theorem works, recall the mis problem in Section 1.7. The property description \( p \) is defined with \( p_1 \) in the following mutumorphic form:

\[
p [x] = \text{True}
\]

\[
p (x : xs) = \text{if marked } x \text{ then } p_1 xs \land p xs \text{ else } p xs
\]

\[
p_1 [x] = \text{not (marked } x)\]

\[
p_1 (x : xs) = \text{not (marked } x)
\]

It follows from the optimization theorem that an efficient linear-time algorithm solving the mis problem can be derived automatically.

It is easy to extend lemma 3 and theorem 3 to allow the property description \( p_0 \) to be defined as mutumorphisms with other functions whose ranges are finite.
4.5 Extension of property

The property description $p$ for the coloring problem can be naturally specified as follows:

\[
\begin{align*}
indep \; xs & = \; indep' \; xs \text{ Neutral} \\
\text{indep'} \; [] \; \text{color} & = \; \text{True} \\
\text{indep'} \; (x : xs) \; \text{color} & = \; \text{kind} \; x \neq \text{color} \land \text{indep'} \; xs \; (\text{kind} \; x).
\end{align*}
\]

But this is not in a required mutumorphic form such that the rule in Sasano et al. [SHTO00] can be applied, because $\text{indep'}$ has an additional accumulating parameter $\text{color}$. Here, $\text{Neutral}$ is used as the initial value of the accumulating parameter, which is different from all the colors used for coloring the elements. The function $\text{kind}$ takes as its argument a marked element and returns the kind of mark of the element. If we insist on specifying $\text{indep}$ in a mutumorphic form, we would have to instantiate all the possible values of $\text{color}$ used by $\text{indep'}$, and could reach the following complicated definition:

\[
\begin{align*}
indep \; [] & = \; \text{True} \\
indep \; (x : xs) & = \; \text{case} \; \text{kind} \; x \; \text{of} \\
& \quad \text{Red} \to \text{indep}_R \; xs \\
& \quad \text{Blue} \to \text{indep}_B \; xs \\
& \quad \text{Yellow} \to \text{indep}_Y \; xs \\
\text{indep}_R \; [] & = \; \text{True} \\
\text{indep}_R \; (x : xs) & = \; \text{case} \; \text{kind} \; x \; \text{of} \\
& \quad \text{Red} \to \text{False} \\
& \quad \text{Blue} \to \text{indep}_B \; xs \\
& \quad \text{Yellow} \to \text{indep}_Y \; xs \\
\text{indep}_B \; [] & = \; \text{True} \\
\text{indep}_B \; (x : xs) & = \; \text{case} \; \text{kind} \; x \; \text{of} \\
& \quad \text{Red} \to \text{indep}_R \; xs \\
& \quad \text{Blue} \to \text{False} \\
& \quad \text{Yellow} \to \text{indep}_Y \; xs \\
\text{indep}_Y \; [] & = \; \text{True} \\
\text{indep}_Y \; (x : xs) & = \; \text{case} \; \text{kind} \; x \; \text{of} \\
& \quad \text{Red} \to \text{indep}_R \; xs \\
& \quad \text{Blue} \to \text{indep}_B \; xs \\
& \quad \text{Yellow} \to \text{False}.
\end{align*}
\]
In fact, this instantiation not only leads to a complicated definition, but also makes the generated program less efficient than that generated by Theorem 3.

For the property \( p \) which is to specify the feasible markings with multiple kinds of marks, the existing approach [SHTO00] (as seen in the definition of \( \text{indep} \) in Chapter 3) only allows \( p \) to be defined in a mutumorphic form with several other functions, say \( p_1, \ldots, p_n \), whose ranges are finite.

\[
\begin{align*}
p \; [\;] & = e \\
p \; (x : xs) & = \phi \; x \; (p \; xs, p_1 \; xs, \ldots, p_n \; xs) \\
\vdots \\
p_i \; [\;] & = e_i \\
p_i \; (x : xs) & = \phi_i \; x \; (p \; xs, p_1 \; xs, \ldots, p_n \; xs) \\
\vdots
\end{align*}
\]

If \( p \) is defined in the mutumorphic form, by applying the tupling transformation [Fok92, HITT97], we can always come up with the following definition for \( p \), a composition of a project function with a \text{foldr}:

\[
p = \text{fst} \circ \text{foldr} \; \psi \; e'
\]

where \( \psi \; x \; es = (\phi \; x \; es, \phi_1 \; x \; es, \ldots, \phi_n \; x \; es) \)

\[
e' = (e, e_1, \ldots, e_n).
\]

### 4.5.1 Finite Accumulative Property

To specify a history-sensitive property, we often want to use an accumulating parameter. So we extend the above \( p \) to a composition of a function with a \text{foldr}_h, a higher order version of \text{foldr}, which is defined as follows:

\[
\begin{align*}
\text{foldr}_h \; (\phi_1, \phi_2) \; \delta \; [\;] \; e & = \phi_1 \; e \\
\text{foldr}_h \; (\phi_1, \phi_2) \; \delta \; (x : xs) \; e & = \phi_2 \; x \; e \; (\text{foldr}_h \; (\phi_1, \phi_2) \; \delta \; xs \; (\delta \; x \; e)).
\end{align*}
\]

Using this function \( \text{foldr}_h \), we define the following form, which we call finite accumulative property.

**Definition 4 (Finite Accumulative Property)** A property \( p \) is a finite accumulative property if it is defined as follows:

\[
p :: [\alpha^*] \rightarrow \text{Bool}
\]

\[
p \; xs = g \; (\text{foldr}_h \; (\phi_1, \phi_2) \; \delta \; xs \; e_0)
\]

where the domain of \( g \) and range of \( \delta \) is finite.
Though we only consider the property with an accumulating parameter on lists, it can be extended to any polynomial data type $D \alpha$.

### 4.5.2 Theorem

Now we propose theorem for property description with an accumulating parameter.

**Theorem 4** Suppose a specification of a maximum multi-marking problem is given as

$$mmp \ w \ p \ ms = \uparrow_w / \circ \ \text{filter} \ p \ \circ \ \text{gen} \ ms.$$ 

If $w$ is a homomorphic weight function

$$w = \oplus / \circ \ \text{map} \ f$$

and $p$ is a finite accumulative property

$$p \ xs = g (\text{foldr}_h (\phi_1, \phi_2) \ \delta \ xs \ e_0),$$

then the maximum multi-marking problem ($mmp \ w \ p \ ms$) can be solved by

$$\text{opt}_{\text{acc}} (f, \oplus, t_{\oplus}) (\lambda(c, e) . g \ c \ \land \ e == e_0) \ \phi_1 \ \phi_2 \ \delta \ ms.$$ 

The definition of $\text{opt}_{\text{acc}}$ is given in Figure 4.3.

This theorem has a form similar to Theorem 3 except for using array in the definition of $\text{opt}_{\text{acc}}$ for efficiency, and it can be proved by induction on the input list. We omit the detailed proof in this thesis. One remark worth making is about the cost of the derived program. Assuming that $\delta$ and $g$ have the types

$$\delta :: \alpha^* \rightarrow \text{Acc} \rightarrow \text{Acc}$$
$$g :: \text{Class} \rightarrow \text{Bool},$$

we can conclude that the generated program using $\text{opt}_{\text{acc}}$ can be computed in $O(|\text{Acc}| \cdot |\text{Class}| \cdot |ms| \cdot n)$ time, where $n$ is the length of input list, $|ms|$ is the number of marks, and $|\text{Acc}|$ and $|\text{Class}|$ denote the size of the type $\text{Acc}$ and the type $\text{Class}$ respectively. That means that our approach is applicable only when the domain of $g$ and the range of $\delta$ is finite. If our approach is applicable, our generated program is much more efficient than the initial specification program $mmp \ w \ p \ ms$, which is exponential.
\begin{verbatim}
let \text{opts} = foldr \psi_2 \psi_1 xs
  in snd (\text{fst} / [(w, r^*) | \text{Just} (w, r^*) \leftarrow [\text{opts}!i \leftarrow \text{range bnds},
   \text{opts}!i \neq \text{Nothing}, \text{accept} i]])

where \psi_1 = \text{array} bnds [(i, g i) | i \leftarrow \text{range bnds}]

\psi_2 x \text{ cand} = \text{accumArray} h \text{ Nothing} bnds
  \left[ ((\phi_2 x^* e c, e), (f x^* \oplus w, x^* : r^*))
   \mid x^* \leftarrow \text{mark ms} x,
   e \leftarrow \text{acclist},
   ((c, _), \text{Just} (w, r^*)) \leftarrow
   [(i, \text{cand}!i) | i \leftarrow [(c', \delta x^* e) | c' \leftarrow \text{classlist}],
     \text{inRange} bnds i,
     \text{cand}!i \neq \text{Nothing}]]

g (c, e) = \text{if} (c == \phi_1 e) \text{ then} \text{Just} (\epsilon, []) \text{ else} \text{Nothing}

h (\text{Just} (w_1, x_1)) (w_2, x_2) = \text{if} w_1 > w_2 \text{ then} \text{Just} (w_1, x_1)
  \text{ else} \text{Just} (w_2, x_2)

h \text{ Nothing} (w, x) = \text{Just} (w, x)

\text{bnds} = ((\text{head} \text{ classlist}, \text{head} \text{ acclist}), (\text{last} \text{ classlist}, \text{last} \text{ acclist}))

\text{acclist} = \text{list of all the values in} \text{ Acc}
\text{classlist} = \text{list of all the values in} \text{ Class}
\end{verbatim}

Figure 4.3: Optimization function \text{opt}_{acc}.
An Example

To see how the theorem works, we demonstrate how to derive a linear algorithm for the coloring problem in Chapter 3. Recall that the specification for the coloring problem has been given in Chapter 3. The weight function has been written in our required form, and the property $\text{indep}$ can be easily rewritten using $\text{foldr}_h$ as follows:

$$
\text{indep } xs = \text{id} \ (\text{foldr}_h \ (\phi_1, \phi_2) \ \delta \ xs \ 0)
$$

where

$$
\phi_1 \ e = \text{True}
$$

$$
\phi_2 \ x \ e \ r = \text{kind } x \neq e \wedge r
$$

$$
\delta \ x \ e = \text{kind } x.
$$

Now applying the theorem quickly yields a linear time algorithm, whose program coded in Haskell is given in Figure 4.4. Notice that in this example, $k = 3$, $|\text{Acc}| = 4$, and $|\text{Class}| = 2$. Evaluating the expression

> coloring [1,2,3,4,5]

gives the result of

$$
[(1,1), (2,3), (3,1), (4,3), (5,1)].
$$

It is worth while to compare the generated algorithms from the two property description with and without an accumulating parameter. Consider the coloring problem with $k$ colors and with certain homomorphic weight function $w$. By using property description with an accumulating parameter, $O(k^2 n)$ algorithm is obtained because $|\text{Acc}| = k + 1$ and $|\text{Class}| = 2$. On the contrary, by using property description in mutumorphic form without accumulating parameters as described in Chapter 3, $O(2^k n)$ algorithm would be obtained by applying the previous method [SHTO00], if it could deal with multiple kinds of marks.

4.6 Calculation Strategy

In this section we show how to derive practical linear-time algorithms that solve the maximum marking problems by applying the optimization theorem.

**Specification** Specify the property description for a given maximum marking problem using a recursive function $p$ on a data structure $R$. 
coloring = optacc (f, (+), 0) accept phi1 phi2 delta [1,2,3]
acclist = [0..3]
classlist = [False, True]
accept (c,e) = c && e==0
f = \x -> case kind x of
  1 -> weight x
  2 -> - (weight x)
  3 -> 0
phi1 e = True
phi2 x e c = kind x /= e && c
delta x e = kind x
kind (_,m) = m
weight (x,_) = x
-----------------(the first half)--------------------------

Figure 4.4: A linear-time Haskell program for the coloring problem (the first half).
optacc (f, oplus, id_oplus) accept phi1 phi2 delta ms xs =
  let opts = foldr psi2 psi1 xs
  in snd (getmax [(w,r) | Just (w,r) <- [ opts!i
                  | i <- range bnds,
                  opts!i /= Nothing,
                  accept i]])

where psi1 = array bnds [(i, g i) | i <- range bnds]
psi2 x cand = accumArray h Nothing bnds
    [((phi2 xm e c, e),
      (f xm `oplus` w, xm:r))
    | xm <- mark ms x,
    e <- acclist,
    ((c,_,),Just (w,r)) <-
    [ (i,cand!i)
    | i <- [(c',delta xm e)
    | c' <- classlist],
    inRange bnds i,
    cand!i /= Nothing
    ]
g (c,e) = if (c == phi1 e) then Just (id_oplus, [])
          else Nothing
h (Just (w1,x1)) (w2,x2) = if w1 > w2 then Just (w1,x1)
                           else Just (w2,x2)
h Nothing (w,x) = Just (w,x)
binds = ((head classlist,head acclist),
         (last classlist,last acclist))
getmax [] = error "No solution."
getmax xs = foldr1 f xs
    where f (w1,cand1) (w2,cand2)
          = if w1>w2 then (w1,cand1) else (w2,cand2)

Figure 4.5: A linear-time Haskell program for the coloring problem (the second half).
Step 1 If $R$ is not a polynomial data type, find a polynomial data structure $D$ into which $R$ can be encoded, and then transform the property description $p$ on $R$ into $p'$ on $D$. If the data structure $R$ is a polynomial data structure, then do nothing. That is, let $p' = p$ and $D = R$.

Step 2 Derive $p'$ in terms of mutumorphisms which consist of several property descriptions $p'_0, p'_1, \ldots, p'_n$ on $D$ by generalizing some part in $p'$ and fusing it. If necessary, do tupling transformation to get a catamorphism, since a catamorphism must be obtained when applying the fusion theorem.

Step 3 Apply the optimization theorem to get a linear-time algorithm that solves the maximum marking problem.

We demonstrate these steps on the party planning problem.

The Party Planning Problem

Specification The inputs in the party planning problem are trees, so we can use the following recursive (regular) data structure.

$$\text{Org} \; \alpha ::= \text{Leader} \; \alpha [\text{Org} \; \alpha]$$

The specification of the problem is

$$\text{pp} :: \text{Org} \; \alpha \rightarrow \text{Org} \; \alpha^*$$

$$\text{pp} = \text{mu} \; \text{s} \; p$$

where the property description $p$ of the party planning problem can be written as follows:

$$p :: \text{Org} \; \alpha^* \rightarrow \text{Bool}$$

$$p \; (\text{Leader} \; v \; []) = \text{True}$$

$$p \; (\text{Leader} \; v \; (t : ts)) =$$

$$\text{not} \; (\text{bothmarked} \; v \; (\text{getLeader} \; t)) \land$$

$$p \; t \land p \; (\text{Leader} \; v \; ts)$$

If the tree contains only a single node, then the property is satisfied. Otherwise we check its root $v_1$ with its children one by one to make certain that both $v_1$ and its child are not marked at the same time, and then we check other parts of the tree recursively. Here $\text{bothmarked} \; v_1 \; v_2$
is used to check whether both $v_1$ and $v_2$ are marked, and $getLeader$ returns the root of the organization tree:

\[
bothmarked :: \alpha^* \to \alpha^* \to \text{Bool} \\
bothmarked v_1 v_2 = \text{marked } v_1 \land \text{marked } v_2
\]

\[
getLeader :: \text{Org } \alpha \to \alpha \\
getLeader (\text{Leader } v \ ts) = v.
\]

Notice that our initial specification is rather straightforward.

**Step 1** The data type $\text{Org } \alpha$ is regular but non-polynomial data type whose nodes can have arbitrarily many children. So we transform this data type $\text{Org } \alpha$ into a polynomial data type. In fact, we can represent the type $\text{Org } \alpha$ by the following binary tree structure, called a rooted tree [BLW87].

\[
\text{RTree } \alpha :: = \text{Root } \alpha \\
| \quad \text{Join } (\text{RTree } \alpha) (\text{RTree } \alpha)
\]

The relation between these two data types can be captured by the following functions.

\[
r2o :: \text{RTree } \alpha \to \text{Org } \alpha \\
r2o (\text{Root } v) = \text{Leader } v \ [] \\
r2o (\text{Join } t_1 t_2) = \text{let Leader } v \ ts = r2o t_2 \\
\text{in Leader } v ((r2o t_1) : ts)
\]

\[
o2r :: \text{Org } \alpha \to \text{RTree } \alpha \\
o2r (\text{Leader } v \ []) = \text{Root } v \\
o2r (\text{Leader } v (t : ts)) = \\
\quad \text{Join } (o2r t) (o2r (\text{Leader } v ts))
\]

These two functions convert data types in linear time.

