MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

Solving Maximum Weighted-sum Problems for Free

Akimasa MORIHATA
(communicated by Masato TAKEICHI)

2009–20 May 2009

DEPARTMENT OF MATHEMATICAL INFORMATICS
GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY
THE UNIVERSITY OF TOKYO
BUNKYO-KU, TOKYO 113-8656, JAPAN

WWW page: http://www.keisu.t.u-tokyo.ac.jp/research/techrep/index.html

Solving Maximum Weighted-sum Problems for Free

Akimasa Morihata
JSPS research fellow
morihata@ipl.t.u-tokyo.ac.jp

Abstract

Efficient algorithms are necessary for solving problems effectively. However, efficient algorithms are hard to develop, implement, and maintain. Therefore, it has been very difficult for those with little algorithmic knowledge to solve their problems based on efficient algorithms. In this paper, we introduce a way to develop efficient algorithms for finding optimal sequences, especially sequences of the maximum weighted sum. we formalize derivation of efficient algorithms by shortcut fusion, which is a program transformation known to be suitable for mechanization, and propose a set of laws that is helpful to derive well-structured consumers that are necessary for utilizing our shortcut fusion laws. Moreover, we introduce a library for obtaining optimal lists. The library consists of several functions useful for describing lists satisfying our requirements; moreover, the library consists of program transformation rules so that programs written by our library will be improved automatically into ones based on efficient algorithms.

1 Introduction

Every day, we would like to find better ways, such as ways to reduce costs, increase benefits, and so on. For example, consider the following problem[1], which might be a typical situation that we would face of.

> We are working on processing costumers' data by our high-performance system. However, because of a bug of the system, consecutive run of the system will decrease its performance, and thus, we should periodically reboot the system, which takes a day. Now, given an amount of data for each of next n days, where we can process the data only on the day, we would like to find out the best scheduling for running the system.

Can we find the best scheduling? It is theoretically not difficult at all. Since there is only finite number of schedules, we can enumerate all of them and find the best one. However, such a trivial way is completely impractical, because the number of schedules is exponential to the number of days. Therefore, we need to develop an efficient algorithm. Here note that "timely" rebooting will be necessary for the best profit. For example, rebooting may be effective if a lot of data is available on the next day. Thereof, naive greedy scheduling would not be the best.

In fact, it would not be very difficult to find the best scheduling for those having algorithmic knowledge. Such problems are called combinatorial optimization problems, which are problems to find the best (optimal) solution among those satisfying requirements. Combinatorial optimization problems have been studied very deeply, and many efficient algorithms have been proposed for several classes of problems.

Despite existing rich studies about efficient algorithms, it is not easy to solve such problems by implementing efficient programs. First, implementer may not have enough algorithmic knowledge, because existing algorithmic results are too vast for nonspecialists to follow. Next, efficient algorithms will require tricky implementation that is error-prone. Moreover, efficient algorithms are hard to modify or maintain. Even an additional requirement or small change of objective value will ruin efficiency and/or correctness of algorithms. As a consequence, most people have solved their problems in an ad-hoc manner and much of algorithmic results have been left unutilized.

Our goal is to construct an environment in which we can utilize algorithmic results in ease. In this paper, we focus on problems to find the best sequences. Especially, we mainly consider *maximum weighted*

[1] This problem is taken from a textbook by Kleinberg and Tardos [KT05].

sum problems [SHTO00, Bir01], in which we would like to find sequences of the maximum weighed sum among those satisfying requirements. First, we will formalize derivation of efficient algorithms for such problems by program transformation rules. Then, we will introduce a library for solving such problems in ease. Based on the program transformation rules proposed, naive programs written by using the library will be improved into ones based of efficient algorithms.

Now let us overview our study.

In Section 3, we first formalize derivation of efficient algorithms by using *shortcut fusion* [GLJ93, Gil96] (also called *shortcut deforestation*), which is a program transformation rule for removing intermediate data structures. In shortcut fusion laws, we consider two kinds of functions, namely producers that generate intermediate results and consumers that consume intermediate results generated by producers. Shortcut fusion laws enable us to fuse compositions of well-structured consumer and well-structured producer and yield a program without intermediate results. We regard "a set of candidates" as intermediate results in combinatorial optimization problems and "an efficient algorithm" as a program that yields the optimal result without generating the intermediate results. Then, we formalize derivation of efficient algorithms as shortcut fusion laws. It is worth noting that our formalization is suitable for automatic implementation as well as shortcut fusion.

Since the key to shortcut fusion is a pair of functions, namely a well-structured consumer and a well-structured producer, a way to develop them is necessary. In our case, we can utilize existing methods [LS95, Chi99, YHT05] for developing well-structured producers; however, our well-structured consumers should satisfy an additional property, called *monotonicity*, and thus, we should develop a method for them. In Section 4, we prepare program transformation rules to derive well-structured consumers. We identify idioms for specifying the best solution and provide laws to deal with them.

As a result of developed laws, we introduce a library for enumerating the best sequences in Section 5. Our library consists of useful functions for enumerating the best sequences. In addition, our library consists of a set of program transformation rules that derive efficient algorithms from compositions of the library functions. The program transformation rules are implemented by *RULES pragma* [PTH01], which is an extensional functionality of Glasgow Haskell Compiler. One of the strengths of our library is that our library does not require algorithmic insight for users at all. It is sufficient to describe the problems in a naive generate-and-find manner for obtaining efficient programs. By virtue of declarative descriptions of problems, programs written by using our library are easy to modify or maintain. Moreover, our library naturally cooperates with usual Haskell programs. We can write a core part of a large system by using our library; besides, we can use Haskell functions as parameters of the library functions. We also report our experiments for confirming effectiveness of our library in Section 6. We solved several problems by our library, and compared some of them with handwritten programs.

Finally we give a conclusion of the paper in Section 7 with discussing relationship to existing works and possible further extensions.

2 Preliminary

2.1 Basic Notions

We basically borrow notations of functional programming language Haskell [Pey03]. We may omit parentheses for function applications; thus, $f\ x$ is equivalent to $f(x)$. Note that function applications precede those for operators, and thus, $a+f\ x$ is equivalent to $a+(f\ x)$. Operators might be sectioned, that is for example, we may write $(+)\ 1\ 4$ instead of $1+4$. An operator $\circ$ is the function composition operator, and its definition is $(f \circ g)\ x \stackrel{\text{def}}{=} f(g\ x)$. We will make use of pattern matches and lambda notations. The underscore _ denotes "don't care" pattern, which matches any value without any bindings. We will only consider terminating functions, and no "undefined" value is taken into account.

A binary relation $\preceq$ is said to be *preorder* (also called *quasi-order*) if it satisfies reflectivity $\forall a.\, a \preceq a$ and transitivity $\forall a\, b\, c.\, (a \preceq b \wedge b \preceq c) \Rightarrow a \preceq c$. A preorder $\sim$ is said to be *equivalence relation* if it satisfies symmetry $\forall a\, b.\, a \sim b \Rightarrow b \sim a$. An *intersection* of two orders $\preceq$ and $\ll$, denoted by $\preceq \cap \ll$, is defined by $a\ (\preceq \cap \ll)\ b \stackrel{\text{def}}{\Leftrightarrow} a \preceq b \wedge a \ll b$. A *lexicographic composition* of two orders $\preceq$ and $\ll$, denoted by $\preceq ; \ll$, is defined by $a\ (\preceq ; \ll)\ b \stackrel{\text{def}}{\Leftrightarrow} a \ll b \wedge (\neg(b \ll a) \vee a \preceq b)$. Intuitively, $\preceq ; \ll$ compares two elements by $\ll$ first, and by $\preceq$ afterwards if they are equivalent on $\ll$. Given a total preorder $\preceq$, $\uparrow_{\preceq}$

$$
\begin{aligned}
&\mathit{map}_{Set}\ f\ x \stackrel{\text{def}}{=} \{f\ a \mid a \in x\}\\
&\mathit{map}_{List}\ f\ x \stackrel{\text{def}}{=} [f\ a \mid a \in x]\\
&\mathit{filter}_{Set}\ p\ x \stackrel{\text{def}}{=} \{a \mid a \in x \wedge p\ a\}\\
&\mathit{all}_{Set}\ p\ x \stackrel{\text{def}}{=} \forall a \in x.\, p\ a\\
&\mathit{sum}\ [a_0, a_1, \ldots, a_{n-1}] \stackrel{\text{def}}{=} a_0 + a_1 + \cdots + a_{n-1}\\
&\mathit{size}\ \{a_0, a_1, \ldots, a_{n-1}\} \stackrel{\text{def}}{=} n\\
&\mathit{length}\ [a_0, a_1, \ldots, a_{n-1}] \stackrel{\text{def}}{=} n\\
&[a_0, a_1, \ldots, a_{n-1}] \mathbin{!!} k \stackrel{\text{def}}{=} a_k\\
&\mathit{tails}\ [a_0, a_1, \ldots, a_{n-1}] \stackrel{\text{def}}{=} \{[], [a_{n-1}], \ldots, [a_0, a_1, \ldots, a_{n-1}]\}\\
&\mathit{foldr}\ f\ e\ [a_0, a_1, \ldots, a_{n-1}] \stackrel{\text{def}}{=} f\ a_0\ (f\ a_1\ (\cdots (f\ a_{n-1}\ e)\cdots))\\
&\mathit{mapAccumR}\ f\ e\ [] \qquad = (e, [])\\
&\mathit{mapAccumR}\ f\ e\ (a : x) = \mathbf{let}\ (s, y) = \mathit{mapAccumR}\ f\ e\ x\\
&\qquad\qquad\qquad\qquad\quad (s', b) = f\ s\ a\\
&\qquad\qquad\qquad\qquad \mathbf{in}\ (s', b : y)
\end{aligned}
$$

Figure 1: definitions of basic lists/sets-manipulating functions

denotes an operator taking maximum of two operands, i.e., $a \uparrow_{\preceq} b \stackrel{\text{def}}{=} \mathbf{if}\ a \preceq b\ \mathbf{then}\ b\ \mathbf{else}\ a$. We will omit the subscript if it is apparent from its context. Given an order $\preceq$ and a function f, $\preceq_f$ denotes another order defied as $a \preceq_f b \stackrel{\text{def}}{\Leftrightarrow} f(a) \preceq f(b)$. Especially, $=_f$ denotes the equivalence relation raised by the function f.

When a function f takes a value of type A and results in a value of type B, we will write $f :: A \to B$. A function might be polymorphic. For example, the identity function id, defined by $id\ x \stackrel{\text{def}}{=} x$, has the type $id :: \forall a.\, a \to a$. Besides, *type classes* [HHJW96] provide a way to specify types having a specific set of operations. For example, equality checking function $(=)$ should takes two elements of the same type in which equality is available; such requirement is expressed by $(=) :: \forall a.\, \mathit{Eq}\ a \Rightarrow a \to a$, where $\mathit{Eq}\ a$ denotes that a should be a subtype of a class Eq. In this paper, we will use classes Ord (comparison by a total preorder is available) and Num (numbers with addition/multiplication/sign), in addition to Eq.

A parentheses split by a comma denotes a pair. The first and second element of a pair can be extracted by functions *fst* and *snd*, respectively, i.e., $\mathit{fst}\ (a, \_) \stackrel{\text{def}}{=} a$ and $\mathit{snd}\ (\_, b) \stackrel{\text{def}}{=} b$. We will consider uniform lists constructed from two constructors: the empty list $[] :: \forall a.\, [a]$ and adding an element to the head of a list $(:) :: \forall a.\, a \to [a] \to [a]$. We also consider uniform sets as a datatype constructed from the singleton operation $\{\cdot\} :: \forall a.\, \mathit{Eq}\ a \Rightarrow a \to \{a\}$ and the union operation $(\cup) :: \forall a.\, \mathit{Eq}\ a \Rightarrow \{a\} \to \{a\} \to \{a\}$. We will make use of several standard lists/sets-manipulating functions. We summarize definition of these functions in Figure 1.

2.2 Shortcut Fusion

Consider that a function, called *producer*, generates a structure, and another function, called *consumer*, takes the generated structure and yields the final result. Such a programming pattern is easy to write, understand, and maintain, but it is not very efficient because producing and discarding intermediate structures would be costly. *Shortcut fusion* (also called *shortcut deforestation*), which was first proposed by Gill et al. [GLJ93], is a transformation to eliminating intermediate data structures.