Next we transform the property description $p$ on $\text{Org } \alpha$ to $p'$ on $\text{RTree } \alpha$. Let $p'$ and $getLeader'$ be the functions on $\text{RTree } \alpha$ which correspond to $p$ and $getLeader$ on $\text{Org } \alpha$. These functions should satisfy the following equations.

\[
p' :: \text{RTree } \alpha^* \to \text{Bool} \\
p' t = p (r2o t)
\]

\[
getLeader' :: \text{RTree } \alpha \to \text{ Bool} \\
getLeader' t = getLeader (r2o t)
\]
A simple fusion calculation yields

\[
\begin{align*}
p' (\text{Root } v) &= \text{True} \\
p' (\text{Join } t_1 t_2) &= \\
&\quad \text{not} (\text{marked} (\text{getLeader}' t_1) \land \\
&\quad \text{marked} (\text{getLeader}' t_2)) \land \\
&\quad p' t_1 \land p' t_2
\end{align*}
\]

\[\text{getLeader}' (\text{Root } v) = v\]
\[\text{getLeader}' (\text{Join } t_1 t_2) = \text{getLeader}' t_2.\]

**Step 2** To represent \(p'\) using mutumorphisms which consist of only property descriptions, we generalize the part \(\text{marked} \circ \text{getLeader}'\) and let it be \(lm'\).

\[
\begin{align*}
lm' &:: \ RTree \alpha^* \rightarrow \text{Bool} \\
lm' &= \text{marked} \circ \text{getLeader}'
\end{align*}
\]

By a simple fusion calculation we can get

\[
\begin{align*}
lm' (\text{Root } v) &= \text{marked } v \\
lm' (\text{Join } t_1 t_2) &= lm' t_2.
\end{align*}
\]

Now we can represent \(p'\) using mutumorphisms which consist of the following two property descriptions.

\[
\begin{align*}
p' (\text{Root } v) &= \text{True} \\
p' (\text{Join } t_1 t_2) &= \text{not} (lm' t_1 \land lm' t_2) \land \\
&\quad p' t_1 \land p' t_2 \\
lm' (\text{Root } v) &= \text{marked } v \\
lm' (\text{Join } t_1 t_2) &= lm' t_2
\end{align*}
\]

**Step 3** By applying our optimization theorem, we get the following linear-time algorithm.

\[
pp = r'^2 o (\text{opt accept } \phi_1 \phi_2) \circ o2r
\]

**where**

- \(\text{accept } (x_0, x_1) = x_0\)
- \(\phi_1 v = (\text{True}, \text{marked } v)\)
- \(\phi_2 (a_1, b_1) (a_2, b_2) = \)
  \[(\text{not} (b_1 \land b_2) \land a_1 \land a_2, b_2)\]

45
So much for the derivation. Note that we can go further to compute $\phi_i$ statically and store it in a table, though this is not necessary. To do so, define four classes by

$$c_0 = (False, False)$$
$$c_1 = (False, True)$$
$$c_2 = (True, False)$$
$$c_3 = (True, True)$$

and simplify the functions $accept$, $\phi_1$, and $\phi_2$. We write $\phi_2$ as a table.

$$accept
c = (c == c_2) \lor (c == c_3)$$

$$\phi_1\ v = if\ marked\ v\ then\ c_3\ else\ c_2$$

<table>
<thead>
<tr>
<th>$\phi_2$</th>
<th>$c_0$</th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_0$</td>
<td>$c_0$</td>
<td>$c_1$</td>
<td>$c_0$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$c_1$</td>
<td>$c_0$</td>
<td>$c_1$</td>
<td>$c_0$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$c_2$</td>
<td>$c_0$</td>
<td>$c_1$</td>
<td>$c_2$</td>
<td>$c_3$</td>
</tr>
<tr>
<td>$c_3$</td>
<td>$c_0$</td>
<td>$c_1$</td>
<td>$c_2$</td>
<td>$c_1$</td>
</tr>
</tbody>
</table>
Chapter 5

Examples

In this chapter, we give more examples, showing that our proposed generation rule is quite general and powerful. Examples include several kinds of knapsack problems, scheduling problem, a data mining problem mentioned in Chapter 1, several problems in the book Algebra of Programming [BdM96].

5.1 Paragraph Formatting Problem

The paragraph formatting problem is the problem of breaking a sequence of words into lines to form a paragraph. At least one blank space must exist between any adjacent two words in the same line. Line length, i.e., the number of characters each line holds, is fixed as $m$. We want to minimize the sum of the number of blank spaces in all the lines excluding the last line. We assume that the input sequence of words is given by a list of words and that a word is expressed by its length since the spelling of words is not needed. For example, the sequence of words "This is a dog. They are cats." is expressed as the list $[4, 2, 1, 4, 4, 3, 5]$.

We would like to treat this problem as a multi-marking problem, that is, to describe this problem in the form

$$mmp \ w \ p \ ms.$$ 

We use three kinds of marks, 1, 2, and 3. So, $k = 3$. If a word is marked 2, then it indicates that the word is the last word of the line it belongs to. If a word is marked 3, then it indicates that the word belongs to the last line. The other words are marked 1. We make special treatment of the last line in order
to exclude the blank spaces when computing the sum of the number of blank spaces. The property $p$ checks whether a marking represents a valid breaking or not. We describe the property $p$ by using an accumulating parameter. The accumulating parameter holds the pair of position $pos$ and mark $mk$, where $pos$ represents the last position filled by the previous words in the current line, and $mk$ represents the kind of mark of the previous word. We define the property $p$ as follows:

\[
p \ x s = p' \ x s \ (0, 2) \\
p' \ [] \ (pos, mk) = mk \neq 1 \\
p' \ (x : x s) \ (pos, mk) =
\]

\[
\text{case} \ mk \ \text{of} \\
1 \rightarrow \text{case} \ \text{kind} \ x \ \text{of} \\
1 \rightarrow \text{pos} + l \ x + 1 \leq m \ \land \ p' \ x s \ (pos + l \ x + 1, 1) \\
2 \rightarrow \text{pos} + l \ x + 1 \leq m \ \land \ p' \ x s \ (0, 2) \\
3 \rightarrow \text{False} \\
2 \rightarrow \text{case} \ \text{kind} \ x \ \text{of} \\
1 \rightarrow \text{pos} + l \ x \leq m \ \land \ p' \ x s \ (pos + l \ x, 1) \\
2 \rightarrow \text{pos} + l \ x \leq m \ \land \ p' \ x s \ (0, 2) \\
3 \rightarrow \text{pos} + l \ x \leq m \ \land \ p' \ x s \ (pos + l \ x, 3) \\
3 \rightarrow \text{case} \ \text{kind} \ x \ \text{of} \\
1 \rightarrow \text{False} \\
2 \rightarrow \text{False} \\
3 \rightarrow \text{pos} + l \ x + 1 \leq m \ \land \ p' \ x s \ (pos + l \ x + 1, 1),
\]

where we use the function $l$ to compute the length of a word.

Next, we have to describe the weight function $w$. We want to minimize the sum of the number of blanks except for the last line. The function $\text{white}$ which returns the sum can be written as follows:

\[
\text{white} = +/ \circ \text{map} \ f \\
\text{where} \ f \ x = \text{case} \ \text{kind} \ x \ \text{of} \\
1 \rightarrow -l \ x \\
2 \rightarrow m - l \ x \\
3 \rightarrow 0.
\]

Using this function, we can define the weight function $w$ as follows:

\[
w \ x = -(\text{white} \ x).
\]
This can be easily transformed into the following form:

\[ w = +/ \circ \text{map } f \]

where \( f \ x = \text{case } \text{kind } x \text{ of} \)

1 \( \rightarrow \ l \ x \)

2 \( \rightarrow -m + l \ x \)

3 \( \rightarrow 0. \)

Now the paragraph formatting problem is written as follows:

\[ \text{mmp } w \ p [1, 2, 3]. \]

By applying the Theorem 4 (by using the rule \text{mmpRule}), we can obtain an \( O(mn) \) algorithm where \( n \) is the number of words. This complexity is achieved by the fact that the number of kinds of marks is three, the size of the accumulating parameter \( |\text{Acc}| \) is \( 3(m + 1) \), and the size of the function \( g \) (in this case \( \text{id} \) \(|\text{Class}| \) is 2.

### 5.2 Security Van Problem

The security van problem can be specified as follows [BdM96].

Suppose a bank has a known sequence of deposits and withdrawals. For security reasons the total amount of cash in the bank should never exceed some fixed amount \( N \), assumed to be at least as large as any single transaction. To cope with demand and supply, a security van can be called upon to deliver funds to the bank or to take away a surplus. The problem is to compute a schedule under which the van visits the bank a minimum number of times.

In order to specify this problem as a maximum multi-marking problem, we consider the security of transactions. A sequence \([a_1, a_2, \ldots, a_n]\) of transactions is called secure if there is an amount \( r \), indicating the total amount of cash in the bank at the beginning of the sequence of transactions, such that each of the sums

\[ r, r + a_1, r + a_1 + a_2, \ldots, r + a_1 + \cdots + a_n \]

lies between zero and \( N \). For example, taking \( N = 10 \), the sequence \([3, -5, 6]\) is secure because the van can take away or deliver enough cash to make an
initial reserve of, for example, 5. Given the constraint that \( N \) is no smaller than any single transaction, every singleton sequence is secure, so a valid schedule certainly exists.

To formalize the constraint, define

\[
\text{ceiling} = \uparrow_+ / \circ \text{inits} \\
\text{floor} = \downarrow_+ / \circ \text{inits}
\]

where \( \text{inits} \) is a function which takes as its argument a list and returns the list which has all the initial segments including empty list. A sequence \( x \) of transactions is secure if and only if

\[
\text{ceiling} x - \text{floor} x \leq N.
\]

Considering this condition, we can define property \( p \) in the following way. We express the time the van visits by marking 1 to a transaction after which the van visits. Transactions marked 2 represent the other transactions. The accumulating parameter holds a triple of sum, ceiling, and floor for each initial segment.

\[
p \ x s = p' \ x s \ (0, 0, 0) \\
p' \ [] \ (s, c, f) = \text{True} \\
p' \ (x : xs) \ (s, c, f) = \\
\quad \text{let} \ (s', c', f') = (s + w x, c \uparrow id s', f \downarrow id s') \\
\quad \text{in case} \ \text{kind} x \ \text{of} \\
\quad \quad 1 \to \text{if} \ c' \leq f' \ \text{then} \ p' \ x s \ (w x, 0 \uparrow id w x, 0 \downarrow id w x) \\
\quad \quad \text{else} \ p' \ x s \ (w x, 0 \uparrow id w x, 0 \downarrow id w x) \\
\quad \quad 2 \to \text{if} \ c' \leq f' \ \text{then} \ p' \ x s \ (s', c', f') \\
\quad \quad \text{else} \ False
\]

Here, the function \( w \) takes as its argument a marked transaction and returns the amount of it. We want to minimize the number of times the van visits, so we first define the function \( \text{times} \) which computes the times the van visits.

\[
\text{times} = +/ \circ \text{map } f \\
\quad \text{where} \ f \ x = \text{case} \ \text{kind} x \ \text{of} \\
\quad \quad 1 \to 1 \\
\quad \quad 2 \to 0
\]

Using this function, we can define the weight function \( w \) as follows:

\[
w \ x = - (\text{times } x).
\]
This can be easily transformed into the following form:

\[ w = +/ \circ \text{map } f \]

where \( f \ x = \text{case } \text{kind } x \text{ of} \)

\[ 1 \rightarrow -1 \]
\[ 2 \rightarrow 0. \]

Now the security van problem is written as follows:

\[ \text{mmp } w \ p \ [\text{True, False}] \]

The weight function \( w \) is written in the required form, and the property \( p \) can be easily rewritten into the required form, though we omit the form. By applying Theorem 4, we obtain \( O(N^3n) \) algorithm because \(|\text{Acc}| = (N + 1)^2(2N + 1), k = 2, \) and \(|\text{Class}| = 2. \)

5.3 Knapsack Problem

The knapsack problem [MT90] is a well known combinatorial optimization problem. There are several problems called knapsack problem such as 0-1 knapsack problem, 0-1 multiple knapsack problem, multidimensional knapsack problem, and so on. Here we consider the simplest one, the 0-1 knapsack problem and derive an efficient linear time algorithm from simple specification [SHT01c].

Input of the 0-1 knapsack problem is a set of items each of which has weight and value. Output is a feasible selection of items whose value sum is maximum in all the feasible item selections. A selection is feasible when sum of weight of selected items does not exceed the given capacity \( C \). We assume weight of items are integers. Without this assumption, this problem becomes NP-hard.

5.3.1 Specification

We express an item by pair of weight and value \((\text{Weight}, \text{Value})\) and a set of items by non-empty list \([\{\text{Weight}, \text{Value}\}]\). Knapsack problem is a maximum marking problem where weight function \( w \) computes sum of value of marked (selected) items and property \( \text{knap} \) checks that sum of weight of marked items does not exceed the given capacity \( C \).
\[ \text{knap } xs = \text{weightsum } xs \leq C \]

The argument of \text{knap} is a list of items, each of which is accompanied by a mark which indicates whether the item is selected or not.

### 5.3.2 Property by Finite Mutumorphisms

In order to express property \text{knap} as finite mutumorphisms on non-empty list, we introduce the following function \text{cut}.

\[
\text{cut} :: \text{Weight} \rightarrow \text{Weight} \\
\text{cut } w = \begin{cases} 
\text{if } w \leq C \text{ then } w & \text{else } C + 1 
\end{cases}
\]

This function \text{cut} takes weight \( w \) as its argument and returns \( C + 1 \) if \( w \) is greater than \( C \) and returns \( w \) otherwise. By using this function \text{cut}, we can express property \text{knap} as follows:

\[
\text{knap } xs = \text{sumw } xs \leq C \\
\text{sumw } [x] = \text{cut } (\text{if } \text{marked } x \text{ then } \text{weight } x \\
\text{else } 0) \\
\text{sumw } (x : xs) = \\
\text{cut } ((\text{if } \text{marked } x \text{ then } \text{weight } x \\
\text{else } 0) + \text{sumw } xs)
\]

The function \text{marked} checks whether the element is marked or not.

### 5.3.3 Property by Finite Accumulative Property

We express selection by marking 1 to selected items and 2 to the others. The property for 0-1 knapsack problem can be described as follows. The accumulating parameter holds a value from 0 to \( C \), which indicates the remaining capacity of knapsack.

\[
\text{knap } xs = \text{knap'} xs \ C \\
\text{knap'} [] e = \text{True} \\
\text{knap'} (x : xs) e = \begin{cases} 
1 & \text{if } e \geq w \ x \text{ then } \text{knap'} xs \ (e - w \ x) \text{ else } \text{False} \\
2 & \text{knap'} xs \ e 
\end{cases}
\]

Here, the function \( w \) returns the weight of the item.
5.3.4 Weight function

We want to maximize the value of selected items, so we can define the weight function $w$ as follows:

$$w = +/ \circ \text{map } f$$

where $f x = \text{case } \text{kind } x \text{ of}$

$$1 \rightarrow \text{value } x$$
$$2 \rightarrow 0.$$  

Here, the function $\text{value}$ returns the value of the item.

Now the 0-1 knapsack problem is written as follows:

$$\text{mmp } w \text{ knap } [\text{True}, \text{False}].$$

The weight function $w$ is written in the required form, and the property $\text{knap}$ can be easily rewritten into the required form, though we omit the form.

5.3.5 Derivation by Theorem 4

By applying Theorem 4, we obtain $O(Cn)$ algorithm because $|\text{Acc}| = C + 1$, $k = 2$, and $|\text{Class}| = 2$.

5.3.6 Derivation by Theorem 3

Since the range of the property $\text{knap}$ is $\{\text{True}, \text{False}\}$ and the range of the function $\text{sumw}$ is $\{0, 1, \ldots, C + 1\}$, the functions $\text{knap, sumw}$ constitute multumorphisms on non-empty lists. So, by applying Theorem 3, we get the following algorithm:

$$\text{knapsack } x = \text{opt accept } \phi_1 \phi_2 x$$

where

$$\text{accept } x = x \leq C$$
$$\phi_1 x = \text{cut } (\text{if } \text{marked } x \text{ then } \text{weight } x$$
$$\text{else } 0)$$
$$\phi_2 (x, y) = \text{cut } ((\text{if } \text{marked } x \text{ then}$$
$$\text{weight } x \text{ else } 0) + y)$$

The complexity of this algorithm is $O(Cn)$ where $n$ is the number of items.
5.4 Knapsack Problem on List

In the previous subsection, we considered the ordinary knapsack problem, which does not have structure on items. In this section, we consider a variant of knapsack problem, where items have list structure. Here as an example we consider an additional condition that we cannot select adjacent items.

5.4.1 Specification

The condition that adjacent items cannot be selected is described as follows:

\[
\text{indep } [x] = \text{True} \\
\text{indep } (x : xs) = \\
\quad \begin{cases} 
\text{if marked } x \text{ then} & \neg (\text{marked } (\text{hd } xs)) \land \text{indep } xs \\
\text{else indep } xs 
\end{cases} \\
\text{hd } [x] = x \\
\text{hd } (x : xs) = x
\]

When the item list is a singleton \([x]\), since the number of items is one and thus adjacent items cannot be selected, \text{indep} returns \text{True}. When the item list is \((x : xs)\), there are more than two items exist. When the top item \(x\) is selected, from the remaining items \(xs\) we should select items independently and must not select the head item of \(xs\). The function \(\text{hd}\) takes a non-empty list as its argument and returns the head element of the list. By using this property \text{indep}, we can describe the property of this problem as follows:

\[ p \; xs = \text{indep } xs \land \text{knap } xs. \]

5.4.2 Derivation

This property \(p\) is not described as finite mutumorphisms, because the range of the function \(\text{hd}\) is not finite generally (for example, in the case that the type of weight of item is Integer). In order to describe this property \(p\) as finite mutumorphisms, let

\[ p_1 = \neg (\text{marked } (\text{hd } xs)). \]

By applying fusion transformation, \(p_1\) is transformed into the following form:

\[ p_1 [x] = \neg (\text{marked } x) \]
\[ p_1 (x : xs) = \neg (\text{marked } x) \]
Thus the property $indep$ is described as

$$
\begin{align*}
    indep \ [x] &= \text{True} \\
    indep \ (x : xs) &= \text{if marked } x \\
    \quad \text{then } p_1 \ xs \land indep \ xs \\
    \quad \text{else indep } xs,
\end{align*}
$$

so the functions $p, indep, p_1, knap, sumw$ constitute finite mutumorphisms on non-empty list. By applying Theorem 3, we get $O(Cn)$ algorithm where $n$ is the number of items, though we omit the result.

### 5.5 Tree Knapsack Problem

In this section, we consider another variant of knapsack problem, where items have tree structure. Here as an example we consider an additional condition that selected items are connected by edges. In [dM95], by using relational calculus, a linear time algorithm for a similar problem to this problem, but one have to check two prerequisites when applying their theorem, which makes the derivation difficult. In contrast, in our optimization theorem, the prerequisites is simple enough for ordinary functional programmers to check, which results in easy derivation and makes it possible to automate the derivation, which will be seen in Chapter 6.

#### 5.5.1 Specification

We consider the binary tree which is defined as follows:

$$
Tree \ \alpha ::= \ \text{Leaf } \alpha \mid \text{Bin } (\alpha, Tree \ \alpha, Tree \ \alpha)
$$

The condition that sum of weight does not exceed the given capacity $C$ can be defined as follows:

$$
\begin{align*}
    tk \ t &= tws \ t \leq C \\
    tws \ (\text{Leaf } x) &= \\
    \quad \text{cut } (\text{if marked } x\ \text{then weight } x\ \text{else } 0) \\
    tws \ (\text{Bin } (x, t_1, t_2)) &= \\
    \quad \text{cut } ((\text{if marked } x\ \text{then weight } x\ \text{else } 0) \\
    \quad + tws \ t_1 \ + tws \ t_2)
\end{align*}
$$
Next, the condition that selected items are connected by edges can be described as follows:

\[
\begin{align*}
tc(Leaf \, x) &= \text{True} \\
tc(Bin (x, t_1, t_2)) &= \\
    &\quad \text{if } \text{marked } x \text{ then } tc' \, t_1 \land tc' \, t_2 \\
    &\quad \text{else } (nm \, t_1 \land tc \, t_2) \lor (tc \, t_1 \land nm \, t_2)
\end{align*}
\]

If the tree is \( Leaf \, x \), then it satisfies the condition, so \( tc \) returns true. If the tree is \( Bin (x, t_1, t_2) \), then if the item \( x \) is selected then \( tc \) checks whether \( t_1 \) and \( t_2 \) satisfy the condition that selected elements constitute a prefix of the tree, \( tc' \), otherwise one of \( t_1 \) and \( t_2 \) should not have any selected items and the other one have to satisfy the property \( tc \). The property that tree does not have any selected items can be defined as follows:

\[
\begin{align*}
nm(Leaf \, x) &= \neg (\text{marked } x) \\
nm(Bin (x, t_1, t_2)) &= \\
    &\quad \neg (\text{marked } x) \land nm \, t_1 \land nm \, t_2
\end{align*}
\]

The property \( tc' \) can be defined as follows:

\[
\begin{align*}
tc'(Leaf \, x) &= \text{True} \\
tc'(Bin (x, t_1, t_2)) &= \\
    &\quad \text{if } \text{marked } x \text{ then } tc' \, t_1 \land tc' \, t_2 \\
    &\quad \text{else } nm \, t_1 \land nm \, t_2
\end{align*}
\]

This property \( tc' \) checks prefix property, that is, if an item \( x \) is selected, then the parent of \( x \) have to be selected. By using these properties, the property of the tree knapsack problem \( p \) can be described as follows:

\[
p \, t = tk \, t \land tc \, t
\]
5.5.2 Derivation

Since the functions $p, tk, tc, tws, nm, tc'$ constitute finite mutumorphisms, by applying Theorem 3, we can obtain the following algorithm:

\[ \text{knap tree } t = \text{opt accept } \phi_1 \phi_2 t \]

where

\[ \text{accept } (a, b, c, d) = a \leq C \land b \]

\[ \phi_1 x = (\text{cut (if marked } x \text{ then weight } x \]

\[ \text{else 0)}, \]

\[ \text{True, not (marked } x\text{), True}) \]

\[ \phi_2 (x, (a_1, b_1, c_1, d_1), (a_2, b_2, c_2, d_2)) = \]

\[ (\text{cut (if marked } x \text{ then weight } x \]

\[ \text{else 0) + } a_1 + a_2), \]

\[ \text{if marked } x \text{ then } d_1 \land d_2 \]

\[ \text{else } (c_1 \land b_2) \lor (b_1 \land c_2), \]

\[ \text{not (marked } x\text{) \land c_1 \land c_2,} \]

\[ \text{if marked } x \text{ then } d_1 \land d_2 \]

\[ \text{else } c_1 \land c_2) \]

The obtained algorithm is $O(C^2 n)$ algorithm, where $n$ is the number of items.

5.6 0-1 Multiple Knapsack Problem

Here we consider the 0-1 multiple knapsack problem, a variant of knapsack problem. In this problem, there are $m$ bags whose capacities are $k_1, k_2, \ldots, k_m$. Each item $x_i$ is given value $v_i$ and weight $w_i$ for $i = 1, \ldots, n$. Required is to find a feasible assignment of items to bags that maximizes the sum of values of selected items. Here, an item can be assigned to no bag. An assignment is feasible when sum of weight of items in each bag does not exceed the given capacity $k_i$ for $i = 1, \ldots, m$. This problem can be specified as a maximum multi-marking problem. We assume that weight of items are integers.

5.6.1 Specification

We express items using a list, as we did in the ordinary knapsack problem. Property $p$ can be defined as follows:
\[ p \, x s = \land (p_1 \, x s, p_2 \, x s, \ldots, p_m \, x s) \]
\[ p_i \, x s = \text{weightsum}_i \, x s \leq k_i \quad (i = 1, \ldots, m) \]

Weight function \( w \) can be defined as follows:

\[ w = +/ \circ \text{map} \, f \]
\[ \text{where } f \, x = \text{case kind } x \, \text{of} \]
\[ 0 \rightarrow 0 \]
\[ \_ \rightarrow \text{value } x \]

Here, the function \( \text{value} \) returns the value of the item.

### 5.6.2 Derivation

We introduce the functions \( \text{cut}_i \) for \( i = 1, \ldots, m \) and describe the property \( p \) as finite mutumorphisms.

\[ p \, x s = \land (p_1 \, x s, p_2 \, x s, \ldots, p_m \, x s) \]
\[ p_i \, x s = \text{weightsum}_i \, x s \leq k_i \quad (i = 1, \ldots, m) \]
\[ \text{sumw}_i \, [x] = \text{cut}_i \, (\text{case kind } x \, \text{of} \]
\[ i \rightarrow \text{weight } x \]
\[ \_ \rightarrow 0) \quad (i = 1, \ldots, m) \]
\[ \text{sumw}_i \, (x : x s) = \text{cut}_i \, (\text{case kind } x \, \text{of} \]
\[ i \rightarrow \text{weight } x \]
\[ \_ \rightarrow 0) + \text{sumw} \, x s \quad (i = 1, \ldots, m) \]
\[ \text{cut}_i \, w = \text{if } w \leq k_i \, \text{then } w \, \text{else} \, k_i + 1 \]

By applying the Theorem, we obtain an \( O(k_1 \cdot k_2 \cdots k_m \cdot m \cdot n) \) algorithm.

### 5.7 Multidimensional Knapsack Problem

Here we consider the multidimensional knapsack problem, another variant of knapsack problem. In this problem, there are \( m \) kinds of resources whose capacities are \( c_1, c_2, \ldots, c_m \). Each item \( x_i \) is given value \( v_i \) and consumes \( r_{ij} \) from \( i \)th resource, for \( i = 1, \ldots, m \) and \( j = 1, \ldots, n \). Required is to find a feasible selection of items that maximizes the sum of values of selected items. Here, a selection is feasible when consumption of each resource does not exceed the given capacity \( c_i \) for \( i = 1, \ldots, m \). This problem can be specified as a maximum marking problem. We assume that resource consumptions of each item are integers.

58
5.7.1 Specification

We express items using a list, as we did in the ordinary knapsack problem. Property $p$ can be defined as follows:

\[
\begin{align*}
    p \; xs &= \land (p_1 \; xs, p_2 \; xs, \ldots, p_m \; xs) \\
    p_i \; xs &= resourcesum_i \; xs \leq c_i \quad (i = 1, \ldots, m)
\end{align*}
\]

Here the function \textit{resourcesum}_i computes the sum of consumption of $i$th resource.

Weight function $w$ can be described as follows:

\[
\begin{align*}
    w &= +/ \circ \text{map} \; f \\
    \text{where } f \; x &= \text{case kind } x \text{ of } \\
    &\quad 1 \to \text{value } x \\
    &\quad 2 \to 0.
\end{align*}
\]

Here, the function \textit{value} returns the value of the item.

5.7.2 Derivation

In order to describe property $p$ in finite mutumorphisms, we introduce the auxiliary functions $\textit{cut}_i$ for $i = 1, \ldots, m$ and describe the property $p$ as finite mutumorphisms.

\[
\begin{align*}
    p \; xs &= \land (p_1 \; xs, p_2 \; xs, \ldots, p_m \; xs) \\
    p_i \; xs &= \text{sumr}_i \; xs \leq c_i \quad (i = 1, \ldots, m) \\
    \text{sumr}_i \; [x] &= \text{cut}_i \; (\text{case kind } x \text{ of } \\
    &\quad 1 \to r_i \; x \\
    &\quad 2 \to 0) \quad (i = 1, \ldots, m) \\
    \text{sumr}_i \; (x : xs) &= \text{cut}_i \; (\text{case kind } x \text{ of } \\
    &\quad 1 \to r_i \; x \\
    &\quad 2 \to 0) + \text{sumr} \; xs \quad (i = 1, \ldots, m) \\
    \text{cut}_i \; r &= \text{if } r \leq c_i \text{ then } r \text{ else } c_i + 1
\end{align*}
\]

Here, $r_i$ takes an item as its argument and returns consumption of $i$th resource. By applying the Theorem, we obtain an $O(c_1 \cdot c_2 \cdots c_m \cdot n)$ algorithm.
5.8 Weighted Interval Selection Problem

Given a set of weighted intervals, the weighted interval selection problem is to select a maximum-weight subset such that any two selected intervals are disjoint [ES00]. An application of this problem is a scheduling of jobs whose start and end times are fixed and only one job can be executed at a time. We assume that start and end times are represented by integers. This assumption is natural in real-world jobs, where we mean that the time unit is a day or an hour or a minute or a second, and so on.

Suppose the job set is given as a list of jobs in the order of start time, that is, if job A starts earlier than job B, then job A appears earlier than job B in the list. We express a job by a 3-tuple of the start time, the time which it takes, and the weight of the job. Here we express start time by the difference from the previous job in the job list except for the first job. We express the time of the first job as 0. This way of expressing start time is for applying Theorem 4. For example, the list

\[ \text{jobs} = [(0, 3, 2), (2, 4, 3), (3, 2, 5)]. \]

is a job list, and \( \text{jobs} \) represents three jobs where the second job starts at time 2, and the third job starts at time 5, provided that the first job starts at time 0. Feasible solutions are selecting the first and the third job or selecting only one job. So, the maximum solution is selecting the first and the third job. We express a selection by marking 1 to selected jobs and 2 to the others. For example, maximum solution for \( \text{jobs} \) is expressed as

\[ [(0, 3, 2), 1), ((2, 4, 3), 2), ((3, 2, 5), 1)]. \]

Property \( p \) checks that the selected jobs do not overlap each other. So, \( p \) can be defined as follows. The accumulating parameter represents the time the currently executed job takes until it ends.

\[
\begin{align*}
    p \ [x] \ e & = p' \ [x] \ 0 \\
    p' \ [] \ e & = \text{True} \\
    p' \ (x : xs) \ e & = \text{case kind } x \ \text{of} \\
        1 & \rightarrow \text{if } e - s \ x > 0 \ \text{then } \text{False} \ \text{else } p' \ xs \ (t \ x) \\
        2 & \rightarrow \text{if } e - s \ x > 0 \ \text{then } p' \ xs \ (e - s \ x) \ \text{else } p' \ xs \ 0
\end{align*}
\]

Here the function \( s \) takes as its argument a job \( x \) and returns the start time of it, that is, the first element of the 3-tuple. The function \( t \) takes as its
argument a job $x$ and returns the time it takes, that is, the second element of the 3-tuple.

We want to maximize the sum of weight of selected jobs, so we can define the weight function $w$ as follows:

$$w = +/ \circ \text{map } f$$

where $f \ x = \begin{cases} 1 \to w \ x \\ 2 \to 0. \end{cases}$

Here the function $w$ takes as its argument a job $x$ and returns the weight of it, that is, the third element of the 3-tuple.

Now the weighted interval selection problem is written as follows:

$$mmp \ w \ p \ [\text{True}, \text{False}].$$

The weight function $w$ is written in the required form, and the property $p$ can be easily rewritten into the required form, though we omit the form. By applying Theorem 4, we obtain $O(Wn)$ algorithm, where $W$ is the maximum length among all jobs, because $|Acc| = W + 1$, $|Class| = 2$, and $k = 2$.

### 5.9 Mining Optimized Gain Association Rules

Data mining, which is a technology for obtaining useful knowledge from large database, has been gradually recognized as an important subject. Algorithms for data mining have to be efficient, since target database is often huge. There have been developed many efficient algorithms for various kinds of data mining problems, among which the problem of mining optimized association rules has attracted researchers [BRS99, FMMT96a, FMMT96b].

To show concretely the problem of mining optimized association rules, we consider the following example. Suppose there is a database recording customers’ transactions in a shop, and we are interested in the association rules like the following form:

$$(age \in [a..b]) \Rightarrow \text{BuyRibbon}$$

whose confidence exceeds a given threshold $\theta$. There are many rules of the above form by changing $a$ and $b$. Among them, we would like to find the
range of age that maximizes the gain: \{ the number of customers who bought ribbon whose age are between \( a \) and \( b \) \} minus \{ the threshold number, \( i.e., \theta \) times the number of customers whose age is between \( a \) and \( b \) \}. Suppose that the shop makes a profit if 100\% of customers buy ribbon. Then, the optimized gain range \( [a..b] \) is the range of customers that maximizes the shop’s profit with respect to the section of ribbon.