Theorem 1 (Shortcut fusion [GLJ93])**.** For a function $g :: \forall \beta.\, (a \to \beta \to \beta) \to \beta \to \beta$, an equation

$$\mathit{foldr}\ f\ e\ (\mathit{build}\ g) = g\ f\ e$$

holds, where the function *build* is defined as $\mathit{build}\ g = g\ (:)\ []$. □

The key to shortcut fusion is a composition of a well-structured producer and a well-structured consumer. The consumer should be specified by *foldr*; moreover, producer should be specified by so called *build form*, namely *build* with a function having a specific polymorphic type. Then, the intermediate structure generated from the producer is removed for free. Recall that *foldr* is a function to replace list constructors, namely (:) and [], by its arguments. In a build form, all list constructors appearing in the intermediate structure is given from *build*, as guaranteed from the type requirement. Therefore, removal of intermediate structure is actually accomplished by supplying the function g with the arguments of *foldr* instead of the constructors from *build*. The strength of shortcut fusion is its suitability for mechanization. Since fusion is accomplished by just canceling out *foldr* and *build*, once programs are specified by well-structured producers and consumers, it is easy to implement shortcut fusion.

For example, consider eliminating intermediate structure in a program $sum\ (map_{List}\ (\times 2)\ x)$. First, notice that *sum* is equivalent to $foldr\ (+)\ 0$; thus, *sum* is a well-structured consumer. Next, let us derive a build-form for $map_{List}\ (\times 2)\ x$. An apparent way is to abstract out all constructors appearing in the result of the producer. From the following program of $map_{List}\ (\times 2)\ x$,

$$\begin{array}{lcl} map_{List}\ (\times 2)\ [] & \stackrel{\text{def}}{=} & [] \\ map_{List}\ (\times 2)\ (a : x) & \stackrel{\text{def}}{=} & (a \times 2) : map_{List}\ (\times 2)\ x \end{array}$$

we replace all occurrences of constructors (:) and [] in the right hand side expression by parameters c and n.

$$\begin{array}{lcl} map'_{List}\ (\times 2)\ []\ c\ n & \stackrel{\text{def}}{=} & n \\ map'_{List}\ (\times 2)\ (a : x)\ c\ n & \stackrel{\text{def}}{=} & c\ (a \times 2)\ (map_{List}\ (\times 2)\ x\ c\ n) \end{array}$$

Then, as required, $map_{List}\ (\times 2)\ x = build\ (map'_{List}\ (\times 2)\ x)$ holds and $map'_{List}\ (\times 2)\ x$ has the type $\forall \beta.\ (Int \to \beta \to \beta) \to \beta \to \beta$. Therefore, from Theorem 1,

$$sum\ (map_{List}\ (\times 2)\ x) = map'_{List}\ (\times 2)\ x\ (+)\ 0$$

holds, and after inlining, we obtain the following equation.

$$\begin{array}{lcl} sum\ (map_{List}\ (\times 2)\ []) & = & 0 \\ sum\ (map_{List}\ (\times 2)\ (a : x)) & = & (a \times 2) + sum\ (map_{List}\ (\times 2)\ x) \end{array}$$

We can see that no intermediate lists appear in the resulted program.

3 Deriving Efficient Algorithms by Shortcut Fusion

In this section, we would like to consider problems to find the best list, and formalize derivation of efficient algorithms by shortcut fusion laws. As an example, we will consider "maximum segment sum problem" [Ben86, Bir89]: Given a list, compute the maximum sum of segments (consecutive sublist) of the list. The goal of this section is to introduce a shortcut to efficient algorithms for such problems.

Our idea is to recognize derivation of efficient algorithms as elimination of intermediate structures. Consider a naive algorithm in which we enumerate all candidates and find the optimal one. From such naive one, we would like to derive an efficient algorithm, such as an algorithm in which we retain only the optimal substructure. What is the difference? We can see that the difference is presence of candidates in the procedure. This view implicates that we can derive efficient algorithms from naive ones by deforesting intermediate structures, namely enumerated candidates. Since we will consider problems for finding the optimal list, here candidates form a set of lists, and then, what we would like to formalize is a shortcut fusion law to deforest a set of lists.

In the next two subsection, we will introduce our idea through considering the simplest case, that is, computing the maximum summation of a set of lists. After that, we will introduce a more generic program transformation rule that enables us to compute the best list in a shortcut manner.

For formalizing computation for a set of lists, we consider "structural recursions" on sets of lists. A structural recursion is computation that collapses an structure by replacing each constructor by a computation, such as *foldr* for lists.

Definition 2 (structural recursion on sets of lists)**.** Given an operator $(\oplus) :: A \to A \to A$ with its unit $\iota_\oplus$, a function $s :: B \to A$, an operator $(\otimes) :: C \to B \to B$, and a value $n :: C$, $(\!|(\oplus), s, (\otimes), n|\!) :: \{[C]\} \to B$ is the function satisfying the following equations.

$$\begin{array}{lcl} (\!|(\oplus), s, (\otimes), n|\!)\ (x \cup y) & = & (\!|(\oplus), s, (\otimes), n|\!)\ x \oplus (\!|(\oplus), s, (\otimes), n|\!)\ y \\ (\!|(\oplus), s, (\otimes), n|\!)\ \{l\} & = & s\ (\mathit{foldr}\ (\otimes)\ n\ l) \\ (\!|(\oplus), s, (\otimes), n|\!)\ \emptyset & = & \iota_\oplus \end{array}$$

□

Note that $(\!|(\oplus), s, (\otimes), n|\!)$ is defined only if $\oplus$ is associative, commutative, and idempotent. For instance, the function maxSum that extracts the maximum sum is a structural recursion.

$$\mathsf{maxSum} = (\!|\uparrow, id, +, 0|\!)$$

3.1 Shortcut Fusion Law for Maximum-Summations

For shortcut fusion, we need to derive "build form" of a function. However, it is not trivial even to provide an appropriate definition of build forms for our purpose. For instance, consider *inits* function.

$$\begin{array}{lcl} \mathit{inits}\ [] & = & \{[]\} \\ \mathit{inits}\ (a : x) & = & \{[]\} \cup \mathit{map}_{Set}\ (a:)\ (\mathit{inits}\ x) \end{array}$$

As usual, we might attempt to abstract out all constructors, namely $(\cup)$, $\{\cdot\}$, $(:)$, and $[]$; however, we would fail to derive a build form because of map_{Set}: map_{Set} does an iteration over a set made by *inits*, and abstracting out the result of *inits* will obstruct the iteration. To resolve this problem, we will make use of *freezing* technique [KGK01, Voi02]. We would like to regard some functions as a constructor and abstract out. In this case, since map_{Set} obstructs to derive a build form, let us freeze it — then, and again, our attempt leads to a failure. Notice that map_{Set} can does arbitrary operation on elements in the set by its parameter function; thus, the parameter function of map_{Set} will obstruct to abstract out the constructors for lists.

Based on the observation above, our idea is to freeze a specific kind of map_{Set} whose parameter function would not iterate over lists. Consider the following function *extend*.

$$\mathit{extend}\ a\ x \stackrel{\text{def}}{=} \mathit{map}_{Set}\ (a:)\ x$$

We freeze *extend* because of the following two reasons. First of all, the function *extend* is a primal operation to generate a set of lists; besides, it never does any iteration over lists. To understand what *extend* is, recall the isomorphism between a set-generating function and a nondeterministic computation. From the viewpoint of nondeterministic computation, *extend* is nothing but construction of lists; thus, it is natural to recognize *extend* as a constructor.

Now let a function gen be the function that supply "constructors" to a function of "build form".

$$\begin{array}{l} \mathsf{gen} :: \forall a.\, \mathit{Eq}\ a \Rightarrow (\forall \beta\ \gamma.\, (\gamma \to \gamma \to \gamma) \to (\beta \to \gamma) \to (a \to \gamma \to \gamma) \to (a \to \beta \to \beta) \to \beta \to \gamma) \to \{[a]\} \\ \mathsf{gen}\ g \stackrel{\text{def}}{=} g\ (\cup)\ \{\cdot\}\ \mathit{extend}\ (:)\ [] \end{array}$$

Then, the following shortcut fusion law holds, which is our key theorem.

Theorem 3 (shortcut fucion law for generator functions)**.** For a function g of the appropriate type,

$$(\!|(\oplus), s, (\otimes), n|\!)\ (\mathsf{gen}\ g) = g\ (\oplus)\ s\ \eta\ (\otimes)\ n$$

holds, where η is a function satisfying the following equations.

$$\begin{array}{lcl} \eta\ a\ \iota_\oplus & = & \iota_\oplus \\ \eta\ a\ (v \oplus s\ w) & = & \eta\ a\ v \oplus s\ (a \otimes w) \end{array}$$

Proof. By Wadler's free theorem [Wad89], we reason as follows.

True
$\Leftrightarrow$ { parametricity }
$(g, g) \in [\![\forall \beta\ \gamma.\, (\gamma \to \gamma \to \gamma) \to (\beta \to \gamma) \to (A \to \gamma \to \gamma) \to (A \to \beta \to \beta) \to \beta \to \gamma]\!]$
$\Leftrightarrow$ { definition of $\forall$ }
$\forall (f :: \gamma_1 \leftrightarrow \gamma_2), (h :: \beta_1 \leftrightarrow \beta_2).$
$(g_{\beta_1,\gamma_1}, g_{\beta_2,\gamma_2}) \in [\![(f \to f \to f) \to (h \to f) \to (A \to f \to f) \to (A \to h \to h) \to h \to f]\!]$
$\Leftrightarrow$ { definition of $\to$ }
$\forall (f :: \gamma_1 \leftrightarrow \gamma_2), (h :: \beta_1 \leftrightarrow \beta_2).$
$\forall ((\oplus_1), (\oplus_2)) \in [\![f \to f \to f]\!], (s_1, s_2) \in [\![h \to f]\!], (\eta_1, \eta_2) \in [\![A \to f \to f]\!],$
$((\otimes_1), (\otimes_2)) \in [\![A \to h \to h]\!], (n_1, n_2) \in [\![h]\!].$
$(g_{\beta_1,\gamma_1}\ (\oplus_1)\ s_1\ \eta_1\ (\otimes_1)\ n_1, g_{\beta_2,\gamma_2}\ (\oplus_2)\ s_2\ \eta_2\ (\otimes_2)\ n_2) \in [\![f]\!]$
$\Rightarrow$ { definition of $\to$, and assume f and h to be functions }
$\forall (f :: \gamma_1 \to \gamma_2), (h :: \beta_1 \to \beta_2).$
$((\forall a, b.\, f\ (a \oplus_1 b) = (f\ a \oplus_2 f\ b)) \wedge (\forall c.\, f\ (s_1\ c) = s_2\ (h\ c)) \wedge (\forall d, x.\, f\ (\eta_1\ d\ x) = \eta_2\ d\ (f\ x)) \wedge$
$(\forall e, y.\, h\ (e \otimes_1 y) = e \otimes_2 h\ y) \wedge h\ n_1 = n_2)$
$\Rightarrow f\ (g_{\gamma_1,\beta_1}\ (\oplus_1)\ s_1\ \eta_1\ (\otimes_1)\ n_1) = g_{\gamma_2,\beta_2}\ (\oplus_2)\ s_2\ \eta_2\ (\otimes_2)\ n_2$
$\Rightarrow$ { let $(\oplus_1) = (\cup)$, $s_1 = \{\cdot\}$, $\eta_1 = \mathit{extend}$, $(\otimes_1) = (:)$, and $n_1 = [\,]$ }
$\forall (f :: \gamma_1 \to \gamma_2), (h :: \beta_1 \to \beta_2).$
$((\forall a, b.\, f\ (a \cup b) = (f\ a \oplus_2 f\ b)) \wedge (\forall c.\, f\ \{c\} = s_2\ (h\ c)) \wedge (\forall d, x.\, f\ (\mathit{extend}\ d\ x) = \eta_2\ d\ (f\ x)) \wedge$
$(\forall e, y.\, h\ (e : y) = e \otimes_2 h\ y) \wedge h\ [\,] = n_2)$
$\Rightarrow f\ (g_{\gamma_1,\beta_1}\ (\cup)\ \mathit{extend}\ \{\cdot\}\ (:)\ [\,]) = g_{\gamma_2,\beta_2}\ (\oplus_2)\ s_2\ \eta_2\ (\otimes_2)\ n_2$
$\Rightarrow$ { let $f = (\![(\oplus_2), s_2, (\otimes_2), n_2]\!]$ and $h = \mathit{foldr}\ (\otimes_2)\ n_2$ }
$\forall d, x.\, (\![(\oplus_2), s_2, (\otimes_2), n_2]\!]\ (\mathit{extend}\ d\ x) = \eta_2\ d\ ((\![(\oplus_2), s_2, (\otimes_2), n_2]\!]\ x)$
$\Rightarrow (\![(\oplus_2), s_2, (\otimes_2), n_2]\!]\ (g_{\gamma_1,\beta_1}\ (\cup)\ \mathit{extend}\ \{\cdot\}\ (:)\ [\,]) = g_{\gamma_2,\beta_2}\ (\oplus_2)\ s_2\ \eta_2\ (\otimes_2)\ n_2$

Therefore, it is sufficient to prove that the following equation holds for all d and x.

$$(\![(\oplus), s, (\otimes), n]\!]\ (\mathit{extend}\ d\ x) = \eta\ d\ ((\![(\oplus), s, (\otimes), n]\!]\ x)$$

We prove it by induction over the size of the set x.