This is an example of mining optimized gain association rules problem. This problem is transformed to the problem called maximum segment sum problem (MSS for short) [FMMT96a], and we have a linear time algorithm [Ben84]. Input of the MSS problem is a number list \( xs \), and output is a consecutive sublist of \( xs \) that has the maximum sum among all the consecutive sublists of \( xs \). For example, in the case of \( xs = [5, -10, 20, -15, 30, -5] \), the result is \([20, -15, 30]\), which has the maximum sum 35.

Rule of the above form may not be satisfactory in some cases. For example, we may hope to find up to \( k \) ranges of age for the rule

\[(\forall_{i=1}^{k} \text{age} \in [a_i..b_i]) \Rightarrow \text{BuyRibbon}\]

that maximizes the gain. This problem, which is transformed to a \( k \)-MSS problem (Section 5.9.1), is more general and more difficult to be solved efficiently. A smart \( O(kn) \) algorithm has been proposed in [BRS99], but its correctness is not easy to verify. Furthermore, it is difficult to adapt the algorithm even for a simple modification. For instance, we may want to compute up to \( k \) ranges such that the length of each range is between 5 and 10.

In this section, we show that an efficient linear time algorithm for mining optimized gain association rules is systematically derived from a simple specification [SHTO01b]. Our approach not only automatically guarantee the correctness of the derived algorithm, but also is easy to derive new algorithms for modification of the problem.

### 5.9.1 The Problem of Mining Optimized Gain Association Rules

In this section, we show the problem of mining optimized gain association rules [BRS99] can be transformed to the \( k \)-MSS problem.

Recall the problem in Chapter 1. Input database is a set of tuples, each of which holds information about a customer, including an integer value that indicates the age of the customer and a boolean value that indicates whether or not the customer bought ribbon.
From the input database, we would like to mine a rule $R_{\text{ribbon}}$ of the following form:

$$(\text{age} \in [a..b]) \Rightarrow \text{BuyRibbon}.$$ 

This rule means that a customer of age between $a$ and $b$ often buys ribbon in a confidence no less than $\theta$. There are many rules of the above form by changing $a$ and $b$. Among them, we would like to find one that maximizes the gain of the rule $R_{\text{ribbon}}$:

$$\text{gain}(R_{\text{ribbon}}) = \sup(\text{age} \in [a..b] \land \text{BuyRibbon}) - \theta \times \sup(\text{age} \in [a..b]).$$

For a condition $C$, $\sup(C)$ is defined as the number of tuples that satisfy the condition $C$ in the database. Intuitively, this gain shows to what extent the number of tuples establishing the rule $R_{\text{ribbon}}$ exceeds what is expected with respect to the least confidence $\theta$.

As an assumption for an efficient algorithm, input tuples are sorted according to customers’ age\(^1\). Then, for each set of tuples that belong to the same age, we compute $g = v - \theta \times u$, where $u$ is the number of customers and $v$ is the number of customers who bought ribbon. As a result, a list $gs$ of values of $g = v - \theta \times u$ is obtained. For example, suppose that record of customers is as in Table 5.1, which is obtained after the sorting process, with $\theta = 0.2$. For this example, $gs = [5, -10, 20, -15, 30, -5]$. The solution of the MSS problem for the list $gs$ gives the solution of the original problem [FMMT96a]. In this example, the solution of the MSS problem is $[20, -15, 30]$, so the corresponding range of age $[17..19]$ is the optimized gain range. Generally, the problem of mining optimized gain association rules can be transformed to the MSS problem [FMMT96a].

\(^1\)The same is assumed also in [BRS99, FMMT96a].
In similar way, the optimized gain problem that allows up to \( k \) ranges can be transformed to the \( k \)-MSS problem. For details, refer to [BRS99, FMMT96a].

Here, we formally define the \( k \)-MSS problem.

\textbf{Definition 5 (\( k \)-MSS problem)} Input of the \( k \)-MSS is a list \( x_s \) of numbers, and output is up to \( k \) consecutive sublists of \( x_s \) that have the maximum sum among all the up to \( k \) consecutive sublists of \( x_s \).

For this problem, \( O(kn) \) algorithm was proposed in [BRS99], which will be explained in the next section.

\textbf{Remark}

Since the numeric attribute can have real numbers, input tuples are usually classified into the small number of buckets ordered with respect to the numeric attribute. This process is called bucketing [FMMT96a, FMMT96b]. After the bucketing, find a rule considering only ranges consisting of consecutive buckets. Generally, this means that obtained rules are approximate rules. The bucketing process takes \( O(n \log m) \) time where \( n \) is the number of tuples in database and \( m \) is the number of buckets [FMMT96b].

\subsection*{5.9.2 Constructing \( k \)-MSS Algorithm Manually}

In this section, we explain the algorithm for \( k \)-MSS developed in [BRS99]. The algorithm is a \( k \)-path algorithm, at the \( i \)-th path of which a solution of \( i \)-MSS is obtained.

\begin{itemize}
  \item \( i = 1 \): At the first path, solve 1-MSS as in [Ben84].
  \item \( i > 1 \): Let the solution of the \((i-1)\)-MSS be \( s_1, s_2, \ldots, s_i \) and the remaining sublists be \( t_1, t_2, \ldots, t_j \). Solve 1-MSS for \( t_1, t_2, \ldots, t_j \) and let one that has the maximum solution be \( t_{\text{max}} \). Solve 1-minimum segment sum problem for \( s_1, s_2, \ldots, s_i \) and let one that has the minimum solution be \( s_{\text{min}} \). If the segment sum of \( t_{\text{max}} \) plus the segment sum of \( s_{\text{min}} \) is less than 0, then split \( s_{\text{min}} \) into three subintervals with the solution of \( s_{\text{min}} \) as the middle interval and delete \( s_{\text{min}} \) from the solution of \((i-1)\)-MSS and add the first and third intervals to it, which gives the solution of \( i \)-MSS. Otherwise, split \( t_{\text{max}} \) into three subintervals with the solution of
$t_{max}$ as the middle interval and add the solution of $t_{max}$ to the solution of $(i - 1)$-MSS, which gives the solution of $i$-MSS.

This algorithm iterates $k$ times the process of finding the most effective sublist and splitting it, and its complexity is $O(kn)$ [BRS99].

For example, consider 2-MSS problem for input list $[5, -10, 20, -15, 30, -5]$. At the first path, solve 1-MSS. As a result, the sublist $[20, -15, 30]$ is obtained. At the second path, let $s_1 = [20, -15, 30]$, $t_1 = [5, -10]$, and $t_2 = [-5]$. In this case, we split $s_1$ to $[20], [-15], [30]$ and get the result $[20], [30]$.

This algorithm is smart, but its correctness is not so obvious. In fact, verifying this algorithm needs careful consideration [BRS99]. On the contrary, we will derive $O(kn)$ algorithm from simple specification, and the correctness is automatically guaranteed.

5.9.3 Deriving $k$-MSS Algorithm Automatically

In this section, we derive an $O(kn)$ algorithm for the $k$-MSS problem by specifying it as a maximum marking problem and applying the theorem proposed in [SHT01].

5.9.4 Specification

We specify this problem as a maximum marking problem: marking up the elements of a data structure with finite kinds of marks such that the marked elements meet certain property $p$ and has the maximum value with respect to certain weight function $w$. First, we determine what kind of marks we use. We use the marks $True$ and $False$, and we attach the mark $True$ to the elements that are selected as part of sublists and the mark $False$ to the others. Second, we describe property $p$. Property $p$ checks whether the number of sublists does not exceed the given $k$. This can be written as follows:
\[ p \ xs \quad = \quad p' \ xs \ (\text{False}, k) \]
\[ p' \ [\,] \ (m, e) \quad = \quad \text{True} \]
\[ p' \ (x : xs) \ (m, e) \quad = \quad \text{case } m \ \text{of} \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{True} \rightarrow \ \text{case } \text{kind} \ x \ \text{of} \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{True} \rightarrow \ \text{p' } xs \ (\text{True}, e) \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{False} \rightarrow \ \text{p' } xs \ (\text{False}, e) \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{False} \rightarrow \ \text{case } \text{kind} \ x \ \text{of} \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{True} \rightarrow \ \text{if } e > 0 \ \text{then} \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quasi
only the final result.

\[ opt_{acc} \, (f, +, 0) \, \text{accept } \phi_1 \, \phi_2 \, \delta \, [True, False] \]

where

\[ \text{accept } (c, e) = c \, \land \, e == (False, k) \]

\[ f \, x = \text{case kind } x \, \text{of} \]
\[ \text{True } \rightarrow \, w \, x \]
\[ \text{False } \rightarrow 0 \]

\[ \phi_1 \, (m, e) = True \]

\[ \phi_2 \, x \, (m, e) \, r = \]

\[ \text{case } m \, \text{of} \]
\[ \text{True } \rightarrow r \]
\[ \text{False } \rightarrow \text{case kind } x \, \text{of} \]
\[ \text{True } \rightarrow \text{if } e > 0 \, \text{then } r \]
\[ \text{else } \, False \]
\[ \text{False } \rightarrow r \]

\[ \delta \, x \, (m, e) = \]

\[ \text{case } m \, \text{of} \]
\[ \text{True } \rightarrow \text{case kind } x \, \text{of} \]
\[ \text{True } \rightarrow (True, e) \]
\[ \text{False } \rightarrow (False, e) \]

\[ \text{False } \rightarrow \text{case kind } x \, \text{of} \]
\[ \text{True } \rightarrow (True, e - 1) \]
\[ \text{False } \rightarrow (False, e) \]

The definition of the function \( opt_{acc} \) is given in Figure 4.3. The complexity is \( O(kn) \) where \( n \) is the length of the list. Algorithm obtained by derivation does not need verification, which is an important benefit of deriving efficient algorithm from specification.

5.9.6 Comparison

Here we compare the algorithm above with the algorithm developed manually in [BRS99] (See Section 5.9.2). The derived algorithm is a dynamic programming algorithm and recursive on the input list. In each step it generates \( 2k \) candidate solutions, where 2 corresponds to whether the current head element is selected or not, and \( k \) corresponds to the number of sublists in the current list. So, the derived algorithm performs \( O(k) \) operations.
\(n\) times. On the contrary, the algorithm in Section 5.9.2 is a greedy algorithm, which generates only one candidate in each step. But in each step, \(O(n)\) operations are performed. So, the algorithm in Section 5.9.2 performs \(O(n)\) operations \(k\) times. So, though order of complexity is same, these two algorithms are essentially different.

### 5.9.7 Dealing with Change of Specification

Here, consider the modified \(k\)-MSS problem with the condition that the length of each sublist must be between 5 and 10. It is not so easy to adapt the algorithm in Section 5.9.2 to this modified problem. However, by our method, the only thing we have to do is to change the property \(p\) as follows:

\[
p \; xs = p' \; xs \; (2, k) \land q \; xs
\]

The property \(q\) checks whether all the selected sublists have length between 5 and 10, which is defined as follows:

\[
q \; xs \quad = \quad q' \; xs \; (2, 0)
q' \; [] \; (m, e) \quad = \quad \text{if} \; m == 1 \; \text{then} \; 5 \leq e \land e \leq 10
\]
\[
\text{else} \; \text{True}
\]
\[
q' \; (x : xs) \; (m, e) \quad = \quad \text{if} \; \text{kind} \; x == 1 \; \text{then}
\]
\[
q' \; xs \; (1, e + 1)
\]
\[
\text{else}
\]
\[
\text{if} \; m == 1 \; \text{then} \; 5 \leq e \land e \leq 10
\]
\[
\text{else} \; \text{True}
\]

Similarly to Section 5.9.5, we obtain an \(O(kn)\) algorithm for the modified problem.

### 5.9.8 Remarks

In this section we show that a linear time algorithm for mining optimized gain association rules is derived from simple specification by reducing it to a maximum marking problem. Although a smart \(O(kn)\) algorithm is presented in [BRS99], its correctness is not easy to verify. Moreover, the algorithm is fragile to modifications of problems. On the contrary, by our method, we can systematically derive an \(O(kn)\) algorithm, and its correctness is automatically guaranteed.
By our previous method in [SHTO00], we can also derive a linear time algorithm for the same problems. However, since its specification does not allow accumulating parameters, the constant of the algorithm grows exponentially and the resulting complexity becomes $O(2^k n)$.

5.10 Features

In this section we highlight several important features of our calculational approach for solving the maximum marking problems: its simplicity, generality, and flexibility.

5.10.1 Simplicity

First, as seen in the derivation of a linear-time algorithm for solving the party planning problem as well as seen in Section 1.7, our calculation is surprisingly simple. With the optimization theorem, all we need to do is derive a mutumorphic form of the property description. Fortunately, there are a lot of handy calculation strategies for deriving mutumorphisms [HITT97], such as generalization of subexpressions to functions and fusion transformation [Jeu93, BdM96].

In the following we shall further illustrate this simplicity by solving two problems inspired by the party planning problem mentioned in the Introduction, whose derivation of linear algorithms are indeed not so straightforward for even an experienced functional programmer.

Group Organizing Problem

The inputs in the group organizing problem are trees, and we use the same data structure $\text{Org}$ that we used for the party planning problem. The property $p$ on the marked tree for this problem is that one of the ancestor nodes of any two marked nodes must be marked. This can be described
recursively by

\[
p :: \text{Org} \alpha^* \rightarrow \text{Bool}
p (\text{Leader } v []) = \text{True}
p (\text{Leader } v (t : ts)) =
\begin{align*}
& \text{if } \text{marked } v \text{ then } \text{True} \\
& \text{else if } nm t \text{ then } p (\text{Leader } v ts) \\
& \text{else } p t \land nm (\text{Leader } v ts).
\end{align*}
\]

The first equation means that a single-node tree always satisfies \( p \). The second equation means that for a tree with the root \( v \), if the root is marked, then any way of marking the nodes of its subtrees is acceptable. Otherwise, we have to check each of the subtrees to make sure that at most a single subtree has marked nodes. The function \( nm t \) checks that the tree \( t \) has no marked nodes; it is defined as follows:

\[
\begin{align*}
\text{nm} (\text{Leader } v []) &= \text{not} (\text{marked } v) \\
\text{nm} (\text{Leader } v (t : ts)) &= \text{nm } t \land \text{nm} (\text{Leader } v ts)
\end{align*}
\]

Using the calculational strategy described in Section 4.6, we can turn these functions into the following mutumorphisms on \( \text{Rtree } \alpha \), which consist of only property descriptions, and thus can obtain a linear-time algorithm for the group organizing problem.

\[
p' :: \text{RTree } \alpha^* \rightarrow \text{Bool}
p' (\text{Root } v) = \text{True}
p' (\text{Join } t_1 t_2) =
\begin{align*}
& \text{if } \text{lm} t_2 \text{ then } \text{True} \\
& \text{else if } nm t_1 \text{ then } p' t_2 \\
& \text{else } p' t_1 \land nm t_2
\end{align*}
\]

\[
\begin{align*}
\text{nm'} (\text{Root } v) &= \text{not} (\text{marked } v) \\
\text{nm'} (\text{Join } t_1 t_2) &= nm' t_1 \land nm' t_2
\end{align*}
\]
Supervisor Chaining Problem

Similarly, for the supervisor chaining problem we can specify the property that the marked nodes form a path chain as follows.

\[
p :: \text{Org} \alpha^* \rightarrow \text{Bool}
p (\text{Leader } v []) = \text{True}
p (\text{Leader } v (t : ts)) =
\]

\[
\text{if marked } v \text{ then}
\]

\[
\text{if nm } t \text{ then } p (\text{Leader } v ts)
\]

\[
\text{else marked (getLeader } t \text{) } \land p t \land
\]

\[
\text{ol (Leader } v ts)
\]

\[
\text{else if nm } t \text{ then } p (\text{Leader } v ts)
\]

\[
\text{else } p t \land \text{nm (Leader } v ts)
\]

The function \( ol \ t \) checks that the leader of the tree \( t \) is marked and the other nodes of \( t \) are not marked.

\[
\text{ol (Leader } v []) = \text{marked } v
\]

\[
\text{ol (Leader } v (t : ts)) = \text{nm } t \land \text{ol (Leader } v ts)
\]

This property \( p \) can be expressed using the following mutumorphisms which consist of only property descriptions.

\[
p' :: \text{RTree} \alpha^* \rightarrow \text{Bool}
p' (\text{Root } v) = \text{True}
p' (\text{Join } t_1 t_2) =
\]

\[
\text{if lm' } t_2 \text{ then}
\]

\[
\text{if nm' } t_1 \text{ then } p' t_2
\]

\[
\text{else lm' } t_1 \land p' t_1 \land \text{ol' } t_2
\]

\[
\text{else if nm' } t_1 \text{ then } p' t_2
\]

\[
\text{else } p' t_1 \land \text{nm' } t_2
\]

\[
\text{ol' (Root } v) = \text{marked } v
\]

\[
\text{ol' (Join } t_1 t_2) = \text{nm' } t_1 \land \text{ol' } t_2
\]

Now we can use our optimization theorem, as we did for the partying planning problem, to obtain a linear-time algorithm that solves the supervisor chaining problem.
5.10.2 Generality

Our approach is general (polytypic) enough to deal with maximum marking problems on data structures that are not lists or trees. To illustrate this, we derive a linear-time algorithm solving the maximum two disjoint paths problem on series-parallel graphs [TNS82].

The series-parallel graph is defined as follows:

\[
SPG ::= Base (Vert, Vert, Edge) \\
  | Series SPG SPG \\
  | Parallel SPG SPG
\]

Here Vert represents the type of vertices and Edge represents the type of edges. Every graph should have a single source and a single sink. The two data constructors are Series and Parallel. Series \(g_1 g_2\) makes sense only when the sink of \(g_1\) is the source of \(g_2\), and Parallel \(g_1 g_2\) makes sense only when \(g_1\) and \(g_2\) share a source and a sink.

Figure 5.1 shows the meaning of the constructors. For example, the middle graph in Figure 5.1 is represented by

\[
g = Series (Base (u, w, e1)) (Base (w, v, e2)).
\]

The maximum two disjoint paths problem is defined as follows: when given two vertices \(s\) and \(t\), find two disjoint paths between the two vertices \(s, t\) such that the sum of the edge weights is maximum. Since the two disjoint paths between the vertices \(s\) and \(t\) can be seen as a cycle containing the vertices \(s\) and \(t\), we can specify the property \(p\) of the problem by

\[
p :: \text{Vert} \rightarrow \text{Vert} \rightarrow SPG \rightarrow \text{Bool} \\
p s t g = \text{cycle } g \land \text{th } s g \land \text{th } t g.
\]

Here cycle judges whether the marked edges in the graph form a cycle, and th \(v\) judges whether there is a marked edge incident to the vertex \(v\). These
two functions can be defined as follows:

\[
\begin{align*}
\text{cycle} \ (\text{Base} \ e) &= \text{not} \ (\text{marked} \ e) \\
\text{cycle} \ (\text{Series} \ g_1 g_2) &= (\text{cycle} \ g_1 \land \text{nm} \ g_2) \lor \\
&\quad (\text{nm} \ g_1 \land \text{cycle} \ g_2) \\
\text{cycle} \ (\text{Parallel} \ g_1 g_2) &= (\text{cycle} \ g_1 \land \text{nm} \ g_2) \lor \\
&\quad (\text{nm} \ g_1 \land \text{cycle} \ g_2) \lor \\
&\quad (\text{span} \ g_1 \land \text{span} \ g_2) \\
\end{align*}
\]

\[
\text{th} \ v \ g \\
= \ \text{anyMarked} \ (\text{inc} \ v \ g)
\]

The function \text{span} judges whether the marked edges in the graph span between the source and sink of the graph, \text{nm} judges whether or not there are marked edges in the graph, \text{anyMarked} takes as its argument a list of edges \(es\) and judges whether or not there is a marked edge in \(es\), and \text{inc} \(v\) \(g\) gathers all the edges of \(g\) which are incident to the vertex \(v\).

\[
\begin{align*}
\text{span} \ (\text{Base} \ e) &= \text{marked} \ e \\
\text{span} \ (\text{Series} \ g_1 g_2) &= \text{span} \ g_1 \land \text{span} \ g_2 \\
\text{span} \ (\text{Parallel} \ g_1 g_2) &= (\text{span} \ g_1 \land \text{nm} \ g_2) \lor \\
&\quad (\text{nm} \ g_1 \land \text{span} \ g_2) \\
\text{nm} \ (\text{Base} \ e) &= \text{not} \ (\text{marked} \ e) \\
\text{nm} \ (\text{Series} \ g_1 g_2) &= \text{nm} \ g_1 \land \text{nm} \ g_2 \\
\text{nm} \ (\text{Parallel} \ g_1 g_2) &= \text{nm} \ g_1 \land \text{nm} \ g_2 \\
\text{inc} \ v \ (\text{Base} \ e@\ (v_1, v_2, \ldots)) &= \text{if} \ v == v_1 \lor v == v_2 \\
&\quad \text{then} \ [e] \ \text{else} \ [] \\
\text{inc} \ v \ (\text{Series} \ g_1 g_2) &= \text{inc} \ v \ g_1 ++ \text{inc} \ v \ g_2 \\
\text{inc} \ v \ (\text{Parallel} \ g_1 g_2) &= \text{inc} \ v \ g_1 ++ \text{inc} \ v \ g_2
\end{align*}
\]

By fusion calculation, we can get the following efficient recursive definition for \text{th}.

\[
\begin{align*}
\text{th} \ v \ (\text{Base} \ e@\ (v_1, v_2, \ldots)) &= \text{if} \ v == v_1 \lor v == v_2 \\
&\quad \text{then} \ \text{marked} \ e \ \text{else} \ \text{False} \\
\text{th} \ v \ (\text{Series} \ g_1 g_2) &= \text{th} \ v \ g_1 \lor \text{th} \ v \ g_2 \\
\text{th} \ v \ (\text{Parallel} \ g_1 g_2) &= \text{th} \ v \ g_1 \lor \text{th} \ v \ g_2
\end{align*}
\]

Now the property description \(p \ s \ t\) is represented as mutumorphisms with property descriptions \text{cycle}, \text{span}, \text{nm}, \text{th} \(s\), and \text{th} \(t\). It follows from our optimization theorem that a practical linear-time algorithm can be obtained.
5.10.3 Flexibility

Here we explain flexibility of our derivation in coping with modification of the specifications of problems. We choose as our example the maximum segment sum problem, which is to compute the maximum of the sums of all segments (contiguous sublist) of a list. This is a quite well-known problem in the program calculation community [Bir89]. In the following we will not only give a new solution to it but will also demonstrate that we can straightforwardly solve a set of related problems that would not be easily solved by using the previously available approaches.

The maximum segment sum problem is actually a maximum marking problem where the property is that all marked elements in a list should be adjacent (connected). This property can be specified as follows:

\[
\begin{align*}
\text{conn} [x] &= \text{True} \\
\text{conn} (x : xs) &= \text{if marked } x \text{ then } \text{nm } xs \vee (\text{marked} (\text{hd } xs) \land \text{conn } xs) \\
&\quad \text{else conn } xs
\end{align*}
\]

If the list contains only a single element, then the property is satisfied. Otherwise, if the head is marked, then either none of the other elements are marked or all marked elements are connected to the head. If the head is not marked, the remaining list is checked recursively. As before, the function \(\text{nm}\) judges whether or not any of the elements are marked.

\[
\begin{align*}
\text{nm} [x] &= \text{not} (\text{marked } x) \\
\text{nm} (x : xs) &= \text{not} (\text{marked } x) \land \text{nm } xs
\end{align*}
\]

To transform \(\text{conn}\) into mutumorphisms which consist only of property descriptions, we generalize the part \(\text{marked} (\text{hd } xs)\) and let it be \(\text{mh } xs\).

\[
\text{mh } xs = \text{marked} (\text{hd } xs)
\]

By a simple fusion calculation, we can easily get the following definition of \(\text{mh}\).

\[
\begin{align*}
\text{mh} [x] &= \text{marked } x \\
\text{mh} (x : xs) &= \text{marked } x
\end{align*}
\]
So we have obtained the following mutumorphisms which consist only of property descriptions.

\[
\begin{align*}
\text{conn } [x] &= \text{True} \\
\text{conn } (x : xs) &= \begin{cases} 
\text{if marked } x \text{ then} \\
\quad \text{nm } xs \lor (\text{mh } xs \land \text{conn } xs) \\
\text{else } \text{conn } xs
\end{cases}
\end{align*}
\]

Application of the optimization theorem gives us a linear-time algorithm. Although our linear algorithm may use a few more operations than that described by Bird [Bir89], it is much easier to derive.

Now consider an extension of the maximum segment sum problem where we are interested only in those segments containing only even numbers. The property for this extended problem is

\[
p xs = \text{conn } xs \land \text{evens } xs
\]

where \(\text{evens}\) can be defined by

\[
\begin{align*}
\text{evens } [x] &= \begin{cases} 
\text{if marked } x \text{ then } \text{even } (\text{weight } x) \\
\text{else } \text{True}
\end{cases} \\
\text{evens } (x : xs) &= \begin{cases} 
\text{if marked } x \text{ then} \\
\quad \text{even } (\text{weight } x) \land \text{evens } xs \\
\text{else } \text{evens } xs
\end{cases}
\end{align*}
\]

It is easy to see that \(p\) can be defined as mutumorphisms with property descriptions \(\text{conn, evens, nm,}\) and \(\text{mh,}\) and thus that we obtain a linear algorithm for solving this extended problem.

Following this line, we can consider many similar segment problems. We can, for example, consider maximum sums of segments that have even numbers of elements by defining the following property

\[
p xs = \text{conn } xs \land \text{even } (\text{mnum } xs),
\]

where \(\text{mnum } xs\) is the number of marked elements in \(xs\). To get mutumorphisms which consist only of property descriptions, we generalize the part \(\text{even } (\text{mnum } xs)\) and let it be \(\text{em } xs\).

\[
\text{em } xs = \text{even } (\text{mnum } xs)
\]
After deriving a recursive form for $em$ by using calculations similar to those described above, we can apply the optimization theorem to get a linear-time algorithm. It should be noted that deriving a linear-time algorithm for this problem requires some creativity when using the approach described in [Bir89, Jeu93] because the property $p$ is not prefix-closed. This makes filter promotion difficult.

Finally, we mention that maximum segment sum problem can be generalized to trees: a segment in a list is simply generalized to a set of connected nodes in a tree. It should be easy for reader to derive a linear-time algorithm to solve this generalized problem.
Chapter 6

Automatic Generation of Efficient Programs

In previous chapters, we show that by our proposing method efficient linear time algorithms for MMP can be derived mechanically. We have showed that our method is quite general and powerful. In this chapter, we show that our method can be implemented as a generation system to automatically generate efficient linear time algorithms for MMP from simple specifications. It was known that generation of linear time algorithms for MMP (precisely maximum weightsum problem) can be done by the method proposed by Borie et al., [BPT92]. Though his method is theoretically appealing, it generates a prohibitively large table (see Section 3.3 and Chapter 7 for details) and thus cannot be put to practical use. For each individual problems, a practical linear time algorithm can be derived by calculation. For example, for a famous instance of MMP, the maximum segment sum problem, a practical linear time algorithm derivation was known [Bir89]. But any method for automatically deriving practical linear time algorithms has not been known for MMP. Our method we propose in this thesis enables it. We implement our proposing method and show that it generates practical linear time algorithms by several examples [YSHT01].

6.1 MAG system

Before we show our implementation, we show that our method can be implemented using an existing transformation system called MAG [dMS98].
Though by using MAG system one can automatically generate efficient algorithms for MMP, one cannot specify the transformation strategy, which is rather inconvenient for describing transformation. So in the next section we implement a flexible system for MMP.

**6.1.1 Implementation**

Our generation rule can be implemented, so that efficient programs can be generated automatically. In this section, we highlight how we can do so using MAG system [dMS98], a transformation system with a powerful higher order pattern matching.

As seen in Figure 4.4, our obtained program can be divided into two parts: the dynamic and static parts. The dynamic part changes from problems to problems, while the static part remains the same. In Figure 4.4, the upper part is dynamic and the lower is static. We show how to generate the dynamic part from the specification \( mmp w p ms \).

Using MAG, we may code the generation of the dynamic part from specification \( mmp w p ms \) as a rule called \texttt{mmpRule} and a rule called \texttt{fusion} as follows:

```plaintext
mmpRule: mmp wfun p ms
  = optacc (fun,oplus,e) (\(c,e) -> c && e==e0)
    phi1 phi2 delta ms,
  if {wfun = \xs -> foldr oplus e (map fun xs);
      p = \xs -> foldrh (phi1, phi2) delta xs e0};

fusion: f (foldr step e xs) = foldrh (phi1', phi2')
  delta xs,
  if {f e = phi1';
      \ y ys acc -> f (step y ys) acc =
        \ y ys acc -> phi2' y acc
        (f ys (delta y acc))}
```

Now for the coloring problem, we can apply this rule to the following specification and obtain a linear time program like in Figure 4.4 automatically.

```plaintext
coloring: coloring = mmp wf indep [1,2,3];
```
wf:  wf xs = (foldr (+) 0 (map w xs));
w:  w x = if kind x == 1 then weight x
         else if kind x == 2 then (-1) * (weight x)
         else 0;
indep: indep xs = indep' (foldr (:) [] xs) 0;

indep'1: indep' [] color = True;
indep'2: indep' (x:xs) color = kind x /= color &&
indep' xs (kind x);

classlist: classlist = [False,True];
acclist: acclist = [0..3]

With these, the MAG system can produce the linear time program as
given in Figure 4.4. The above corresponds to Theorem 4. In the following,
we show the case for Theorem 3 using the maximum independent-sublist sum
problem. The rule that corresponds to Theorem 3 can be coded as follows:

mmpRule: mmp wfun p ms
            = opt (fun, oplus, e) accept phi1 phi2 ms,
              if {wfun = \ xs -> foldr oplus e (map fun xs);
               p = \ ys -> accept (foldr phi2 phi1 ys)};

split2: split2 f1 f2 x = (f1 x, f2 x);
uncurry2: uncurry2 e = \ x (a1, a2) -> e x a1 a2;
tupling2: split2 f1 f2 (foldr (:) [] xs)
          = foldr (uncurry2 g) c xs,
            if {split2 f1 f2 [] = c;
                 \ x xs -> split2 f1 f2 (x:xs) =
                 \ x xs -> g x (f1 xs) (f2 xs)}

Now for the mis problem, we can apply this rule to the following specifica-
tion and obtain a linear time program like in Figure 1.1 automatically.

mis: mis = mmp w ind [True, False];
w: w = (foldr (+) 0) . map (\ (x,m) -> if m then x else 0);
ind: ind = (\(a1,a2) -> a1) .
      split2 ind0 ind1 . foldr (:) [];
ind01: ind0 [] = True;
ind02: ind0 (x:xs) = if marked x then ind1 xs
    else ind0 xs;
ind11: ind1 [] = True;
ind12: ind1 (x:xs) = if marked x then False
    else ind0 xs

As seen above, we can implement our proposing method using MAG, but the current version of the MAG system has several restrictions such as not allowing case expression. Moreover, the ordering of rules affects the derivation. So we make a system by extending the MAG system, which we name ”Yicho System”.

6.2 Yicho System

In this section we implement a system for generating efficient linear time algorithms for MMP, which enables flexible description of transformation. We would like to use a language for describing rules and strategy of generation. Generally speaking, firstly an effective generation rule is a rule which is general enough to be able to match many program pattern. Secondly an effective generation rule is constructive and can be applied in practical time. For example, the rule ”if there exist two natural numbers \( p, q \) that satisfies \( n = p \times q \), then rewrite \( n \) into \( p \times q \)” is not constructive. By using a language for describing transformation strategy, one can specify how to apply transformation rules [HT99, TH00, Vis01]. Though there exist systems which does not have such language to describe strategy like MAG system [dMS98], it becomes difficult to perform derivation as user intended. So a language for describing strategy is desired.

In the following, we propose a language for describing transformation strategy for generating practical linear time algorithms for MMP and implement it. The system has the following features.

- Our transformation rule is general enough to be applied to MMP and thus it can be applied to many problems. It is constructive and can be easily implemented.
- We use a meta language, called ”calculation carrying program (CCP for short)” [TH00], for describing transformation strategy. Previously available transformation description languages [Vis01] have only first
order matching and thus restrict the rules which can be directly described. In contrast, CCP has higher order matching, it enlarge the rules which can be directly described. Another feature of CCP is description about control of transformation. For example, it is possible to apply rules when certain condition satisfied, to repeat application of some rule, or to use some strategy in another strategy. Description of strategy in intuitive form is one reason for CCP to be appropriate to describe strategy.

- In our system, one can compile CCP to script, which is an intermediate language in our system. Environment to execute obtained script is available, which makes it easy to debug.

6.2.1 Structure of automatic generation system

In this section, we show a structure and implementation of the automatic generation system for MMP, which is implemented using a functional programming language Haskell [PJH99, Bir98].