If x is empty, we reason as follows.

$$\begin{aligned}
(\![(\oplus), s, (\otimes), n]\!]\ (\mathit{extend}\ d\ \emptyset) &= \{ \text{ definition of } \mathit{extend} \ \} \\
&\quad (\![(\oplus), s, (\otimes), n]\!]\ \emptyset \\
&= \{ \text{ definition of } (\![(\oplus), s, (\otimes), n]\!] \ \} \\
&\quad \iota_\oplus \\
&= \{ \text{ requirement of } \eta \ \} \\
&\quad \eta\ d\ \iota_\oplus \\
&= \{ \text{ definition of } (\![(\oplus), s, (\otimes), n]\!] \ \} \\
&\quad \eta\ d\ ((\![(\oplus), s, (\otimes), n]\!])\ \emptyset)
\end{aligned}$$

For the step case, we reason as follows.

$$\begin{aligned}
(\![(\oplus), s, (\otimes), n]\!]\ (\mathit{extend}\ d\ (x \cup \{a\})) &= \{ \text{ definition of } \mathit{extend} \ \} \\
&\quad (\![(\oplus), s, (\otimes), n]\!]\ (\mathit{extend}\ d\ x \cup \{d : a\}) \\
&= \{ \text{ definition of } (\![(\oplus), s, (\otimes), n]\!] \ \} \\
&\quad (\![(\oplus), s, (\otimes), n]\!]\ (\mathit{extend}\ d\ x) \oplus s\ (d \otimes \mathit{foldr}\ (\otimes)\ n\ a) \\
&= \{ \text{ hypothesis } \} \\
&\quad \eta\ d\ ((\![(\oplus), s, (\otimes), n]\!]\ x) \oplus s\ (d \otimes \mathit{foldr}\ (\otimes)\ n\ a) \\
&= \{ \text{ premise } \} \\
&\quad \eta\ d\ ((\![(\oplus), s, (\otimes), n]\!]\ x \oplus s\ (\mathit{foldr}\ (\otimes)\ n\ a)) \\
&= \{ \text{ definition of } (\![(\oplus), s, (\otimes), n]\!] \ \} \\
&\quad \eta\ d\ ((\![(\oplus), s, (\otimes), n]\!]\ (x \cup \{a\}))
\end{aligned}$$

Then, the claim certainly holds, and we accomplish this proof. □

From Theorem 3, we can immediately obtain the following corollary.

Corollary 4 (Shortcut fusion law for maximum sum problems)**.** For a function g of the appropriate type,

$$\mathsf{maxSum}\ (\mathsf{gen}\ g) = g\ (\uparrow)\ id\ (+)\ (+)\ 0$$

holds.

Proof. Recall $\mathsf{maxSum} = (\![\uparrow, id, +, 0]\!)$ with $\iota_\uparrow = -\infty$. Thus, from Theorem 3, it is sufficient to confirm $a + (-\infty) = (-\infty)$ and $a + (r \uparrow id\ s) = (a + r) \uparrow (a + id\ s)$, which are trivial. □

Corollary 4 enables us to derive efficient algorithms in a shortcut manner. Notice that the fusion of maxSum and $\mathsf{gen}\ g$ will remove all set-concerning operations such as $\cup$ and map_{Set}. More specifically, maximum operations (namely $\uparrow$) take the place of joining operations (namely $\cup$), and thus, only the maximum value, instead of all candidates, is considered throughout the computation. In other words, Corollary 4 derives efficient programs that compute the maximum value in a greedy manner.

It is worth noticing that Corollary 4 is founded on the monotonicity of $+$ operation, that is, $x \geq y$ implies $a + x \geq a + y$. It is well-known that monotonicity is the key to efficient algorithms.

3.2 Example: Maximum Segment Sum Problem

To see how our shortcut fusion law works, let us solve the maximum segment sum [Ben86, Bir89] problem, which is a problem to compute the maximum sum of consecutive sublists. A natural specification of the problem would be the following.

$$\begin{array}{l} mss\ x = \mathsf{maxSum}\ (segs\ x) \\ \quad \textbf{where} \quad \begin{array}{lcl} segs\ [] & = & \{[]\} \\ segs\ (a:x) & = & inits\ (a:x) \cup segs\ x \\ inits\ [] & = & \{[]\} \\ inits\ (a:x) & = & \{[]\} \cup extend\ a\ (inits\ x) \end{array} \end{array}$$

What we should do is to derive the build form of *segs*. It is easy, and just replacing occurrences of constructors by parameters suffices.

$$\begin{array}{l} segs\ x = \mathsf{gen}\ (segs'\ x\ u\ s\ m\ c\ n) \\ \quad \textbf{where} \quad \begin{array}{lcl} segs'\ []\ u\ s\ m\ c\ n & = & s\ n \\ segs'\ (a:x)\ u\ s\ m\ c\ n & = & u\ (inits'\ (a:x)\ u\ s\ m\ c\ n)\ (segs'\ x\ u\ s\ m\ c\ n) \\ inits'\ []\ u\ s\ m\ c\ n & = & s\ n \\ inits'\ (a:x)\ u\ s\ m\ c\ n & = & u\ (s\ n)\ (m\ a\ (inits'\ x\ u\ s\ m\ c\ n)) \end{array} \end{array}$$

Then, $segs'\ x$ exactly has the required type.

Now that the build form is obtained, Corollary 4 immediately derives the following program for *mss* after inlining parameters.

$$\begin{array}{l} mss\ x = segs''\ x \\ \quad \textbf{where} \quad \begin{array}{lcl} segs''\ [] & = & 0 \\ segs''\ (a:x) & = & inits''\ (a:x) \uparrow segs''\ x \\ inits''\ [] & = & 0 \\ inits''\ (a:x) & = & 0 \uparrow (a + inits''\ x) \end{array} \end{array}$$

Recall that the original definition is an $O(n^3)$-time algorithm, where n is the length of the input list. The derived algorithm is $O(n^2)$, and certainly efficiency is improved.

In fact, the derived program is not efficient enough, and there is a linear-time algorithm. Notice that $inits''$ with the same parameter is called many times, and this is the source of inefficiency. Here is the place that memoization technique plays its role. We apply tupling transformation [HITT97, CKJ06], which is a way to achieve memoization, and obtain the following linear-time program.

$$\begin{array}{l} mss\ x = \textbf{let}\ (m, i) = aux\ x\ \textbf{in}\ m \\ \quad \textbf{where} \quad \begin{array}{lcl} aux\ [] & = & (0, 0) \\ aux\ (a:x) & = & \textbf{let}\ (m, i) = aux\ x \\ & & \quad\ i' = 0 \uparrow (a + i) \\ & & \textbf{in}\ (i' \uparrow m, i') \end{array} \end{array}$$

The auxiliary function *aux* is defined as $aux\ x = (segs''\ x, inits''\ x)$, and computes the result of $segs''$ and $inits''$ in the same time. Then, since memoization is completed, the derived program runs in time proportional to the length of the list, and actually the program is equivalent to the efficient program firstly introduced by Bentley [Ben86].

Why Corollary 4 cannot derive the efficient one? The reason is the original definition of *segs* is not sufficiently efficient: *segs* will call *inits* of the same parameter many times, and memoization is necessary for efficiency! Corollary 4 cannot do everything. It will yields relatively inefficient programs from relatively inefficient ones. However, this issue would not be very serious, because of the following two reasons.

- A set of candidates commonly follows some typical patterns, such as subsets of a set. Thus, by providing efficient implementations of the typical patterns, together with their build forms, we can solve many optimization problems without worrying such issues.
- Practically, calling the same function with the same parameter is the source of inefficiency. Therefore, combining our technique with existing memoization techniques would resolve the inefficiency.

3.3 Shortcut Fusion Law for Enumerating Maximals

Corollary 4 enables us to derive efficient algorithms for free, once the build form of the candidate generator is specified. However, Corollary 4 is not satisfactory because the resulted programs yield just a summation rather than a list. What we want is not the maximum sum but the list of the maximum sum.

Here is an issue — if there are plural lists having the maximum sum, which should be resulted in? This issue seems insignificant at first sight but actually important. From practical viewpoint, we can hardly formalize our criterion of optimality by a single maximization operation. We may want to obtain the best in a criterion among those being the best in another criterion; besides, we may want to find out the best one by hand from those being wroth considering. For such purpose, enumerating all of the best solutions are useful. From theoretical viewpoint, it is a common idiom that we first enumerate candidates that are worth considering and find the best one afterward. In summary, it is important to consider enumeration of all candidates that is worth considering.

Now let us introduce a notion of *maximals*. A maximal of a set is an element that is not strictly smaller than others. In other words, a maximal corresponds to a candidate that is worth considering, because it is not worse than others. Formally, the maximal-extracting function on a preorder $\preceq$, denoted by $\mathsf{maximals}_{\preceq}$ is defined as follows.

$$\mathsf{maximals}_{\preceq}\ X \stackrel{\text{def}}{=} \{a \mid a \in X \wedge \forall b \in X.\, b \preceq a \Rightarrow a \preceq b\}$$

For convenience, we also define the binary-operator version $\overline{\uparrow}_{\preceq}$, as $X\ \overline{\uparrow}_{\preceq}\ Y \stackrel{\text{def}}{=} \mathsf{maximals}_{\preceq}\ (X \cup Y)$. By definition, the operator $\overline{\uparrow}_{\preceq}$ is associative, commutative, and idempotent.

Well, let us introduce a shortcut fusion law that is similar to Corollary 4 but dealing with computations to obtaining a set of list, rather than those to obtaining a list. The following corollary proves that enumerating maximals can be done in a shortcut manner.

Corollary 5. For a function g of the appropriate type,

$$\mathsf{maximals}_{\preceq}\ (\mathsf{gen}\ g) = g\ (\overline{\uparrow}_{\preceq})\ \{\cdot\}\ (\lambda a.\, \mathsf{maximals}_{\preceq} \circ extend\ a)\ (:)\ []$$

holds, provided that $(x \preceq y \wedge \neg(y \preceq x)) \Rightarrow ((a : x) \preceq (a : y) \wedge \neg((a : y) \preceq (a : x)))$ holds.