Structure of the system

The system consists of two layers. The above layer is for high level description of transformation strategy, which is described by a language based on CCP [TH00] (Section 6.2.2). CCP has higher order matching, so it is possible to describe a wide range of transformation rules. The lower layer is for description of transformation steps, which is described by a script language. After checking the syntax and type of CCP, CCP is transformed into the script language (Section 6.2.3).

CCP is appropriate to describe high level transformation, so a strategy described in CCP can be reused for several programs by no change or small change. Script is for low level transformation, so it can be reused for exactly same structure programs.

Lower level provides an interactive environment, which is for step by step execution, undo of a transformation, showing bindings of variables, showing rules that can be applied to the current expression, and so on (Section 6.2.2).

CCP and script language is independent, so, it is possible to use a script that is output of some other system.
6.2.2 Calculation Carrying Program (CCP)

CCP is a program accompanied by calculation rules [HT99]. Program calculation is transforming program to efficient one by applying calculation rules. Generally speaking, program calculation is done by hand and calculation rules and how to apply them varies according to object program. When calculating programs on a computer, how to calculate should be written in some way. CCP gives one way for that. CCP is a program accompanied by calculation rules and how to apply them, so programmer can specify how to calculate efficient programs. General rules such as fusion rule or tupling rule are reusable once it is written. Furthermore, CCP has higher order matching, so it is possible to describe calculations directly.

An Example — sumsq

Here, we show what is CCP by using a problem to get sum of square of list of numbers. This problem can be solved by squaring each element and then summing up. By letting the square function be $sq$ and the sum function be $sum$,

$$sumsq = sum \circ map \ sq$$

gives one solution to this problem. This generates an intermediate list which is the result of $map \ sq$, but it is possible to describe it not to generate this intermediate list. We would like to transform the above function $sumsq$ to one which does not generate this intermediate list. This can be done by the fusion transformation.

**Theorem 5 (Fusion Rule)** If for any $a, x$

$$g \ a \ (f \ x) = f \ (step \ a \ x)$$

holds, then

$$f \circ foldr \ step \ e = foldr \ g \ (f \ e)$$

holds. \qed

The function $foldr$ is a list folding function, which is defined as follows:

- $foldr \ step \ e \ [] = e$
- $foldr \ step \ e \ (x : xs) = step \ x \ (foldr \ step \ e \ xs)$
In order to apply this fusion theorem, we change \( \text{map} \ sq \) into the form of \( \text{foldr} \ step \ e \).

\[
\text{map} \ sq = \text{foldr} \ step \ [] \\
\text{where} \ step \ y \ ys = (sq \ y) : ys
\]

Next, we should find function \( g \) that satisfies the following equation:

\[
g \ y \ (\text{sum} \ ys) = \text{sum} \ (\text{step} \ y \ ys)
\]

The following function \( g \) satisfies this equation.

\[
g \ y \ s = sq \ y + s
\]

Since \( \text{sum} \ [] = 0 \), by applying the fusion theorem, we get

\[
\text{sum} \circ \text{map} \ sq = \text{foldr} \ g \ 0
\]

This function \( \text{foldr} \ g \ 0 \) does not produce an intermediate list when computing square sum, so efficiency is improved by this transformation.

When doing this kind of calculation on a computer, we should describe how to calculate in some way. CCP gives one way of describing it. In CCP, we accompany object program with calculation. For example, CCP for the \( \text{sumsq} \) problem is written as in Figure 6.1.

CCP can be divided into three parts. In Figure 6.1, the first part is object program, the second part is calculation rules, \( \langle \text{fusion}, \text{applyFusion} \rangle \), and the last part is to specify how an efficient function \( \text{sumsqOpt} \) should be obtained from the object program \( \text{sumsq} \).

The first part is the specification of the \( \text{sumsq} \) problem. Though this specification generates an intermediate list when computing square sum, we can transform it to efficient one that does not generates the intermediate list. The second part describe this transformation process. The rule \( \langle \text{fusion} \rangle \) is an implementation of the fusion theorem (Theorem 5). When applying the fusion rule, the right function \( g \) in the function composition \( f \ . \ g \) should be described in the form \( \text{foldr} \ step \ e \). For that purpose, we define the calculation rule \( \text{applyFusion} \).

In the rule \( \text{applyFusion} \), the right most function in the given function composition is rewrited into the form \( \text{foldr} \ step \ e \) by fusing with identity function \( \text{foldr} \ (:) \ [] \) and then the fusion rule \( \langle \text{fusion} \rangle \) is applied successively. By applying this calculation rule \( \text{applyFusion} \) to the expression \( \text{sum} \ . \ \text{map} \ sq \), we get the form that does not generate an intermediate list.
---------- Object Program ----------
sumsq = sum . map sq;
sum [] = 0;
sum (_x:_xs) = _x + sum _xs;
sq _x = _x * _x;
map _f [] = [];
map _f (_x:_xs) = f _x : map _f _xs;
foldr _step _e [] = _e;
foldr _step _e (_a:_x) = _step _a (foldr _step _e _x);
(_f . _g) _x = _f (_g _x);

---------- Calculation Rules ----------
<fusion> _f (foldr _step _e _xs) =
  letm _g a (_f x) = _f (_step a x);
  _c = _f _e
  in foldr _g _c _xs;

<applyFusion> (_f . _g) = <fusion> (_f . <applyFusion> _g);
<applyFusion> _f = <fusion> (_f . foldr (:) []);

---------- Application of Calculation Rule ----------
sumsqOpt = <applyFusion> (<unfold> sumsq)

Figure 6.1: An Example of CCP — sumsq
The last part describes that. First the definition of \texttt{sumsq} is unfolded by the rule \texttt{<unfold>} and then \texttt{applyFusion} is applied to that. As a result, the function \texttt{sumsqOpt} which does not generate an intermediate list is obtained. The rule \texttt{<unfold>} is a built-in calculation rule that unfolds a definition of function.

In CCP, one can describe function definitions and calculation rule definitions together. We distinguish function from rule by surrounding the name of calculation rule. Furthermore, in a definition of calculation rule, another calculation rule or itself can be called. Allowing this kind of description enables it to describe complex calculation rule by combining basic calculation rules as seen in the rule \texttt{<applyFusion>.

In CCP, the name of pattern variables starts by \texttt{_} in order to distinguish them from local variables. For example, in the matching of \texttt{letm} expression

\begin{verbatim}
  _g a (_f x) = _f (_step a x)
\end{verbatim}

\texttt{a} and \texttt{x} does not start with \texttt{_}, which means that these are not pattern variables but local variables. The locality means that the equation

\begin{verbatim}
  _g a (_f x) = _f (_step a x)
\end{verbatim}

holds for any \texttt{a} and \texttt{x} with appropriate types.

\textbf{The language for describing CCP}

We define the language for describing CCP as in Figure 6.2.

In CCP, we can describe function definitions (\texttt{funDef}) and rule definitions (\texttt{ruleDef}) together. We can also use, in an expression, rule applications, meta lambda abstractions, meta let expressions, meta case expressions where higher order pattern matching \cite{dMS99} is performed. When pattern matching is performed, substitutions for pattern variables are obtained. We define the meaning of expressions including higher order pattern matching as follows:

\begin{itemize}
  \item Meta let Expression
    \begin{verbatim}
    letm e_{p1} = e_{b1}; \ldots; e_{pn} = e_{bn} in e
    \end{verbatim}
  \end{itemize}

Obtain substitutions for pattern variables in expressions \(e_{p_i}\), \ldots, \(e_{p_n}\) by a higher order pattern matching algorithm \cite{dMS99}, evaluate the expression \(e\) with the obtained substitutions, and finally let the obtained value of \(e\) as the value of the whole meta let expression.

85
<table>
<thead>
<tr>
<th>CCP:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{ccp ::= def}_1; \ldots; \text{def}_n ] Definitions</td>
</tr>
<tr>
<td>Definition:</td>
</tr>
<tr>
<td>[ \text{def ::= funDef Function Definition} ]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Function Definition:</td>
</tr>
<tr>
<td>[ \text{funDef ::= } f \ pats = e \text{ Function Definition} ]</td>
</tr>
<tr>
<td>Rule Definition:</td>
</tr>
<tr>
<td>[ \text{ruleDef ::= } \langle r \rangle \ e_p = e_b \text{ Rule Definition} ]</td>
</tr>
<tr>
<td>Expression:</td>
</tr>
<tr>
<td>[ e ::= \text{haskellExp Expression of Haskell} ]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Expression including higher order pattern matching:</td>
</tr>
<tr>
<td>[ \text{em ::= letm } e_p = e_b; \ldots; e_p_n = e_b_n \text{ in } e \text{ Meta let Expression} ]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Figure 6.2: Definition of CCP
• Meta case Expression

\[
\text{case } e \text{ of } e_{p_1} \rightarrow e_1; \ldots; e_{p_n} \rightarrow e_n
\]

Perform higher order pattern matching with respect to the expression \( e \) with the expressions \( e_{p_1}, \ldots, e_{p_n} \) using the higher order pattern matching algorithm \([dMS99]\). Obtain substitutions for the expression \( e_{p_i} \) that is the first expression to succeed in matching, evaluate the expression \( e_i \) with the obtained substitutions, and finally let the obtained value of \( e_i \) as the value of the whole meta case expression.

• Rule Application \( \langle r \rangle \ e \)

Apply the rule \( r \) to the expression \( e \). When the rule \( r \) is defined as

\[
\langle r \rangle \ e_p = e_b
\]

perform higher order pattern matching with respect to the expression \( e \) with the expression \( e_p \) using the higher order pattern matching algorithm \([dMS99]\) to obtain substitutions for pattern variables in \( e_p \), evaluate the expression \( e_b \) with the obtained substitutions, and finally let the obtained value of \( e_b \) as the value of the whole rule application expression.

In the definition of CCP, \( \text{haskellExp} \) represents expressions of the functional language Haskell \([PJH99, Bir98]\), we implement part of them in the current version of the system. We adopt that the name of a pattern variable starts with underscore \( _ \) and the one of the other kind of variables does not start with \( _ \). This distinction is for distinguish local variables from pattern variables in letm expressions.

\textbf{script}

We define the syntax of script as in Figure 6.3. Script is a description of transformation steps. By using script, we can execute transformation interactively for debugging. Script is divided into three parts, commands for setting environment, commands for doing transformation, and commands for debugging.

There are four kinds of commands for setting environment. Command \textit{setPath} is for specifying path for the file of CCP. Command \textit{loadTheory} is for loading rules. Command \textit{setExp} sets an expression as starting expression.
Script

\[ Command ::= EnvComm \quad \text{Command for setting environment} \]
\[ \quad | \quad IntComm \quad \text{Command for doing transformation} \]
\[ \quad | \quad DebugComm \quad \text{Command for debugging} \]

Command for setting environment

\[ EnvComm ::= \quad \text{setPath PathName} \quad \text{Specifying path of file of ccp} \]
\[ \quad | \quad \text{loadTheory TheoName} \quad \text{Loading transformation rules} \]
\[ \quad | \quad \text{setExp ExpName} \quad \text{Setting starting expression} \]

Command for doing transformation

\[ IntComm ::= \quad \text{step LawName Path} \quad \text{rule application} \]
\[ \quad | \quad \text{beginDerivation LawName Path} \quad \text{starting local derivation} \]
\[ \quad | \quad \text{match} \quad \text{obtaining substitution by pattern matching} \]
\[ \quad | \quad \text{endDerivation} \quad \text{ending local derivation} \]
\[ \quad | \quad \text{createRule RuleName} \quad \text{creating a new rule} \]

\[ Path ::= \quad \text{Loc}_1 \text{Loc}_2 \cdots \quad \text{path for subexpression} \]

\[ Loc ::= \quad B \quad \text{body of a \lambda expression} \]
\[ \quad | \quad F \quad \text{function part of function application} \]
\[ \quad | \quad A \quad \text{argument part of function application} \]

Command for debugging

\[ DebugComm ::= \quad \text{help} \quad \text{displaying help} \]
\[ \quad | \quad \text{showDerivation} \quad \text{showing derivation up to the present} \]
\[ \quad | \quad \text{quit} \quad \text{quitting} \]
\[ \quad | \quad \text{showBindings} \quad \text{showing bindings of local variables} \]
\[ \quad | \quad \text{showStack} \quad \text{showing all the local derivations} \]
\[ \quad | \quad \text{undo} \quad \text{undoing a transformation} \]
\[ \quad | \quad \text{showSubExpressions} \quad \text{showing all the subexpressions} \]
\[ \quad | \quad \text{showTheory} \quad \text{showing all the rules} \]

Figure 6.3: The syntax of script
Commands for doing transformation are divided into five kinds. Command \texttt{step} takes a rule and location of subexpression in the current expression and applies the rule to the subexpression indicated by the location. Location of a subexpression is written by a sequence of $B$ or $F$ or $A$ where $B$ represents body of lambda expression, $F$ represents function part of function application, and $A$ represents argument part of function application.

Command \texttt{beginDerivation} takes a rule and location of subexpression and starts sub-derivation. Command \texttt{match} match the pattern of condition and current expression, and memorize the obtained substitution. Command \texttt{endDerivation} ends the sub-derivation and substitute the pattern variables in the right hand side of the rule using the obtained substitutions in the sub-derivation, and substitute the subexpression with the right hand side of the rule substituted.

Command \texttt{createRule} adds a rule that transforms the starting expression to the current expression with a specified rule name.

Commands for debugging are used when debugging. Command \texttt{help} displays commands currently executable. Command \texttt{showDerivation} displays the derivation up to the present. Command \texttt{quit} quits the session even if there are commands not executed yet below the command. Command \texttt{showBindings} displays local variables in the current expression. Command \texttt{showStack} displays several informations before entering the current sub-derivation and the current sub-derivation. Command \texttt{undo} undo one step. Command \texttt{showSubExpressions} displays all the subexpressions of the current expression. Command \texttt{showTheory} displays all the rules that can be applied.

Here we show an example of script, which is for improving efficiency of a function \texttt{sumsq} that computes sum of square of each elements. The script is shown in Figure 6.4. A number and a colon at the head of each line is not included in the script, only for explanation. In the first line, \#! represents that this file is script. In the second line, the command \texttt{setPath} specifies directory of file in which CCP is written. In the third line, the command \texttt{loadTheory} loads the file \texttt{sumsq} where calculation rules are described. In the fourth line, the command \texttt{setExp} sets the expression \texttt{sumsqOpt} as the starting expression. In the fifth to eighth lines, the command \texttt{step} transforms expression. For example, in the fifth line, the rule \texttt{sumsq} is applied to the function part of function application in the body of lambda expression, which is the current expression. Expressions are represented in the \eta-expanded form in the system, so the expression \texttt{sumsq} is represented as $\lambda x . \texttt{sumsq} x$. The
1: #!

2: setPath "~/ysys/examples"
3: loadTheory "sumsq"
4: setExp "sumsqOpt"

5: step "sumsqOpt" "BF"
6: step "sumsq" "BF"
7: step "applyFusion1" "BF"
8: step "applyFusion2" "BFA"

9: beginDerivation "fusion" "BFABF"
10: step "map2" "BB"
11: match
12: step "map1" ""
13: match
14: endDerivation

15: beginDerivation "fusion" "BF"
16: step "sum2" "BB"
17: match
18: step "sum1" ""
19: match
20: endDerivation

21: showDerivation
22: quit

Figure 6.4: An example of script – sumsq.ys
body of the $\lambda$ expression is $\text{sumsq } xs$ and the function part of it is $\text{sumsq}$. Since this expression $\text{sumsq}$ matches with the left hand side of the rule $\text{sumsq}$ in the second line of 6.5, By applying the rule $\text{sumsq}$, we obtain the following expression.

$$\lambda xs . \text{sum } (\text{map } \text{sq } xs)$$

This expression is displayed in the $\eta$-contracted form as in the fifth line of Figure 6.6. In the ninth line, the command $\text{beginDerivation}$ starts sub-derivation for the rule $\text{fusion}$. In the sub-derivation, by transforming left hand side to right hand side in each conditional equation in the $\text{if}$ clause, we get substitution for pattern variables. By using the obtained substitutions, the rule $\text{fusion}$ is applied. Here, by the command $\text{match}$ substitutions are obtained and by the command $\text{endDerivation}$ the sub-derivation is ended. In the 21th line, the command $\text{showDerivation}$ displays the derivation up to the present, which is shown in Figure 6.6. In the 22th line, the command $\text{quit}$ finishes the session.

Transformation rules are written as in Figure 6.5. The part surrounded by "{-}" and "-}" is comment. Rules are separated by a semicolon ";". In each rule definition, rule name is written before colon ";" and the remaining part represents a rule where left hand side and right hand side are separated by equal "=". The $\text{if}$ clause is used when describing conditional rule. Conditional equations are separated by semicolon ";".

The result of execution of the above script is as in Figure 6.6.

### 6.2.3 Execution of CCP and transformation to script

In this section, we show how to execute CCP. Script is record of execution of CCP. So, executing CCP is also a transformation of CCP to script.

When executing CCP, as a preprocess, CCP is transformed into the form that each rule does not call another rule. Each function definition is simply transformed into rule to unfold the definition. Definition of calculation rule is transformed into the form where rule applications in the right hand side of it are removed. These transformations are performed on expressions including higher order pattern matching as follows:

- Meta let expression
  
  ```
  \text{letm } e_p = e_b; \ldots; e_p = e_b \text{ in } e
  ```

  This is transformed to a conditional expression as follows:
{- sumsq.eq -}

```
sumsq : sumsq = sum . map sq;
sum1 : sum [] = 0;
sum2 : sum (x:xs) = x + sum xs;
sq : sq x = x * x;
map1 : map f [] = [];
map2 : map f (x:xs) = f x ; map f xs;
foldr1 : foldr step e [] = e;
foldr2 : foldr step e (a:x) = step a (foldr step e x);

fusion : _f (foldr _step _e _x) = foldr _g _c _x,
       if{ \ x a -> _f (_step a x) = \ x a -> _g a (_f x);
           _f _e = _c };

applyFusion1: _f . _g = _f . _g;
applyFusion2: _f = _f . foldr (:) [];

sumsqOpt: sumsqOpt = sumsq
```

Figure 6.5: Rule definitions – sumsq.eq
\begin{verbatim}
sumsqOpt
= { sumsqOpt }

sumsq
= { sumsq }
  sum . map sq
= { applyFusion1 }
  sum . map sq
= { applyFusion2 }
  sum . (map sq . foldr (:) [])
= { fusion
    (\ a x -> map sq (a : x ))
= { map2 }
    (\ b d -> sq b : map sq d)

  map sq []
= { map1 }
  []
}

sum . foldr (\ g -> (:) (sq g)) []
= { fusion
    (\ a x -> sum (sq a : x))
= { sum2 }
    (\ b c -> sq b + sum c)

  sum []
= { sum1 }
  0
}
foldr (\ f -> (+) (sq f)) 0
\end{verbatim}

Figure 6.6: Result of derivation – sumsq.out
e, if \{eb1 = ep1; 
  eb2 = ep2; 
  ... 
  ebn = epn\} 

In equations for matching, left hand side and right hand side are swapped, which is for writing pattern in the right hand side in a rewriting rule.

- Meta case expression
  \texttt{casem} \texttt{e} \texttt{of} \ e_p_1 \to e_1; \ldots; e_p_n \to e_n 

The following \( n \) rules are generated and the meta case expression is transformed into \( e \).

\begin{align*}
  ep1 &= e1; \\
  ep2 &= e2; \\
  ... \\
  epn &= en \\
\end{align*}

CCP is executed by applying the obtained rewriting rules in the order specified in CCP. The record of applied rules and location to which rules are applied is script. We show the process of execution of CCP by an example in the following.

Here as an example, we transform CCP for the \texttt{sumsq} problem to script. CCP for the \texttt{sumsq} problem is shown in Figure 6.1. We execute the CCP according to the procedure given above, as a result of which script is obtained. Firstly as a preprocess, each function definitions are transformed into the rule which unfolds the definition itself. Each calculation rules are transformed into the rule that is obtained by removing rule applications in the right hand side mechanically. As a result, we obtain the rewriting rules as in Figure 6.7.

The rule \texttt{applyFusion1} rewrite nothing since the original rule does not do other than application of rules \texttt{<fusion>} and \texttt{<applyfusion>}. 

Next, the obtained rewrite rules in Figure 6.7 are applied in the order specified in CCP in Figure 6.1. The calculation process is as in Figure 6.6. Firstly \texttt{sumsqOpt} is transformed to \texttt{sumsq} and then it is unfolded by \texttt{<unfold>}. The rule \texttt{<applyFusion>} is applied to it when \texttt{sum . map sq} is tested for matching against \texttt{_f . _g} and the rule \texttt{applyFusion1} is applied. Here nothing is changed since nothing other than rule application is done. Later the rule \texttt{applyFusion} will be applied to \texttt{map sq} and then the
sumsq: sumsq = sum . map sq;
sum1: sum [] = 0;
sum2: sum (x:xs) = x + sum xs;
sq: sq x = x * x;
map1: map f [] = [];
map2: map f (x:xs) = f x : map f xs;
foldr1: foldr step e [] = e;
foldr2: foldr step e (a:x) = step a (foldr step e x);
compose: (_f . _g) _x = _f (_g _x);
fusion: _f (foldr _step _e _xs) = foldr _g _c _xs,
  if{ _f (_step a x) = _g a (_f x);
       _f _e = _c}
applyFusion1: _f . _g = _f . _g;
applyFusion2: _f = _f . foldr (:) [];
sumsqOpt: sumsqOpt = sumsq;

Figure 6.7: Rewriting rules for sumsq
rule <fusion> will be applied. Next the rule <applyFusion> is applied to map sq, but map sq does not match with \_f . \_g. applyFusion2 is applied. As a result sum . (map sq . foldr (:) []) is obtained. Later the rule <fusion> will be applied to (map sq . foldr (:) []). After that the rule <fusion> will be applied twice and an efficient program that does not produce any intermediate list is obtained.

By recording the applied rules and locations they are applied as script form, we get a script as in Figure 6.4.

### 6.2.4 Example of maximum marking problems

Here we automatically generate efficient linear time algorithms for several maximum marking problems by using the system described in Section 6.2.1.

**Maximum segment sum problem**

Here we automatically generate an efficient linear time algorithm for solving the maximum segment sum problem (mss for short). The mss problem can be specified as a maximum marking problem.

\[
mss = \uparrow w / \circ \ \text{filter conn}_0 \circ \ \text{gen}
\]  
(6.1)

Firstly \(2^n\) marked lists are generated by the generation function \(\text{gen}\). Then by filter \(\text{conn}_0\) only lists that satisfies the property \(\text{conn}_0\) is extracted. Finally by \(\uparrow w /\) a marked list that maximizes the value of the weight function \(w\).

We would like to obtain an efficient linear time algorithm for mss from this specification. We describe CCP for that as in Figure 6.8. The equation (6.1) is written in the first line in Figure 6.8. The function \(bmax w\) represents \((\uparrow w)\) and the function \(\text{reduce}\) represents (/).

The rule \texttt{mmpRule} represents the optimization theorem (Theorem 3), that transforms to the optimization function \(\text{opt}\). If properties satisfy the condition in the optimization theorem, the property \(\text{conn}_0\) can be decomposed to a composition of an accept function and a finite catamorphism by the rule <tupling>. By compiling this CCP, we obtain a script similar to the one in Figure 6.4.


-------------- Program -----------------

\[ \text{mss} = \text{reduce}(\text{bmax} \: \text{w}) \: . \: \text{filter} \: \text{conn0} \: . \: \text{gen}; \]

\[
\text{reduce } f \ [x] = x;
\text{reduce } f \ (x:xs) = f \ x \ (\text{reduce } f \ xs);
\]

\[
\text{bmax } f \ a \ b = \text{if } f \ a > f \ b \ \text{then } a \ \text{else } b;
\]

\[
\text{w} = \text{reduce} \ (+) \ . \ \text{map} \ \text{fc}
\]

\[
\text{where } \text{fc } (x,m) = \text{if } \text{marked } (x,m) \ \text{then } x \\
\text{else } 0;
\]

\[
\text{filter } p \ [] = [];
\text{filter } p \ (x:xs) = \text{if } p \ x \ \text{then } x : \text{filter } p \ xs \\
\text{else } \text{filter } p \ xs;
\]

\[
\text{conn0 } [] = \text{True};
\text{conn0 } (x:xs) = \text{if } \text{marked } x \ \text{then } \text{conn1} \ xs \ \text{else } \text{conn0} \ xs;
\]

\[
\text{conn1 } [] = \text{True};
\text{conn1 } (x:xs) = \text{if } \text{marked } x \ \text{then } \text{conn1} \ xs \ \text{else } \text{conn2} \ xs;
\]

\[
\text{conn2 } [] = \text{True};
\text{conn2 } (x:xs) = \text{if } \text{marked } x \ \text{then } \text{False} \ \text{else } \text{conn2} \ xs;
\]

\[
\text{marked } (x,b) = b;
\text{gen } [] = [[]];
\text{gen } (x:xs) = [ x':xs' | \\
x' \leftarrow [\text{mark } x, \text{unmark } x], \\
xs' \leftarrow \text{gen} \ xs ];
\]

\[
\text{mark } x = (x,\text{True});
\text{unmark } x = (x,\text{False});
\]

\[
\text{foldr } \_\text{step} \ _e \ [] = _e;
\text{foldr } \_\text{step} \ _e \ (_a:_x) = \_\text{step} \ _a \ (\text{foldr} \ _\text{step} \ _e \ _x);
\]

Figure 6.8: CCP for the maximum segment sum problem (the first half)
---------- calculation rules ----------

\[ \text{<mmpRule> (reduce (bmax \_w) \cdot filter conn0 \cdot gen) =} \]
\[ \text{letm \_accept . \_h = <tupling> conn0;} \]
\[ \text{reduce \_oplus \cdot map \_f = \_w;} \]
\[ \text{foldr \_phi2 \_phi1 = <applyFusion> \_h} \]
\[ \text{in opt \(_oplus,\_f\)_accept \_phi1 \_phi2;} \]

\[ <\text{tupling}> \text{ conn0 =} \]
\[ \text{letm \_h xs = (conn0 xs,conn1 xs,conn2 xs);} \]
\[ \text{\_accept (t0,t1,t2) = t0;} \]
\[ \text{in \_accept . \_h;} \]

\[ <\text{fusion}> \_\_f (foldr \_step \_e \_xs) = \]
\[ \text{letm \_g a (_\_f x) = \_f (_\_step a x);} \]
\[ \text{\_c = \_f \_e} \]
\[ \text{in foldr \_g \_c \_xs;} \]

\[ <\text{applyFusion}> (_\_f . _\_g) = <\text{fusion}> (_\_f . <\text{applyFusion}> _\_g); \]
\[ <\text{applyFusion}> _\_f = <\text{fusion}> (_\_f . foldr (:) [])]; \]

-------- Application of calculation rule --------

\[
\text{mssOpt} = <\text{mmpRule> (<unfold> mss)}
\]

Figure 6.9: CCP for the maximum segment sum problem (the second half)
Figure 6.10: CCP for the extended version of mss

An extension of the mss problem

Here we consider a problem that is an extension of the mss problem, extended with respect to the property a solution should satisfy. We require that the number of selected elements in the input list should be even. By letting the additional condition \( \text{even} \), we can describe the property \( p \) for the extended problem as follows:

\[
p \; xs = \text{conn}_0 \; xs \land \text{even} \; xs
\]

where the property \( \text{conn}_0 \) is for the original mss problem.

The extended problem is also a maximum marking problem. CCP for the extended problem can be described as in Figure 6.10 where only different part from Figure 6.8 is described. The calculation rule \texttt{mmpRule} is not changed at all.
6.2.5 Summary

In this chapter, we show an implementation of the theorem we proposed, that is, we design a system for automatically generating efficient linear time algorithms for maximum marking problems based on our theory proposed in this thesis. In concrete aspect, we propose a language for describing transformation, implement the theorem in the language, and show the system implementation. We showed effectiveness of our method by automatically generating efficient linear time algorithms for several maximum marking problems using the implemented system.
Chapter 7

Related Work

Since the mid-1980s there have been several powerful approaches by dynamic programming, and many NP-complete problems have thus been reduced to linear-time problems for families of recursively constructed graphs. The pioneering work in this field was on series-parallel graphs [TNS82].

The backbone concept is a class of graphs with bounded tree-width [RS86], which was independently developed as partial k-tree [Arn87] and is also discussed in terms of separators [Tho90a] and cliques [KT90]. The set of graphs with tree-width at most $k$ is equal to the set of $k$-terminal graphs constructed by algebraic composition rules [ACPS93]. NP-complete problems on graphs, such as the Hamilton path problem, are often linear in the size and beyond exponential in the tree width by the divide-and-conquer strategy according to this algebraic construction.

Much work has been done [Bod93, Arn95], but most of it is on individual graph problems. Courcelle showed that whether a (hyper) graph with bounded tree width satisfies a closed monadic second order (MSOL, for short) formula (of graphs) is solvable in linear time [Cou90a]. Borie et al. further showed that the recognition, enumeration, and optimization problems specified by an extension of MSOL formula can be solved in linear time [BPT92]. This graph variant of MSOL uses $Inc (v,e)$, which means a vertex $v$ is an incident of an edge $e$, instead of the use of a successor function in ordinary MSOL [Tho90b].

Although appealing in theory, these methods are hardly useful in practice due to a huge constant factor for space and time. This arises from the manipulation of huge tables:
• The construction of tables reflects the decomposition of the property description into primitive ones; previous methods adapt fixed basic predicates as primitives, whereas our method freely adapts new primitives if they are in the form of mutumorphisms.

• Their construction of tables causes the exponential blow-up at each occurrence of quantifiers. Our method replaces the occurrences of quantifiers with recursions, which are computationally more efficient.

Aspvall et al. proposed a method to reduce the run-time memory storage by reducing the number of simultaneously needed tables [APT00]. Our work reduces the size of tables directly, complementing their work.

To investigate the relation between mutumorphisms and MSOL is a good research topic, to which recently one solution seemed to be given by Kassios [Kas01]. He gave an algorithm which transforms logical formula in [BPT92] to mutumorphisms which consist of small number of functions.

Bird calculated a linear-time algorithm for solving the maximum segment sum problem on lists [Bir89], which is a kind of maximum marking problem. Bird and de Moor studied optimization problems, which include maximum marking problems, in a more general way that uses relational calculus [BdM96]. For instance, a greedy linear functional program for the party planning problem can be derived using relation calculus. Using relational calculus, they developed a very general framework to treat optimization problems. It is called thinning theory, and the thinning theorem plays the central role. But when applying the thinning theorem, one has to find two preorders which meet prerequisites of the thinning theorem, which makes it difficult for the theorem to be used for mechanical program generation. And they didn’t show the relation between the complexity of derived algorithms and specifications, in return for discussing in a very general framework. We instead focus on a useful class of optimization problems, maximum marking problems, propose a very simple way to derive efficient algorithms, and assure the complexity of the derived algorithms. When the target problem is a maximum marking problem, the two preorders for the thinning theorem can be derived automatically by applying the optimization theorem. To apply our optimization theorem, the only thing one has to do is to describe the property in the form of mutumorphisms. One significant reason for achieving this simplicity in derivation is that we have recognized the importance of the finiteness of the range of a catamorphism as in the optimization lemma, which received little attention in [BdM96].
De Moor considered a generic program for sequential decision processes [dM95] which are specified as follows:

\[
\text{listmin } r . \ \text{filter } p . \ \text{fold (choice } fs) \ [c]
\]

The target problems are on lists and trees. They include maximum marking problems by letting the list of functions \( fs \), used in the above specification of sequential decision processes, be a list of marking functions. But property \( p \) is restricted to be suffix-closed. There are many examples whose property is not suffix-closed.

Johan Jeuring proposed several fusion theorems, each of which deals with a class of optimization problems such as subsequence problems on lists, partition problems on lists, and so on [Jeu93]. In order to derive an efficient program for a problem by his method, one has to select a suitable fusion theorem, which is not necessary in our method.

Recently, Bird showed that the maximum marking problems can be treated in the framework of thinning theory [Bir00]. He assured the derived algorithm is a linear time algorithm, and showed the generic Haskell program for solving the maximum marking problems on polynomial data types. His method also requires that the property \( p \) should be suffix-closed. We show his method later in detail and compare it with ours.

In graph algorithms, Borie et al. proposed a method which enables the derivation of a linear time algorithm for solving the maximum marking problems on \( k \)-terminal graphs, a restricted class of graphs, from logical description of properties by a graph variant of monadic second order formula [BPT92]. This graph variant of MSOL uses \( Inc(v,e) \), which means a vertex \( v \) is an incident of an edge \( e \), instead of the use of a successor function in ordinary MSOL [Tho90b]. Although appealing in theory, these methods are hardly useful in practice due to a huge constant factor for space and time.