Proof. Notice $\mathsf{maximals}_{\preceq} = (\!|\overline{\uparrow}_{\preceq}, \{\cdot\}, (:)\ []|\!)$ with $\iota_{\overline{\uparrow}_{\preceq}} = \emptyset$. Thus, it is sufficient to confirm the premise of Theorem 3, and the only nontrivial one is the following equation.

$$\mathsf{maximals}_{\preceq}\ (extend\ a\ (X\ \overline{\uparrow}_{\preceq}\ \{y\})) = \mathsf{maximals}_{\preceq}\ (extend\ a\ X)\ \overline{\uparrow}_{\preceq}\ \{a : y\}$$

Now we reason as follows.

$$\begin{aligned}
\mathsf{maximals}_{\preceq}\ (extend\ a\ X)\ \overline{\uparrow}_{\preceq}\ \{a : y\} &= \{\text{ definition of } \overline{\uparrow}_{\preceq}\ \} \\
&\quad \mathsf{maximals}_{\preceq}\ (\mathsf{maximals}_{\preceq}\ (extend\ a\ X) \cup \{a : y\}) \\
&= \{\text{ Lemma 3.26 of [Mor09]}\ \} \\
&\quad \mathsf{maximals}_{\preceq}\ (extend\ a\ X \cup \{a : y\})
\end{aligned}$$

Then, because of Lemma 3.28 of [Mor09] and the fact that $(\mathit{extend}\ a\ X \cup \{a : y\}) \supseteq \mathit{extend}\ a\ (X \overline{\uparrow}_{\preceq} \{y\})$ holds, it is sufficient to prove $\mathsf{maximals}_{\preceq}(\mathit{extend}\ a\ X \cup \{a : y\}) \subseteq \mathit{extend}\ a\ (X \overline{\uparrow}_{\preceq} \{y\})$, which is proved as follows.

$$
\begin{array}{ll}
& \mathsf{maximals}_{\preceq}\ (\mathit{extend}\ a\ X \cup \{a : y\}) \\
= & \{\ \text{unfolding definitions}\ \} \\
& \{a : w \mid w \in (X \cup \{y\}) \wedge \forall v \in (X \cup \{y\}).\, (a : v) \preceq (a : w) \Rightarrow (a : w) \preceq (a : v)\} \\
\subseteq & \{\ \text{contraposition of the premise: } (\neg((a : v) \preceq (a : w)) \vee (a : w) \preceq (a : v)) \Rightarrow (\neg(v \preceq w) \vee w \preceq v)\ \} \\
& \{a : w \mid w \in (X \cup \{y\}) \wedge \forall v \in (X \cup \{y\}).\, \neg(v \preceq w) \vee w \preceq v\} \\
= & \{\ \text{folding definitions}\ \} \\
& \mathit{extend}\ a\ (X \overline{\uparrow}_{\preceq} \{y\})
\end{array}
$$

□

Therefore, the issue left is how to derive an order satisfying the premise. Here we would like to name the premise for convenience.

Definition 6 (monotonicity)**.** Extending lists is said to be *monotonic* on a order $\preceq$ if $x \preceq y \Rightarrow (a : x) \preceq (a : y)$ holds for all x, y, and a. Extending lists is said to be *strictly monotonic* on a order $\preceq$ if $(x \preceq y \wedge \neg(y \preceq x)) \Rightarrow ((a : x) \preceq (a : y) \wedge \neg((a : y) \preceq (a : x)))$ holds. Extending lists is said to be *completely monotonic* on a order $\preceq$ if it is both monotonic and strictly monotonic. □

4 Deriving Monotonic Orders

We have formalized derivation of efficient algorithms by shortcut fusion laws. For utilizing the laws, it is necessary to prepare two kinds of functions. One is a well-structured producer, that is, a function of "build form". For derivation of build forms, existing results [LS95, Chi99, YHT05] will be helpful; in addition, we can prepare build forms of typical list-enumeration patterns. The other is a well-structured consumer, that is maximization operation specified by an order on which extending lists is monotonic. Derivation of well-structured consumers is a problem, because it is not easy to confirm or obtain monotonicity condition for involved problems, such as problem having additional requirements. For example, consider finding the maximum sum segment whose length is in a given lower/upper bounds. For utilizing Corollary 5, we need to prepare an order that corresponds to our object and monotonicity condition holds on it. However, it is not trivial to deal with the requirements about length.

In this section, we consider such involved cases and introduce program transformation rules that enable us to derive monotonicity condition systematically. We refer to many supplemental lemmas from the author's thesis [Mor09], which provide a formal background of this paper.

4.1 Program Transformations for Obtaining Monotonicity Conditions

Our strategy is to construct monotonicity conditions constructively: Starting from a trivial order on which extending lists is monotonic, we construct involved orders without breaking monotonicity.

First, let us consider two trivial orders that satisfy monotonicity condition: summation and the lexicographic ordering.

Lemma 7. Extending lists is completely monotonic on the order $\leq_{sum}$.

Proof. It is sufficient to prove $\mathit{sum}\ x \leq \mathit{sum}\ y \Leftrightarrow \mathit{sum}\ (a : x) \leq \mathit{sum}\ (a : y)$, and it is evident from the definition of sum. □

Lemma 8. Extending lists is completely monotonic on the order $\leq$ that denotes the lexicographic ordering of sequences.

Proof. It is sufficient to prove $x \leq y \Leftrightarrow (a : x) \leq (a : y)$, and it is evident from the definition of $\leq$. □

Though trivial, they are useful in practice. Moreover, we can extend them to more involved ones.

Lemma 9. Extending lists is completely monotonic on an order $\preceq_{map_{List}\ f}$ if so is on $\preceq$.

Proof. Let $x' = map_{List}\ f\ x$ and $y' = map_{List}\ f\ y$. Then, it is sufficient to prove both of $x' \preceq y' \Rightarrow (f\ a : x') \preceq (f\ a : y')$ and $(x' \preceq y' \wedge \neg(y' \preceq x')) \Rightarrow ((f\ a : x') \preceq (f\ a : y') \wedge \neg((f\ a : y') \preceq (f\ a : x')))$, and they are evident from complete monotonicity of $\preceq$. □

Lemma 10. Given an order $\preceq$ on which extending lists is completely monotonic, let $\ll$ be the order such that $\ll \stackrel{\text{def}}{=} \preceq_{snd \circ mapAccumR\ f\ e}$ and f' be a function such that $f'\ a\ s \stackrel{\text{def}}{=} fst\ (f\ s\ a)$. Then, $\mathsf{maximals}_{\ll} = \mathsf{maximals}_{\ll} \circ \mathsf{maximals}_{\ll \cap =_{foldr\ f'\ e}}$ holds, and extending lists is completely monotonic on $\ll \cap =_{foldr\ f'\ e}$.

Proof. The equation, $\mathsf{maximals}_{\ll} = \mathsf{maximals}_{\ll} \circ \mathsf{maximals}_{\ll \cap =_{foldr\ f'\ e}}$, is evident from Lemma 3.29 of [Mor09]. Then, we prove complete monotonicity. In the following, ϕ_x, ϕ_y, α_x, and α_y respectively denote $fodlr\ f'\ e\ x$, $foldr\ f'\ e\ y$, $snd\ (mapAccumR\ f\ e\ x)$, and $snd\ (mapAccumR\ f\ e\ y)$. First, we prove monotonicity.

$$
\begin{array}{lll}
x\ (\ll \cap =_{foldr\ f'\ e})\ y & \Leftrightarrow & \{ \text{ definition of } \cap \text{ and } \ll \ \} \\
 & & \alpha_x \preceq \alpha_y \wedge \phi_x = \phi_y \\
 & \Rightarrow & \{\ f' \text{ is a function } \} \\
 & & \alpha_x \preceq \alpha_y \wedge f'\ b\ \phi_x = f'\ b\ \phi_y \\
 & \Rightarrow & \{ \text{ monotonicity of } \preceq \} \\
 & & (f'\ b\ \phi_x : \alpha_x) \preceq (f'\ b\ \phi_y : \alpha_y) \wedge f'\ b\ \phi_x = f'\ b\ \phi_y \\
 & \Leftrightarrow & \{\ foldr\ f'\ e\ z = fst\ (mapAccumR\ f\ e\ z) \text{ holds for all } z\ \} \\
 & & (b : x) \preceq_{snd \circ mapAccumR\ f\ e} (b : y) \wedge f'\ b\ \phi_x = f'\ b\ \phi_y \\
 & \Leftrightarrow & \{ \text{ definition of } \cap \text{ and } \ll \ \} \\
 & & (b : x)\ (\ll \cap =_{foldr\ f'\ e})\ (b : y)
\end{array}
$$

Next, we prove strict monotonicity.

$$
\begin{array}{ll}
 & x\ (\ll \cap =_{foldr\ f'\ e})\ y \wedge \neg (y\ (\ll \cap =_{foldr\ f'\ e})\ x) \\
\Leftrightarrow & \{ \text{ definition of } \cap \text{ and } \ll \ \} \\
 & \alpha_x \preceq \alpha_y \wedge \phi_x = \phi_y \wedge \neg(\alpha_y \preceq \alpha_x) \\
\Rightarrow & \{\ f' \text{ is a function } \} \\
 & \alpha_x \preceq \alpha_y \wedge f'\ b\ \phi_x = f'\ b\ \phi_y \wedge \neg(\alpha_y \preceq \alpha_x) \\
\Rightarrow & \{ \text{ strict monotonicity of } \preceq \} \\
 & (f'\ b\ \phi_x : \alpha_x) \preceq (f'\ b\ \phi_y : \alpha_y) \wedge f'\ b\ \phi_x = f'\ b\ \phi_y \wedge \neg(f'\ b\ \phi_y : \alpha_y \preceq f'\ b\ \phi_x : \alpha_x) \\
\Leftrightarrow & \{\ foldr\ f'\ e\ z = fst\ (mapAccumR\ f\ e\ z) \text{ holds for all } z\ \} \\
 & (b : x) \preceq_{snd \circ mapAccumR\ f\ e} (b : y) \wedge f'\ b\ \phi_x = f'\ b\ \phi_y \wedge \neg((b : y) \preceq_{snd \circ mapAccumR\ f\ e} (b : x)) \\
\Leftrightarrow & \{ \text{ definition of } \cap \text{ and } \ll \ \} \\
 & (b : x)\ (\ll \cap =_{foldr\ f'\ e})\ (b : y) \wedge \neg((b : y)\ (\ll \cap =_{foldr\ f'\ e})\ (b : x))
\end{array}
$$

□

Lemmas 9 and 10 provide a way to solve "weighed" problems, such as maximum weighed-sum problems. If the weight is specified by *map* or *mapAccumR*, the lemmas immediately provide an order that leads to shortcut fusion.

There is another way to construct involved orders, that is lexicographic composition of two orders. Lexicographic composition is useful to solve multi-objective optimization problems, in which we would like to find the best one among those the best in another criterion. We can solve such problems by the following lemma.

Lemma 11. The following equation holds.

$$\mathsf{maximals}_{\ll} \circ \mathsf{maximals}_{\preceq} = \mathsf{maximals}_{\ll} \circ \mathsf{maximals}_{\ll ; \preceq}$$

In addition, extending lists is completely monotonic on the order $\ll ; \preceq$ if so is on both of $\ll$ and $\preceq$.

Proof. The equation above holds from Lemma 3.38 of [Mor09], where note that both of $\mathsf{maximals}_{\preceq}\ X \supseteq \mathsf{maximals}_{\ll ; \preceq}\ X$ and $\mathsf{maximals}_{\ll}\ (\mathsf{maximals}_{\preceq}\ X) \subseteq \mathsf{maximals}_{\ll ; \preceq}\ X$ hold. The complete monotonicity is proved in Lemma 4.14 of [Mor09]. □

Well, let us consider problems in which solutions should satisfy some requirements. Though there is generally little hope to solve problems efficiently when some requirements are imposed, we can derive efficient algorithms for specific forms of restriction, as the following lemma shows.

Lemma 12. The following equation holds.

$$\mathsf{maximals}_{\preceq} \circ \mathit{filter}\ (p \circ \mathit{foldr}\ f\ e) = \mathsf{maximals}_{\preceq} \circ \mathit{filter}\ (p \circ \mathit{foldr}\ f\ e) \circ \mathsf{maximals}_{\preceq \cap =_{foldr\ f\ e}}$$

Moreover, extending lists is completely monotonic on $\preceq \cap =_{foldr\ f\ e}$ if so is on $\preceq$.

Proof. The equation can be easily proved from Lemmas 3.29 and 4.28 of [Mor09]. For proving the complete monotonicity, it is sufficient to confirm that extending lists is completely monotonic on $=_{\mathit{foldr}\ f\ e}$ because of Lemma 4.11 of [Mor09]. Since $=_{\mathit{foldr}\ f\ e}$ is an equivalence relation, proving $\mathit{foldr}\ f\ e\ x = \mathit{foldr}\ f\ e\ y \Rightarrow \mathit{foldr}\ f\ e\ (a : x) = \mathit{foldr}\ f\ e\ (a : y)$ suffices, which is apparent from the definition of *foldr*. □

Lemma 12 shows an sufficient condition to propagate maximization operations over filtering operations; then, we can to discard worse candidates while generating them.

We can obtain more efficient programs when a candidate cannot recover violated requirements by extending itself. For example, if solutions should shorter than a given limit, then candidates longer than the limit are useless. Such requirements frequently occur in practice, and we would like to prepare a special rule for them.

Lemma 13. Given a predicate $p = q \circ \mathit{foldr}\ f\ e$ such that $\neg(p\ x)$ implies $\neg(p\ (a : x))$, consider the following order $\lessdot$.

$$\begin{aligned} x \lessdot y \quad \stackrel{\text{def}}{=} \quad & (p\ x \wedge p\ y \wedge x =_{\mathit{foldr}\ f\ e}\ y) \vee \\ & (\neg(p\ x) \wedge p\ y) \vee \\ & (\neg(p\ x) \wedge \neg(p\ y) \wedge \mathit{size}\ \{a \mid a \in \mathit{tails}\ x \wedge \neg(p\ a)\} \geq \mathit{size}\ \{b \mid b \in \mathit{tails}\ y \wedge \neg(p\ b)\}) \end{aligned}$$

Then, $\mathit{filter}_{Set}\ p = \mathit{filter}_{Set}\ p \circ \mathsf{maximals}_{\lessdot}$ holds; besides, extending lists is completely monotonic on $\lessdot$.

Proof. The equation is evident because $\mathsf{maximals}_{\lessdot}$ never discards elements satisfying p.

For confirming complete monotonicity condition, we consider the following three cases, where let $|x|_p = \mathit{size}\ \{a \mid a \in \mathit{tails}\ x \wedge \neg(p\ a)\}$.

First, assume that $p\ x \wedge p\ y \wedge x =_{\mathit{foldr}\ f\ e}\ y$ holds, which implies $x \lessdot y \wedge y \lessdot x$. Here note that $|x|_p = |y|_p = 0$ holds because $(p\ (a : z))$ implies $p\ z$ from the premise. Then, $(a : x) \lessdot (a : y) \wedge (a : y) \lessdot (a : x)$ holds, because that since $p = q \circ \mathit{foldr}\ f\ e$ and $\mathit{foldr}\ f\ e\ x = \mathit{foldr}\ f\ e\ y$, either (i) $p\ (a : x) \wedge p\ (a : y) \wedge \mathit{foldr}\ f\ e\ (a : x) = \mathit{foldr}\ f\ e\ (a : y)$ or (ii) $\neg(p\ (a : x)) \wedge \neg(p\ (a : y)) \wedge |a : x|_p = |a : y|_p = 1$ holds.

Second, assume that $\neg(p\ x) \wedge p\ y$ holds, which implies $x \lessdot y \wedge \neg(y \lessdot x)$. Then, $(a : x) \lessdot (a : y) \wedge \neg((a : y) \lessdot (a : x))$ holds, because both of $\neg(p\ (a : x))$ and $|a : x|_p > |a : y|_p$ hold from $\neg(p\ x) \Rightarrow \neg(p\ (a : x))$.

Third, assume that $\neg(p\ x) \wedge \neg(p\ y)$. Then, $x \lessdot y \Leftrightarrow (a : x) \lessdot (a : y)$ holds, because both of $\neg(p\ (a : x)) \wedge \neg(p\ (a : y))$ and $|x|_p \geq |y|_p \Leftrightarrow |a : x|_p \geq |a : y|_p$ holds from $\neg(p\ x) \Rightarrow \neg(p\ (a : x))$.

In all, extending lists is completely monotonic on $\lessdot$. □

Although the definition of $\lessdot$ is a bit complicated, the $\mathsf{maximals}_{\lessdot}$ in the lemma above does essentially the same computation as *filter p*: it discards all elements that does not satisfy the requirement p, whenever there are at least one element that satisfies p. Moreover, it can cooperate with our shortcut fusion laws thanks to the complete monotonicity of $\lessdot$.

4.2 Example: Length-Constrained Maximum Sum Segment Problem

Now let us consider the length-constrained maximum sum segment [Mu08] problem. The problem is to find the segment of maximum sum among those whose length is in the given range. Unconstrained maximum sum segment will be very long or a short one containing very large number, though such results might be undesirable. Therefore, it is natural to consider segments of length in a specific range. The problem can be formalized as follows, in which l and u respectively stand for the upper and lower bound.

$$\mathit{lcmss}\ l\ u\ x = \mathsf{maximals}_{\leq_{sum}}\ (\mathit{filter}_{Set}\ (\lambda x.\, l \leq \mathit{length}\ x \leq u)\ (\mathit{segs}\ x))$$

Since we have known the build form of *segs* x, what we need is an order on which extending lists is completely monotonic. Though extending lists is completely monotonic on $\leq_{sum}$ by Lemmas 7, the filtering operation blocks to connect $\mathsf{maximals}_{\leq_{sum}}$ to *segs*, and we cannot apply the shortcut fusion law. Here, notice that $(\lambda x.\, l \leq \mathit{length}\ x \leq u)$ is equivalent to $(\lambda v.\, l \leq v \leq u) \circ \mathit{length}$ and $\mathit{length} = \mathit{foldr}\ (\lambda a\, r.\, 1 + r)\ 0$ holds. Then, from Lemma 12, the following equation holds.

$$\begin{aligned} & \mathsf{maximals}_{\leq_{sum}} \circ \mathit{filter}_{Set}\ (\lambda x.\, l \leq \mathit{length}\ x \leq u) = \\ & \qquad \mathsf{maximals}_{\leq_{sum}} \circ \mathit{filter}_{Set}\ (\lambda x.\, l \leq \mathit{length}\ x \leq u) \circ \mathsf{maximals}_{\leq_{sum} \cap =_{\mathit{length}}} \end{aligned}$$

Moreover, Lemma 12 proves that extending lists is completely monotonic on $\leq_{sum} \cap =_{length}$. Therefore, Corollary 5 derives efficient program.

Let us summarize the calculation.

$$
\begin{aligned}
&\mathit{lcmss}\ l\ u\ x \\
&\quad = \quad \{ \text{ definition } \} \\
&\qquad \mathsf{maximals}_{\leq_{sum}}\ (\mathit{filter}_{Set}\ (\lambda x.\, l \leq \mathit{length}\ x \leq u)\ (\mathit{segs}\ x)) \\
&\quad = \quad \{\ (\lambda x.\, l \leq \mathit{length}\ x \leq u) = (\lambda v.\, l \leq v \leq u) \circ \mathit{length}\ \} \\
&\qquad \mathsf{maximals}_{\leq_{sum}}\ (\mathit{filter}_{Set}\ ((\lambda v.\, l \leq v \leq u) \circ \mathit{length})\ (\mathit{segs}\ x)) \\
&\quad = \quad \{\ \mathit{length} = \mathit{foldr}\ (\lambda a\, r.\, 1 + r)\ 0, \text{ and Lemma 12 } \} \\
&\qquad \mathsf{maximals}_{\leq_{sum}}\ (\mathit{filter}_{Set}\ ((\lambda v.\, l \leq v \leq u) \circ \mathit{length})\ (\mathsf{maximals}_{\leq_{sum} \cap =_{length}}\ (\mathit{segs}\ x))) \\
&\quad = \quad \{\ \mathit{segs}\ x = \mathsf{gen}\ (\mathit{segs}'\ x), \text{ and Corollary 5 } \} \\
&\qquad \mathsf{maximals}_{\leq_{sum}}\ (\mathit{filter}_{Set}\ ((\lambda v.\, l \leq v \leq u) \circ \mathit{length}) \\
&\qquad\qquad\qquad (\mathit{segs}'\ x\ \overline{\uparrow}_{\leq_{sum} \cap =_{length}}\ \{\cdot\}\ (\lambda a.\, \mathsf{maximals}_{\leq_{sum} \cap =_{length}} \circ \mathit{extend}\ a)\ (:)\ [\,]))
\end{aligned}
$$

In the derived program, only restricted number of candidates are considered: a candidate is to be retained only if it has the maximum sum among those candidate of the same length. Thus, derived program runs in $\mathrm{O}(n^2)$ time for a list of length n if we does memoization.

By using Lemma 13, we can derive a bit more efficient one. Observe that $\mathit{filter}_{Set}\ (\lambda x.\, l \leq \mathit{length}\ x \leq u)$ is equivalent to $\mathit{filter}_{Set}\ (\lambda x.\, l \leq \mathit{length}\ x) \circ \mathit{filter}_{Set}\ (\lambda x.\, \mathit{length}\ x \leq u)$, and $\neg(\mathit{length}\ x \leq u)$ implies $\neg(\mathit{length}\ (a : x) \leq u)$. Therefore, we can use Lemma 13 for the latter filtering operation, and then, the derived program discards candidates being longer than u. This program runs in $\mathrm{O}(u \cdot n)$ time with memoization.

For efficient implementation, memoization is necessary. For example, a naive implementation of *lcmss* above will iteratively compute length and summation of lists for every comparison by $\mathsf{maximals}_{\leq_{sum} \cap =_{length}}$. To avoid such inefficiency, we would like to introduce another shortcut fusion law that is very similar to Corollary 5 but accomplishes memoization in the same time.

Theorem 14. Assume that extending list is completely monotonic on an order $\lessdot$ defined by $a \lessdot b \overset{\text{def}}{\Leftrightarrow} (\mathit{foldr}\ f\ e\ a) \ll (\mathit{foldr}\ f\ e\ b)$. Then, for a function g having the appropriate type, the following equation holds.

$$
\begin{aligned}
&\mathsf{maximals}_{\lessdot}\ (\mathsf{gen}\ g) = \mathit{map}_{Set}\ \mathit{fst}\ (g\ \overline{\uparrow}_{\preceq}\ \{\cdot\}\ \eta\ f'\ ([\,], e)) \\
&\quad \mathsf{where}\ (\_, a) \preceq (\_, b) = a \ll b \\
&\qquad\qquad f'\ a\ (x, r) = (a : x, f\ a\ r) \\
&\qquad\qquad \eta\ a = \mathsf{maximals}_{\preceq} \circ \mathit{map}_{Set}\ (f'\ a)
\end{aligned}
$$

Proof. It is easily confirmed by induction that $\mathsf{maximals}_{\lessdot} = \mathit{map}_{Set}\ \mathit{fst} \circ (\!|\overline{\uparrow}_{\preceq}, \{\cdot\}, f', ([\,], e)|\!)$ holds. Moreover, $(\!|\overline{\uparrow}_{\preceq}, \{\cdot\}, f', ([\,], e)|\!) = (\!|\overline{\uparrow}_{\lessdot_{fst}}, \{\cdot\}, f', ([\,], e)|\!)$ holds, because $\mathit{foldr}\ f\ e\ x = a$ holds for each element (x, a) throughout the computation. Then, since extending lists is completely monotonic on $\lessdot$, the proof follows from Corollary 5. □

5 A Library for Enumerating Optimal Lists

Based on the theory we have developed, we implemented a Haskell library for obtaining optimal lists. The library consists of functions that would be useful for describing criterion of desirable lists; moreover, the library consists of a set of rewriting rules, called *RULES pragma* [PTH01], that derive efficient algorithms automatically.

Here let us give a brief introduction of RULES pragma. RULES pragma is an extensional functionality of Glasgow Haskell Compiler[2]. By using RULES pragma, we can provide rewriting rules that will be applied to programs on their compilation. For example, consider the following rewriting rule, where $\rightsquigarrow$ denotes left-to-right rewriting.

$$\forall f\ g.\, \mathit{map}_{List}\ f \circ \mathit{map}_{List}\ g \rightsquigarrow \mathit{map}_{List}\ (f \circ g)$$

If the rewrite rule is specified and the compiler find a composition of two map_{List}, e.g., $\mathit{map}_{List}\ (+1) \circ \mathit{map}_{List}\ (*3)$, the composition will be compiled as $\mathit{map}_{List}\ ((+1) \circ (*3))$.

[2]The Glasgow Haskell Compiler: available from `http://www.haskell.org/ghc/`.

5.1 Functions for Enumerating Lists

Now let us introduce the functions in our library.