**Program Transformation System**

There are several program transformation system. One of them is stratego [Vis01], which is a language for describing a program transformation. The system we implemented in this thesis depends on a calculation describing language called Calculation Carrying Program (CCP for short) [HT99]. The words ”calculation carrying” represent that the code which specifies how to calculate an efficient program is accompanied with the original program.
This method is useful for the user who use the program in the sense that he can know the meaning of the program easily by the original program in CCP, and at the same time he can obtain an efficient program by calculating it. Another possible approach for describing program transformation is to use meta programming language such as Meta-ML, which is an extension of the language ML.

**Relational Calculus**

Here we show the derivation of a linear time algorithm for MMP by relational calculus. First, we show the Bird’s derivation [Bir00]. Second, we extend the specification to one which can express the same class as ours specification can do.

**Specification by Bird**

Maximum marking problem can be specified by a relation as follows:

\[
\textit{mmp} = \text{max} (\leq) \circ \Lambda (w \circ \text{dom} \ p \circ \text{map}_T \ \text{mark})
\]

This specification is interpreted not as Haskell functions, but as multifunctions. A multifunction is a nondeterministic function, that is, a relation that associates zero or more results with each argument. Type of multifunction is written as \(A \rightsquigarrow B\) rather than \(A \to B\). For example,

\[
\begin{align*}
\text{mark} & \:: \alpha \rightsquigarrow \alpha^* \\
\text{mark } n & \quad = \quad (n, \text{True}) \sqcup (n, \text{False})
\end{align*}
\]

The type of \(p\) is

\[
p :: T \ \alpha^* \rightsquigarrow \text{Class}
\]

For the mss problem, \(p\) can be defined as follows:

\[
\begin{align*}
\text{conn} & \:: \quad T \ \alpha^* \rightsquigarrow \text{Class} \\
\text{conn} & \quad = \quad \text{fold}_{\text{List}} \ f \\
\text{where } f (1, (\)) & \quad = \quad c_1 \\
\text{f } (2, ((x, b), \ r)) & \quad = \quad \begin{cases} 
    r \quad == \quad c_1 \land b & \rightarrow \quad c_2 \\
    r \quad == \quad c_1 \land \neg b & \rightarrow \quad c_1 \\
    r \quad == \quad c_2 \land b & \rightarrow \quad c_2 \\
    r \quad == \quad c_2 \land \neg b & \rightarrow \quad c_3 \\
    r \quad == \quad c_3 \land \neg b & \rightarrow \quad c_3
\end{cases}
\end{align*}
\]
Here $F_{\text{List}}$ is defined as follows:

$$F_{\text{List}}(a, b) = 1 + a \times b$$

The box $\Box$ signifies nondeterministic choice. The type $\alpha^*$ is defined as follows:

$$\alpha^* = \alpha \times \text{Bool}$$

The type of $\text{map}_T$ is

$$\text{map}_T :: (a \leadsto b) \rightarrow (T \ a \leadsto T \ b)$$

This is like the ordinary map function associated with a data type $T$ except that it can take a multifunction as argument and return a multifunction as result. The combination $\text{map}_T \ \text{mark}$ denotes the operation of marking an element of $T \ a$ in a completely nondeterministic way.

The function $\text{dom}$, which takes a multifunction as argument and returns a partial function as result, is defined as follows:

$$\text{dom} :: (a \leadsto b) \rightarrow (a \rightarrow a)$$
$$\text{dom} \ p = \text{fst} \circ \langle \text{id}, p \rangle$$

The split operation $\langle f, g \rangle$ is defined by $\langle f, g \rangle \ x = (f \ x, g \ x)$. The expression $\langle f, g \rangle$ denotes a multifunction that returns a result on an argument $x$ if and only if both $f$ and $g$ do.

The weight function $w$ is defined as follows:

$$w :: T \ \alpha^* \rightarrow \text{Weight}$$
$$w = \text{sum} \circ \text{map}_T \ mw$$
$$mw \ (e, b) = \text{if} \ b \ \text{then} \ e \ \text{else} \ 0$$

The subsidiary function $\text{sum} :: T \ \text{Weight} \rightarrow \text{Weight}$ for summing a structure of weights will be defined later.

The operation $\Lambda$ turns a multifunction into the corresponding set-valued function:

$$\Lambda :: (a \leadsto b) \rightarrow (a \rightarrow \text{Set} \ b)$$
$$\langle \Lambda f \rangle \ a = \{b \mid b \leftarrow f \ a\}$$

We write $b \leftarrow f \ a$ to denote the fact that $b$ is a possible value of $f \ a$. Thus $\langle \Lambda f \rangle \ a$ returns the set of all possible values $b$ that can be returned as the result of applying the multifunction $f$ to $a$. 

105
The multifunction \( \text{max} \) is defined by

\[
\text{max} :: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow (\text{Set} \ a \rightarrow a)
\]

\[
a \leftarrow \text{max} \ (\leq) \ as \ \equiv \ a \in as \wedge (\forall b \in as : b \leq a)
\]

In words, \( \text{max} \) takes an ordering \( \leq \) and a set \( as \) as argument, and returns some maximum element in \( as \) under \( \leq \). It is required that \( \leq \) should be a connected preorder. A relation \( r \) is said to be preorder if \( r \) is a reflexive and transitive relation. A relation \( r \) is said to be connected if for all \( x \) and \( y \), either \( x \leq y \) or \( y \leq x \). Then it is guaranteed that \( \text{max} \ (\leq) \ as \) produces at least one result for all nonempty sets \( as \). It is different from functional specification that \( \text{max} \ (\leq) \ as \) does not specify which element of \( as \) should be chosen.

**Calculation**

The specification of \( \text{mmp} \) is as follows:

\[
\text{mmp} = \text{max} \ (\leq) \circ \Lambda (w \circ \text{dom} \ p \circ \text{map}_T \ \text{mark})
\]

First, rewrite the subexpression \( w \circ \text{dom} \ p \):

\[
w \circ \text{dom} \ p
\]

\[
= \ { \{ \text{definition of dom} \} }
\]

\[
w \circ \text{fst} \circ \langle id, p \rangle
\]

\[
= \ { \{ \text{claim} \} }
\]

\[
\text{fst} \circ \langle w, p \rangle
\]

\[
= \ { \{ \text{definition of w} \} }
\]

\[
\text{fst} \circ \langle \text{fold}_F \ (\text{plus}_F \circ \text{map}_F \ mw, p) \rangle
\]

\[
= \ { \{ \text{type functor fusion} \} }
\]

\[
\text{fst} \circ \langle \text{fold}_F \ (\text{plus}_F \circ \text{map}_F \ (mw, id)), p \rangle
\]

\[
= \ { \{ \text{assume} \ p = \text{fold}_F \ \text{part}_F \} }
\]

\[
\text{fst} \circ \langle \text{fold}_F \ (\text{plus}_F \circ \text{map}_F \ (mw, id)), \text{fold}_F \ \text{part}_F \rangle
\]

\[
= \ { \{ \text{banana split} \} }
\]

\[
\text{fst} \circ \langle \text{fold}_F \ f \rangle
\]

Here, \( f \) is defined as follows:

\[
f :: F(\alpha^*, (\text{Weight}, \text{Class})) \rightarrow (\text{Weight}, \text{Class})
\]

\[
f = \langle \text{plus}_F \circ \text{map}_F (mw, \text{fst}), \text{part}_F \circ \text{map}_F (id, \text{snd}) \rangle
\]
The claim is a consequence of two laws:

\[ f \circ \text{fst} = \text{fst} \circ (f \times \text{id}) \]
\[ (f \times g) \circ \langle h, k \rangle = \langle f \circ h, g \circ k \rangle \]

where \((f \times g)(x, y) = (f \, x, g \, y)\).

Now we can rewrite the subexpression \(w \circ \text{dom} \circ \text{map}_T \text{mark}\):

\[ w \circ \text{dom} \circ \text{map}_T \text{mark} = \{ \text{above} \} \]
\[ \text{fst} \circ \text{fold}_F \, f \circ \text{map}_T \text{mark} = \{ \text{type functor fusion} \} \]
\[ \text{fst} \circ \text{fold}_F \, g \]

where \(g :: F(\alpha, (\text{Weight}, \text{Class})) \leadsto (\text{Weight}, \text{Class})\) is defined by \(g = f \circ \text{map}_F (\text{mark}, \text{id})\). Using this result, we now obtain

\[ \max (\leq) \circ \Lambda (w \circ \text{dom} \circ \text{map}_T \text{mark}) \]
\[ = \{ \text{above} \} \]
\[ \max (\leq) \circ \Lambda (\text{fst} \circ \text{fold}_F \, g) \]
\[ = \{ \text{claim} \} \]
\[ \text{fst} \circ \max (\leq_1) \circ \Lambda (\text{fold}_F \, g) \]

where \((\leq_1) :: (\text{Weight}, \text{Class}) \rightarrow (\text{Weight}, \text{Class}) \rightarrow \text{Bool}\) is defined by

\[ (a_1, b_1) \leq_1 (a_2, b_2) \overset{\text{def}}{=} (a_1 \leq a_2) \]

The proof of the claim is in [BdM96].

Now we can apply the thinning theorem to the expression

\[ \max (\leq_1) \circ \Lambda (\text{fold}_F \, g). \]

**Theorem 6 (Thinning)**

\[ \max (\leq) \circ \Lambda (\text{fold}_F \, f) \supseteq \max (\leq) \circ \text{fold}_F \, g \]

where

\[ g :: F(a, \text{Set} \, b) \leadsto \text{Set} \, b \]
\[ g = \text{thin} (\leq) \circ \Lambda (f \circ \text{map}_F (\text{id}, \text{choose})) \]

provided that: (i) \(x \leq y \Rightarrow x \leq y\); and (ii) \(f\) is monotonic under \(\leq\). \(\square\)
Here \( \text{thin} \) is defined as follows:

\[
\text{thin} :: (a \to a \to \text{Bool}) \to (\text{Set } a \to \text{Set } a)
\]

\[
bs \leftarrow \text{thin } (\leq) \text{ as } \equiv \ bs \subseteq \text{as } \land (\forall a \in \text{as} : \exists b \in bs : a \leq b)
\]

To apply the thinning theorem, we have to find an order \( \leq \) which satisfies the condition (i) and (ii) in the theorem. In this case, the following order \( \leq_2 \) satisfies them.

\[
(a_1, b_1) \leq_2 (a_2, b_2) \overset{\text{def}}{=} (a_1 \leq a_2 \land b_1 == b_2)
\]

Now the specification is calculated to the following form:

\[
\text{mmp } \supseteq \ fst \circ \text{max } (\leq_1) \circ \text{fold}_F g
\]

\[
g = \text{thin } (\leq_2) \circ \Lambda(f \circ \text{map}_F (id, \text{choose}))
\]

\[
f = (\text{plus}_F \circ \text{map}_F (mw, \text{fst}), \text{part}_F \circ \text{map}_F (id, \text{snd})) \circ \text{map}_F (\text{mark}, id)
\]

The thinning theorem says that we can compute an optimum result by maintaining a representative number of partial solutions at each stage of the folding process. If \( \text{Class} \) has size \( k \), then we need keep only \( k \) partial solutions at each stage. For the mss problem, \(|\text{Class}| = 3\), so only three partial solutions have to be kept.

**Property by a relational fold**

Properties are restricted to those which can be expressed using \( \text{fold}_F \):

\[
p = \text{fold}_F \text{ part}_F
\]

This restricts properties to “suffix-closed” properties. Originally the term “suffix closed” is used for lists. If the property \( p \) doesn’t hold for the list \( xs \), then \( p \) doesn’t hold for the list \( ys ++ xs \) for all \( ys \). We use the suffix-closed for data type \( T \), that is, if \( p \) doesn’t hold for the property \( p \) on the data \( x \), then \( p \) doesn’t hold for data which have \( x \) as its sub-data.

**Extension of Property class**

We extend properties to those which can be written as follows:

\[
p :: T \overset{\alpha^*}{\to} \text{Class}
\]

\[
p = \text{accept } \circ \text{fold}_F \text{ part}_F
\]
Here \textit{accept} is a partial function which has following type:

\[
\text{accept} :: \text{Class} \rightarrow \text{Class}
\]

Though we omit detail, the specification \textit{mmp} can be calculated to the following expression:

\[
fst \circ \text{dom} \ (\text{accept} \circ \text{snd}) \circ \text{max} \ (\leq 1) \circ \text{fold}_F \ g
\]

\textbf{Discussion}

The derivation by relational calculus is clearer than that by functional calculus because by functional calculus, essentially nondeterministic part have to be represented by a function. So, it may seem that it may be better to replace the proof of the optimization function \textit{opt} by one using a relational calculus. It is true that the relational calculus is clear, but it is remained to implement the obtained relation as a function. This step is slightly complicated and actually it takes about 4 pages to implement obtained result as a function in [Bir00].
Chapter 8

Conclusion

In this thesis we present a new method for deriving practical linear-time algorithms for maximum marking problems on recursive data structures. From a specification represented as a functional program, our method obtains practical linear-time algorithms by deriving a catamorphism whose range size is small. Though our method does not guarantee that the derived catamorphism has a range with the smallest number of elements, in many cases it has range consisting of about 10 or 20 elements because the properties of many problems can be described in mutumorphisms which consist of three or four property descriptions. This means our derived linear-time algorithms are practical in many cases. Our method also enables linear time algorithms to be derived in a simple, systematic, and natural way, and it can flexibly cope with modification of the specification.

Though our current focus is on well-used recursive data structures such as lists and trees, our method should be able to deal with the graphs with bounded tree width because a class of the graphs with bounded tree width can be represented as a recursive data structure like that described in Chapter 2. Our next target problems are linear-time control flow analyses for structured procedural programs, for which the control flow graphs are known to have bounded tree width [Tho98]. Though we currently restrict data types to polynomial data types, we think our optimization theorem can be extended to regular data types. Although we concentrate on the maximizing problems, our method is easily extended for the minimization, existence and enumeration problems in [BPT92].

Our method is mechanical, so it is appropriate for automatic generation. We have shown implementation of a system for automatically generat-
ing efficient linear time algorithms for maximum marking problems. It was known that generation of linear time algorithms for MMP (precisely maximum weightsum problem) can be done by the method proposed by Borie et al., [BPT92]. Though his method is theoretically appealing, it generates a prohibitively large table and thus cannot be put to practical use. We have shown that our system generates practical linear time algorithms by using several examples.

Our method of course has limitations, and the most fundamental is that tree width is bounded. The two-dimensional maximum segment sum problem, for example, cannot be dealt with by our method because a two-dimensional matrix corresponds to a grid graph, which will have unbounded tree width.
Bibliography


[FMFT96a] Takeshi Fukuda, Yasuhiro Morimoto, Shinichi Morishita, and
Takeshi Tokuyama. Data mining using two-dimensional optimized
association rules: Scheme, algorithms, and visualization. In
Proceedings of the 1996 ACM SIGMOD International Conference
on Management of Data (SIGMOD’96), pages 13–23, Montreal,

[FMFT96b] Takeshi Fukuda, Yasuhiro Morimoto, Shinichi Morishita, and
Takeshi Tokuyama. Mining optimized association rules for numeric
attributes. In Proceedings of the fifteenth ACM SIGACT-
SIGMOD-SIGART symposium on Principles of database systems
(PODS’96), pages 182–191, Montreal, Canada, June 1996.
ACM Press.

[Fok89] Maarten M. Fokkinga. Tupling and mutumorphisms. Squig-

[Fok92] Maarten M. Fokkinga. Law and Order in Algorithmics. PhD
thesis, University of Twente, Dept INF, Enschede, The Netherlands,

short cut to deforestation. In Proceedings of the 6th Inter-
national Conference on Functional Programming Languages
and Computer Architecture (FPCA’93), pages 223–232, Copen-
hagen, Denmark, June 1993. ACM Press.

[Gri90] D. Gries. The maximum-segment sum problem. In E. W. Di-
ijkstra, editor, Formal Development of Programs and Proofs.
Addison-Wesley, 1990.

[HITT97] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko
Takano. Tupling calculation eliminates multiple data traversals.
In Proceedings of the 2nd ACM SIGPLAN International Con-
fERENCE ON FUNCTIONAL PROGRAMMING (ICFP’97), pages 164–175,

[HT99] Zhenjiang Hu and Masato Takeichi. Calculation carrying pro-
grams. Technical Report METR 99-07, Department of Mathe-
matical Engineering, University of Tokyo, Japan, 1999.