First of all, our library contains the function gen that is the key to shortcut fusion and stands for enumeration of candidates.

$$\mathsf{gen} :: \forall a.\, Eq\ a \Rightarrow (\forall \beta\ \gamma.\, (\gamma \to \gamma \to \gamma) \to (\beta \to \gamma) \to (a \to \gamma \to \gamma) \to (a \to \beta \to \beta) \to \beta \to \gamma) \to \{[a]\}$$
$$\mathsf{gen}\ g \stackrel{\text{def}}{=} g\ (\cup)\ \{\cdot\}\ \mathit{extend}\ (:)\ []$$

The function *gen* is the same as that we have introduced. Based on the definition of gen above, we implemented build forms of several typical lists-enumeration patterns, such as *inits*, *tails*, *segs*, *subsequences*, *permutations*, and so on. Therefore, we can solve many problems without caring derivation of build forms.

Next, let us introduce functions for filtering out undesirable candidates. Several maximal-enumerating functions are prepared in our library.

$$\mathsf{maxBySum} :: \forall a.\, (\mathit{Num}\ a, \mathit{Ord}\ a) \Rightarrow \{[a]\} \to \{[a]\}$$
$$\mathsf{maxBySum} \stackrel{\text{def}}{=} \mathsf{maximals}_{\leq_{sum}}$$

$$\mathsf{maxByMapSum} :: \forall a\ b.\, (\mathit{Num}\ b, \mathit{Ord}\ b) \Rightarrow (a \to b) \to \{[a]\} \to \{[a]\}$$
$$\mathsf{maxByMapSum}\ f \stackrel{\text{def}}{=} \mathsf{maximals}_{\leq_{sum} \circ map_{List}\ f}$$

$$\mathsf{maxByAccumSum} :: \forall a\ b\ c.\, (\mathit{Ord}\ b, \mathit{Num}\ c, \mathit{Ord}\ c) \Rightarrow (b \to a \to (b, c)) \to b \to \{[a]\} \to \{[a]\}$$
$$\mathsf{maxByAccumSum}\ f\ e \stackrel{\text{def}}{=} \mathsf{maximals}_{\leq_{sum} \circ snd \circ mapAccumR\ f\ e}$$

$$\mathsf{maxByLexico} :: \forall a.\, \mathit{Ord}\ a \Rightarrow \{[a]\} \to \{[a]\}$$
$$\mathsf{maxByLexico} \stackrel{\text{def}}{=} \mathsf{maximals}_{\leq}$$

$$\mathsf{maxByMapLexico} :: \forall a\ b.\, \mathit{Ord}\ b \Rightarrow (a \to b) \to \{[a]\} \to \{[a]\}$$
$$\mathsf{maxByMapLexico}\ f \stackrel{\text{def}}{=} \mathsf{maximals}_{\leq_{map_{List}\ f}}$$

$$\mathsf{maxByAccumLexico} :: \forall a\ b\ c.\, (\mathit{Ord}\ b, \mathit{Ord}\ c) \Rightarrow (b \to a \to (b, c)) \to b \to \{[a]\} \to \{[a]\}$$
$$\mathsf{maxByAccumLexico}\ f\ e \stackrel{\text{def}}{=} \mathsf{maximals}_{\leq_{snd \circ mapAccumR\ f\ e}}$$

The function maxBySum enumerates lists of the maximum summation. The functions maxByMapSum and maxByAccumSum take additional functions to compute the weight of each element, and enumerate lists of the maximum weighted summation. The functions maxByLexico, maxByMapLexico, and maxByAccumLexico are similar but use the lexicographic ordering instead of weighted summations. They are designed so that we can obtain orders on which extending lists is completely monotonic by Lemmas 7–10[3]. Their duals, namely minBySum, minByMapSum, minByAccumSum, minByLexico, minByMapLexico, and minByAccumLexico, are also implemented.

For enumerating lists satisfying a specific properties, we prepare the following two functions.

$$\mathsf{constraint} :: \forall a\ b.\, \mathit{Ord}\ a \Rightarrow (a \to \mathit{Bool}) \to (b \to a \to a) \to a \to \{[b]\} \to \{[b]\}$$
$$\mathsf{constraint}\ p\ f\ e \stackrel{\text{def}}{=} \mathit{filter}_{Set}\ (p \circ \mathit{foldr}\ f\ e)$$

$$\mathsf{always} :: \forall a\ b.\, \mathit{Ord}\ a \Rightarrow (a \to \mathit{Bool}) \to (b \to a \to a) \to a \to \{[b]\} \to \{[b]\}$$
$$\mathsf{always}\ p\ f\ e \stackrel{\text{def}}{=} \mathit{filter}_{Set}\ (\mathit{all}_{Set}\ (p \circ \mathit{foldr}\ f\ e) \circ \mathit{tails})$$

The function constraint discards lists that do not satisfy a requirement, and is designed so that we can utilize Lemma 12. The function always is similar to constraint, but requires all its tail segments to satisfy the requirement. An important property is that $\neg(\mathit{all}_{Set}\ (p \circ \mathit{foldr}\ f\ e)\ (\mathit{tails}\ x))$ implies $\neg(\mathit{all}_{Set}\ (p \circ \mathit{foldr}\ f\ e)\ (\mathit{tails}\ (a : x)))$; thus, we can utilize Lemma 13, and derive a more efficient program than the case of constraint.

[3] In fact, the type requirements such as $(\mathit{Num}\ a, \mathit{Ord}\ a)$ are insufficient to utilize the lemmas, because the type requirements does not guarantee that $+$ is completely monotonic on $\leq$. While it, complete monotonicity holds for all types that are available in the standard library and instances of both of *Num* and *Ord* — such as *Int*, *Rational* and so on (yet, we neglect peculiar cases such as overflowing).

5.2 Rewriting Rules

For deriving efficient programs, we would like to encode Corollary 5 together with Lemmas 7–13 as a rewriting rules. For this purpose, here we introduce another function maxIR that provides an intermediate representation of derivation.

$$\begin{aligned}&\mathsf{maxIR} :: \forall a\ b.\, (a \to b \to b) \to b \to \mathit{CMP}\ b \to (\{[a]\} \to \{[a]\}) \to \{[a]\} \to \{[a]\}\\ &\mathsf{maxIR}\ f\ e\ (\preceq)\ k \stackrel{\text{def}}{=} k \circ \mathsf{maximals}_{\preceq_{\mathit{foldr}\ f\ e}}\end{aligned}$$

The function maxIR $f\ e\ (\preceq)\ k$ computes maximals by using the given comparison operator $\preceq$ on values of $\mathit{foldr}\ f\ e$; in addition, it takes a finalizer k that will be applied after the maximization. The $\mathit{CMP}\ b$ is the type of comparison operator for a set b, and we will explain in detail in the next subsection. For guaranteeing the correctness of rewrite rules we will introduce, we require following two properties for maxIR. For maxIR $f\ e\ (\preceq)\ k$, (i) extending lists must be completely monotonic on $\preceq_{\mathit{foldr}\ f\ e}$, and (ii) for all order $\ll$, $\mathsf{maximals}_{\ll} \circ k \circ \mathsf{maximals}_{\preceq_{\mathit{foldr}\ f\ e}} = \mathsf{maximals}_{\ll} \circ k \circ \mathsf{maximals}_{\ll;\preceq_{\mathit{foldr}\ f\ e}}$ must hold.

Since function maxIR is a slight extension of maximals, it is easy to translate our library functions to it. Here we only introduce rules for maximization functions for weighed sums. Rules for the other maximization/minimization functions are also implemented, and they are very similar and thus omitted.

First, maxBySum can be rewritten into maxIR by the following rewriting rule.

$$\mathsf{maxBySum} \rightsquigarrow \mathsf{maxIR}\ (+)\ 0\ (\leq)\ \mathit{id} \tag{1}$$

The translation is straightforward, and lists are compared by its summation $\leq_{\mathit{foldr}\ (+)\ 0}$. Since no finalizing computation is necessary, the fourth parameter is the identity function id. It is worth noting that the derived maxIR certainly satisfies the requirements. The first requirement is complete monotonicity, which can be confirmed by Lemma 7. The second requirement can be easily confirmed from Lemma 11.

Similarly, we can rewrite maxByMapSum into maxIR

$$\mathsf{maxByMapSum}\ f \rightsquigarrow \mathsf{maxIR}\ (\lambda a\, r.\, f\ a + r)\ 0\ (\leq)\ \mathit{id} \tag{2}$$

As the same as the previous case, we can easily confirm the requirements of maxIR by Lemmas 7, 9, and 11.

For maxByAccumSum, we will introduce a bit indirect rewriting rule for guaranteeing monotonicity condition, as follows. Note that maxByAccumSum′ is an alias of maxByAccumSum and introduced to avoid infinite rewriting.

$$\begin{aligned}&\mathsf{maxByAccumSum}\ f\ e \rightsquigarrow\\ &\quad \mathsf{maxIR}\ (\lambda a\, (s, r).\, \mathbf{let}\ (s', a') = f\ s\ a\ \mathbf{in}\ (s', a' + r))\ (e, 0)\ (\leq_{\mathit{snd}} \cap =_{\mathit{fst}})\ (\mathsf{maxByAccumSum}'\ f\ e)\end{aligned} \tag{3}$$

Though it is a bit complicated, this rule exactly corresponds to Lemma 10. Notice that the first component of the pairs retains the value of $\mathit{foldr}\ f'\ e$, where $f'\ a\ s \stackrel{\text{def}}{=} \mathit{fst}\ (f\ s\ a)$; besides, the second component retains the value of $\mathit{sum} \circ \mathit{snd} \circ \mathit{mapAccumR}\ f\ e$. Thus, the maxIR in the right hand side first does a maximization by an order $\leq_{\mathit{snd} \circ \mathit{mapAccumR}\ f\ e} \cap =_{\mathit{foldr}\ f\ e}$, and does maxByAccumSum as its finalization. As proved Lemma 10, complete monotonicity certainly holds, which is the first requirement of maxIR. The second requirement also holds, as confirmed by the following calculation, in which $\preceq$ and $\sim$ respectively denote $\leq_{\mathit{snd} \circ \mathit{mapAccumR}\ f\ e}$ and $=_{\mathit{foldr}\ f\ e}$.

$$\begin{aligned}\mathsf{maximals}_{\ll} \circ \mathsf{maximals}_{\preceq} &= \{\ \mathsf{maximals}_{\sim} = \mathit{id}\ \}\\ &\quad\ \mathsf{maximals}_{\ll} \circ \mathsf{maximals}_{\preceq} \circ \mathsf{maximals}_{\sim}\\ &= \{\ \text{Lemma 11 (note that ; is associative)}\ \}\\ &\quad\ \mathsf{maximals}_{\ll} \circ \mathsf{maximals}_{\ll;\preceq} \circ \mathsf{maximals}_{\ll;\preceq;\sim}\\ &= \{\ \preceq;\sim\ =\ \preceq \cap \sim \text{ because } \sim \text{ is an equivalence relation}\ \}\\ &\quad\ \mathsf{maximals}_{\ll} \circ \mathsf{maximals}_{\ll;\preceq} \circ \mathsf{maximals}_{\ll;(\preceq\cap\sim)}\\ &= \{\ \text{Lemma 11}\ \}\\ &\quad\ \mathsf{maximals}_{\ll} \circ \mathsf{maximals}_{\preceq} \circ \mathsf{maximals}_{\ll;(\preceq\cap\sim)}\end{aligned}$$

For always, we can provide the following rule that exactly corresponds to Lemma 13, where note that *True* is larger than *False*.

$$\mathsf{always}\ p\ f\ e \rightsquigarrow \mathsf{constraint}\ p\ f\ e \circ \mathsf{maxIR}\ f\ e\ (\leq_p)\ \mathit{id} \tag{4}$$

That is, always is rewritten into a composition of maxIR and constraint. Again, the requirements of maxIR certainly hold.

Next, let us consider composition of the library functions. Since the rewrite rules we have introduced translate all of the library functions into compositions of maxIR and constraint, considering combinations of them is sufficient.

It is easy to provide a rule for combination of two constraint functions.

$$\begin{array}{l}\mathsf{constraint}\ p_1\ f_1\ e_1 \circ \mathsf{constraint}\ p_2\ f_2\ e_2 \rightsquigarrow \\ \quad \mathsf{constraint}\ (\lambda(r_1, r_2).\, p_1\ r_1 \wedge p_2\ r_2)\ (\lambda a\,(r_1, r_2).\, (f_1\ a\ r_1, f_2\ a\ r_2))\ (e_1, e_2)\end{array} \tag{5}$$

Nontrivial cases are combinations concerning maxIR. The following provides a rule for combination of maxIR and constraint.

$$\begin{array}{l}\mathsf{maxIR}\ f\ e\ (\preceq)\ k \circ \mathsf{constraint}\ p\ g\ z \rightsquigarrow \\ \quad \mathsf{maxIR}\ (\lambda a\,(r, s).\, (f\ a\ r, g\ a\ s))\ (e, z)\ (\preceq_{fst} \cap =_{snd})\ (k \circ \mathsf{maximals}_{\preceq_{foldr\ f\ e}} \circ \mathsf{constraint}\ p\ g\ z)\end{array} \tag{6}$$

This rule provides an implementation of Lemma 12. The left hand side is a composition of $\mathsf{maximals}_{\preceq_{foldr\ f\ e}}$ and constraint, and for pushing maximization before constraint, we introduce an order $\preceq_{foldr\ f\ e} \cap =_{foldr\ g\ z}$, which is used in the maxIR of the right hand side. It is worth noting that application of this rule does not break the requirements of maxIR. Complete monotonicity follows from Lemma 12. The second requirement can be confirmed as follows.

$$\begin{array}{ll} & \mathsf{maximals}_{\ll} \circ k \circ \mathsf{maximals}_{\preceq_{foldr\ f\ e}} \circ \mathsf{constraint}\ p\ g\ z \\ = & \quad \{\ \text{requirement of maxIR}\ \} \\ & \quad \mathsf{maximals}_{\ll} \circ k \circ \mathsf{maximals}_{\ll ; \preceq_{foldr\ f\ e}} \circ \mathsf{maximals}_{\preceq_{foldr\ f\ e}} \circ \mathsf{constraint}\ p\ g\ z \\ = & \quad \{\ \text{Lemma 11 (note that } \ll\ ;\preceq_{foldr\ f\ e}\ ;\preceq_{foldr\ f\ e} = \ll\ ;\preceq_{foldr\ f\ e})\ \} \\ & \quad \mathsf{maximals}_{\ll} \circ k \circ \mathsf{maximals}_{\ll ; \preceq_{foldr\ f\ e}} \circ \mathsf{constraint}\ p\ g\ z \\ = & \quad \{\ \text{Lemma 12 and the requirement of maxIR}\ \} \\ & \quad \mathsf{maximals}_{\ll} \circ k \circ \mathsf{maximals}_{\preceq_{foldr\ f\ e}} \circ \mathsf{constraint}\ p\ g\ z \circ \mathsf{maximals}_{\ll ; \preceq_{foldr\ f\ e} \cap =_{foldr\ g\ z}} \end{array}$$

Composition of two maxIRs can be fused into one by the following rule.

$$\begin{array}{l}\mathsf{maxIR}\ f_1\ e_1\ (\preceq)\ k_1 \circ \mathsf{maxIR}\ f_2\ e_2\ (\ll)\ k_2 \rightsquigarrow \\ \quad \mathsf{maxIR}\ (\lambda a\,(r_1, r_2).\, (f_1\ a\ r_1, f_2\ a\ r_2))\ (e_1, e_2)\ (\preceq_{fst}\ ;\ll_{snd})\ ()\ \ (k_1 \circ \mathsf{maximals}_{\preceq_{foldr\ f\ e}} \circ k_2)\end{array} \tag{7}$$

Let us see the intuition of this rule from the following calculation.

$$\begin{array}{ll} & \mathsf{maxIR}\ f_1\ e_1\ (\preceq)\ k_1 \circ \mathsf{maxIR}\ f_2\ e_2\ (\ll)\ k_2 \\ = & \quad \{\ \text{definition of maxIR}\ \} \\ & \quad k_1 \circ \mathsf{maximals}_{\preceq_{foldr\ f_1\ e_1}} \circ k_2 \circ \mathsf{maximals}_{\ll_{foldr\ f_2\ e_2}} \\ = & \quad \{\ \text{requirement of maxIR}\ \} \\ & \quad k_1 \circ \mathsf{maximals}_{\preceq_{foldr\ f_1\ e_1}} \circ k_2 \circ \mathsf{maximals}_{\preceq_{foldr\ f_1\ e_1} ; \ll_{foldr\ f_2\ e_2}} \circ \mathsf{maximals}_{\ll_{foldr\ f_2\ e_2}} \\ = & \quad \{\ \text{Lemma 11 (note that } \preceq_{foldr\ f_1\ e_1} ; \ll_{foldr\ f_2\ e_2} ; \ll_{foldr\ f_2\ e_2} = \preceq_{foldr\ f_1\ e_1} ; \ll_{foldr\ f_2\ e_2})\ \} \\ & \quad k_1 \circ \mathsf{maximals}_{\preceq_{foldr\ f_1\ e_1}} \circ k_2 \circ \mathsf{maximals}_{\preceq_{foldr\ f_1\ e_1} ; \ll_{foldr\ f_2\ e_2}} \end{array}$$

The rule (7) corresponds to the calculation above. In fact, the maxIR in the right hand side of the rule certainly performs maximization operation on the order $\preceq_{foldr\ f_1\ e_1} ; \ll_{foldr\ f_2\ e_2}$. Again, this rewriting rule does not break the requirements of maxIR: complete monotonicity follows from Lemma 11 and the second one can be confirmed as similar to the previous case.

So far, we have introduced rules to put the library functions together and assemble a single maxIR. Then, our shortcut fusion law will derive an efficient program. The following rule does the fusion.

$$\begin{array}{l}\mathsf{maxIR}\ f\ e\ (\preceq)\ k\ (\mathsf{gen}\ g) \rightsquigarrow k\ (map_{Set}\ fst\ (g\ \overline{\uparrow}_{\preceq_{snd}}\ \{\cdot\}\ \eta\ f'\ ([], e))) \\ \quad \mathsf{where}\ f'\ a\ (x, r) = (a : x, f\ a\ r) \\ \qquad\qquad \eta\ a = \mathsf{maximals}_{\preceq_{snd}} \circ map_{Set}\ (\lambda(x, r).\, (a : x, f\ a\ r))\end{array} \tag{8}$$

The correctness of this rule follows from Theorem 14, because the premise, complete monotonicity, holds from the requirement of maxIR.

5.3 Implementation Issues and Further Improvements

Our rewrite rules will derive algorithms that consider only maximals in each steps. Therefore, derived programs will invoke maximals a lot of times, and efficient implementation of maximals is crucial. However, only providing a comparison operator to maximals is not sufficient for efficiency, especially when a non-total order is given. For example, assume that we would like to obtain all maximals in a set of size n in which no two elements are comparable; then, to confirm that all elements in the set are maximals, it is necessary to do comparison operations n^2 times. This is not satisfactory.

In our library, because only a small numbers of functions are considered, we can supply more information than a comparison operation, and the additional information enables us to improve efficiency. To be concrete, we use the following data structure to represent orders.

$$\begin{array}{rcl} \textbf{data}\ \mathit{CMP}\ a & = & \forall b.\ \mathit{Ord}\ b \Rightarrow \mathit{Equiv}\ (a \to b) \\ & | & \forall b.\ \mathit{Ord}\ b \Rightarrow \mathit{TOrd}\ (a \to b) \\ & | & \mathit{Seq}\ (\mathit{CMP}\ a)\ (\mathit{CMP}\ a) \end{array}$$

The data structure $\mathit{CMP}\ a$ consists of three cases. $\mathit{Equiv}\ f$ represents an equivalence relation $=_f$, namely elements having the same f value is recognized to be equivalent. Note that the type requirement $\forall b.\ \mathit{Ord}\ b$ means that we would not know the exact range of f (namely b) beforehand, but we know that the range is a totally ordered set; in other words, the universal quantifier is interpreted as an existential quantifier from users. Similarly, $\mathit{TOrd}\ f$ represents a total order, on which two elements will be compared by their f values. $\mathit{Seq}\ (\preceq)\ (\ll)$ represents lexicographic composition of two orders.

Based on the structure CMP, we can provide an efficient implementation of maximals. The idea is similar to the one introduced by Henglein [Hen08] for multiset discrimination. We iteratively do sorting and partitioning according the structure of the ordering. Then, we can compute maximals for n valued set in $\mathrm{O}(n \log n)$ times of comparisons.

We implemented two additional improvements. One is compaction of equivalent elements. Recall that after the shortcut fusion law, namely the rewrite rule (8), each candidate consists of two values: one is the candidate itself, and the other is memoized value used for comparisons. We compress elements having the same memoized value into one. In our setting, no decompression is necessary because of complete monotonicity condition. Complete monotonicity indicates that extension of two equivalent elements results in being equivalent, i.e., $x \preceq y \wedge y \preceq x$ implies $(a : x) \preceq (a : y) \wedge (a : y) \preceq (a : x)$. Therefore, the compaction works very effectively. The other improvement is removal of surfeit of maximization operations. Since maximization operations will be the most costly part of generated programs, too many invocation of maximization will decrease efficiency. Therefore, we reduce the number of maximization. For example, $\mathsf{maximals}_{\preceq}\ (\mathsf{maximals}_{\preceq}\ X \cup \mathsf{maximals}_{\preceq}\ Y)$ is rewritten to $\mathsf{maximals}_{\preceq}\ (X \cup Y)$[4].

6 Examples and Experiments

To confirm effectiveness of our library, we did several experiments. The environment of the experiments is this: Intel Quad-Core Xeon 3.0GHz CPUs, 8 GB memory, Mac OS X, and Glasgow Haskell Compiler 6.10.1. Note that all of our programs use only one core. All computational times given in the following are averages of 100 executions, and exclude times for preparing inputs and outputting results.

6.1 Optimal Scheduling Problems

As the first example, let us solve the optimal scheduling problem that we have introduced in the introduction, and discuss strong points of our library. The problem is to find the best scheduling for rebooting of system whose performance will decrease by consecutive run.

First, how can we express a schedule as a list? Here we refer to the notion of *maximum marking problems* [SHTO00, Bir01]. Given a list representing the amount of data of each day, we generate a scheduling by adding a mark, either *Left* or *Right*, for each value. A value labeled *Left* stands for that the system runs in the day, and one labeled *Right* means that we reboot the system in the day. Actually our library consists of such generator *marking*.

$$\mathit{marking}\ [a_1, a_2, \ldots, a_n] \stackrel{\text{def}}{=} \{[a'_1, a'_2, \ldots, a'_n] \mid 1 \leq \forall i \leq n.\ (a'_i = \mathit{Left}\ a_i) \vee (a'_i = \mathit{Right}\ a_i)\}$$

[4] Correctness of such rules can be proved by Lemma 3.26 of [Mor09]

Of course, we have prepared the build form of *marking*, and thus, programs written by using *marking* will be improved by our rules.

6.1.1 Case 1: Performance Decreases in Rate

Next, we would like to specify the best schedule. Here let us consider the case that the performance of the system decreases 10 percent per day. Then, the performance of the next day can be specified by the following function *performance*, where *initSpeed* denotes the performance of the day just after rebooting.

$$\begin{array}{lcl} \mathit{performance}\ [\,] & = & \mathit{initSpeed} \\ \mathit{performance}\ (\mathit{Left}\ \_ : x) & = & \mathit{performance}\ x \times 9 \div 10 \\ \mathit{performance}\ (\mathit{Right}\ \_ : x) & = & \mathit{initSpeed} \end{array}$$

Performance returns to *initSpeed* if we did rebooting; otherwise, performance decreases 10 percent in comparison with the previous day. Then, it is easy to compute the amount of data processed by the schedule.

$$\begin{array}{lcl} \mathit{amount}\ [\,] & = & 0 \\ \mathit{amount}\ (\mathit{Left}\ a : x) & = & (a \downarrow \mathit{performance}\ x) + \mathit{amount}\ x \\ \mathit{amount}\ (\mathit{Right}\ \_ : x) & = & \mathit{amount}\ x \end{array}$$

Observe that *amount* can be specified by *sum* and *mapAccumR*; thus, the problem can be specified by our library as follows.

$$\begin{array}{l} \mathit{optScheduling1}\ x \stackrel{\text{def}}{=} \mathsf{maxByAccumSum}\ \mathit{step1}\ 0\ (\mathit{marking}\ x) \\ \quad \textbf{where} \quad \begin{array}{lcl} \mathit{step1}\ s\ (\mathit{Left}\ a) & = & (s \times 9 \div 10, a \downarrow s) \\ \mathit{step1}\ \_\ (\mathit{Right}\ \_) & = & (\mathit{initSpeed}, 0) \end{array} \end{array}$$

6.1.2 Case 2: Performance Decreases according to Amounts of Data

We have developed a program for a scheduling problem. Then, it is very easy to solve variants of the problem, because our library enables us to write programs similar to specifications of problems.

As the next example, let us consider the case where performance decreasing depends on the amount of data processed. To be concrete, we assume that the performance decreases by one-tenth of the amount of processed data of the day. Then, we can solve the problem by a program that is very similar to the previous one.

$$\begin{array}{l} \mathit{optScheduling2}\ x \stackrel{\text{def}}{=} \mathsf{maxByAccumSum}\ \mathit{step2}\ 0\ (\mathit{marking}\ x) \\ \quad \textbf{where} \quad \begin{array}{lcl} \mathit{step2}\ s\ (\mathit{Left}\ a) & = & (s - (a \downarrow s) \div 10, a \downarrow s) \\ \mathit{step2}\ \_\ (\mathit{Right}\ \_) & = & (\mathit{initSpeed}, 0) \end{array} \end{array}$$

The only difference is the difference between *step1* and *step2*, both of which take charge of computing the performance of each day.

6.1.3 Case 3: Weekly Rebooting is Required

As the third case, let us consider an additional requirement. Assume that for a safety reason, we should reboot the system once a week. Even for such a case, we can easily solve the problem by virtue of the specification-like program.

$$\begin{array}{l} \mathit{optScheduling3}\ x \stackrel{\text{def}}{=} \mathsf{maxByAccumSum}\ \mathit{step1}\ 0\ (\mathsf{always}\ (<7)\ \mathit{count}\ 0\ (\mathit{marking}\ x)) \\ \quad \textbf{where} \quad \begin{array}{lcl} \mathit{step1}\ s\ (\mathit{Left}\ a) & = & (s \times 9 \div 10, a \downarrow s) \\ \mathit{step1}\ \_\ (\mathit{Right}\ \_) & = & (\mathit{initSpeed}, 0) \\ \mathit{count}\ (\mathit{Left}\ \_)\ c & = & 1 + c \\ \mathit{count}\ (\mathit{Right}\ \_)\ \_ & = & 0 \end{array} \end{array}$$

Here we consider the case where performance decreases per day. We insert a function always that counts the number of consecutive runs and ensures the number should less than 7.

list length	1000	2000	3000	5000	10000	20000
optScheduling1	0.09	0.17	0.26	0.45	0.98	2.33
optScheduling2	0.10	0.21	0.32	0.55	1.17	2.80
optScheduling3	0.02	0.03	0.05	0.09	0.24	0.67

Table 1: Running times of programs for scheduling problems (unit: second).

list length	10000	20000	30000	50000	100000	200000
handwritten	0.02	0.04	0.07	0.14	0.37	1.12
our library	0.02	0.05	0.08	0.17	0.46	1.46

Table 2: Comparison of our library with a handwritten program for the maximum sum segment problem (unit: second).

6.1.4 Experiment

So far, we have developed three programs without caring efficiency, and we did experiments to check their efficiency. We generated various lengths of lists, whose each element is an integer uniformly chosen from 100 to 10000. We let *initSpeed* be 10000.

Table 1 shows the result of our experiments. First of all, all of three programs runs in time proportional to the length of lists, while number of schedules is exponential to the length. This is because our rewriting rules certainly derive efficient programs from the descriptions.

As seen, programs developed on our library are robust against changes of problem descriptions. We can easily modify programs according to the changes, and such modifications do not harm their efficiency in most of the cases. This is one of the strong points of our library.

6.2 Comparison with Handwritten Programs

In the next three examples, the maximum sum segment problem [Ben86, Bir89], the 0-1 knapsack problem [CSRL01], and the longest common subsequence problem [CSRL01], we compare programs based on our library with handwritten Haskell programs. Note that the handwritten ones are rather textbook programs than very finely tuned ones.

6.2.1 Maximum Sum Segment Problem

Let us consider the maximum sum segment problem, that is a problem to enumerate all segments of the maximum sum. It is trivial to specify this problem in our library.

$$mss'\ x = \mathsf{maxBySum}\ (segs\ x)$$

While it, it is not trivial to program its efficient solution by hand — in fact, we prepared over ten lines of Haskell codes for it.

Table 2 shows a comparison of two implementations. We generated several length of lists whose elements are uniformly chosen from -5000 to 10000. We can see that the program generated from our library runs with a little overhead in comparison with the handwritten one for this simple case.

6.2.2 0-1 Knapsack Problem

Next, let us consider the 0-1 knapsack problem. The problem is that given a set of items each of which is specified by its weight and value, we would like to find all its subsets of the maximum value among those whose weights are less than the given limit. Here we assume that weight of each item is a positive integer.

It is easy to specify the problem by using our library.

$$\begin{aligned} &knapsack\ x\ limit\ value\ weight = \\ &\quad \mathsf{maxBySumMap}\ value\ (\mathsf{always}\ (<limit))\ (\lambda a\, r.\, weight\ a + r)\ 0\ (subsequences\ x)) \end{aligned}$$

list length	100	200	300	500	1000	2000
handwritten	0.11	0.24	0.37	0.63	1.29	2.61
our library	0.17	0.37	0.58	0.99	2.03	4.16

Table 3: Comparison of our library with a handwritten program for the 0-1 knapsack problem (unit: second).

list length	100	200	300	500	1000	2000
handwritten	0.01	0.06	0.14	0.46	2.05	8.72
our library (without memoization of *step*)	0.01	0.08	0.24	0.89	4.70	24.23
our library (with memoization of *step*)	0.01	0.05	0.14	0.50	2.78	15.62

Table 4: Comparison of our library with a handwritten program for the longest common subsequence problem (unit: second).

The functions *value* and *weight* respectively specify the value and weight of an item, and *limit* is the limit of weight. In the program above, we first enumerate all subsequences (subsets) of items, and find ones having the maximum value among those whose weights are less than the limit.

Table 3 shows a comparison of two implementations. The generated lists that consisted of elements each of which had a value from -1000 to 10000 and a weight from 10 to 50. The limit was 1000. For this problem, programs based on our library is at most two times slower than the handwritten one, and the overhead would be acceptable.

6.2.3 Longest Common Subsequence Problem

As the final example, let us consider the longest common subsequence problem, which is the problem to compute the longest common subsequence of given two sequences. The longest common subsequence problem is an important problem because it appears practical situation such as finding modifications in sentences.

It is a bit difficult to specify the problem by using our library.

$$
\begin{aligned}
&\mathit{lcs}\ x\ y = \mathsf{maxByLexico}\ (\mathsf{maxByMapSum}\ (\lambda\_.\,1)\ (\mathsf{always}\ (\geq 0)\ \mathit{step}_y\ (\mathit{length}\ y)\ (\mathit{subsequences}\ x)))\\
&\quad \textbf{where}\ \mathit{step}_y\ a\ k = \textbf{if}\ k \leq 0\ \textbf{then}\ -1\\
&\qquad\qquad\qquad\qquad \textbf{else if}\ a = y!!(k-1)\ \textbf{then}\ k-1\ \textbf{else}\ \mathit{step}_y\ a\ (k-1)
\end{aligned}
$$

In the program above, we first generate all subsequences of x, and filter out ones that are not subsequences of y by always. In the always, the function step_y sweeps x and y from their tails to heads with confirming their correspondence. Then, we find the longest one. Since exponentially many of the longest common subsequences may exist, enumerating all of them is impractical; instead, we pick up one of them, and the function maxByLexico is useful for this purpose, because no two candidates are equivalent on lexicographic ordering. Note that we assume that the (!!) operation can be done in a constant time, and actually in our implementation, we covert y to an array beforehand.

In fact, the program above is not very efficient — it takes $\mathrm{O}(n^3)$ time for two lists of length n. The reason is the cost of *step*. *step* will be computed many times with the same argument. Therefore, memoization is effective. In this case, we can prepare the results *step a k* in advance, which will be finished in $\mathrm{O}(n^2)$ time. Then, we can obtain a program that finishes in $\mathrm{O}(n^2)$ time.

Table 4 shows a comparison of three implementations. In addition to a handwritten one, we prepared two programs based on our library: one is the program above, and the other is a program with memoization of *step*. We generated pairs of lists of the same length, and each elements of the lists are uniformly chosen from a set of alphabets of size 100.

We can see that programs based on our library is not very slow. The program without memoization would be still acceptable, though its asymptotic complexity is worse than others. Moreover, the program with memoization runs fast even for large inputs, and it is at most two times slower than the handwritten one.

7 Discussion

In this paper, we have proposed a theory and an implementation for developing efficient algorithms for finding optimal lists. We formalized derivation of efficient algorithms by shortcut fusion laws, pointed out that it is generally difficult to derive well-structured consumers, namely maximization computations by orders satisfying monotonicity condition, and introduced several laws that are useful for deriving well-structured consumers. Based on the theory developed, we introduced a Haskell library for enumerating optimal lists. The library functions are designed so that they will be not only useful for specifying requirements of optimal lists but also effective for deriving efficient algorithms. We implemented our program transformation rules by RULES pragma, and then, programs written by using our library are automatically improved. We confirmed effectiveness of our library with several examples. Our experiments showed that we can easily specify solutions of problems based on our library; besides, we can easily modify or refine programs on our library. Moreover, programs based on our library are efficient by virtue of embedded rewriting rules, and they are only about two times slower than handwritten codes in our experiments.

We formalized derivation of efficient algorithms by shortcut fusion [GLJ93, Gil96]. On one hand, fusion-based program developments have been studied, and among others, Bird and de Moor did an intensive study for deriving efficient algorithms of combinatorial optimization problems based on fusion transformations [dM92, BdM96], though it is not suitable for mechanization. On the other hand, shortcut fusion is one of the best-known fusion techniques, and known to be suitable for mechanization; however, there are few studies for algorithm development based on shortcut fusion. Our aim was to combine these two studies and provide an algorithm-development method that is suitable for mechanization. An extra profit from the use of shortcut fusion is that we can utilize existing methods for deriving build-forms or deriving programs written in some specific higher-order functions [LS95, Chi99, YHT05].

We have introduced laws for obtaining well-structured consumers, namely maximization computations by orders on which extending lists is completely monotonic. A key idea is to consider an intersection of a given order and an equivalent relation. Then, since the order specifies the maximum element for each equivalent class, derived algorithms will be *dynamic programming algorithms*. In dynamic programming methodology, we develop an efficient algorithm by considering the optimal solutions of subproblems and caching the optimal solutions so as to avoid recomputation. There has been a great deal of studies for systematically developing dynamic programming algorithms, such as an incrementalization-based method by Liu et al. [LS03, LSLR05], a selective-memoization-based method by Acar et al. [ABH03], a domain-specific-language-based method by Gergerich et al. [GMS04], and much more references are shown in these papers. On one hand, most of the existing studies assume that the structures of subproblems are explicitly specified by users. Actually, the structure of subproblems used in dynamic programming algorithms depends on details of problem descriptions, and therefore, explicit use of structures of subproblems would obstruct to modify or maintain programs. Our method derives the structure from programs, and this concealing improves robustness for modifications of problem descriptions. On the other hand, programs based on our method still requires effective memoization for efficient implementations. In summary, our method is complementary to them: ours gives a higher-level programming interface together with derivations of structures of subproblems, and others derive an efficient implementation from the structures of subproblems.

Our study is highly motivated by works for maximum marking problems [ALS91, BPT92, SHTO00, Bir01, SOH05]. Those studies report that for a class of combinatorial optimization problems called maximum marking problems, we can derive efficient algorithms from specifications of problems, once problems are described in specific forms. Our objective is to provide a clear and generic formalization for them and generalize them. Ours is certainly a generalization of them for problems to find optimal lists. They discussed problems concerning more generic structures, such as trees, rather than lists. Actually, it is not difficult to extend our results so as to deal with problems to find optimal (possibly non-lists) structures, because there is a generalization of shortcut fusion, called acid rain theorem [TM95], that can eliminate even non-list intermediate structures.

Programs based on our library are not very efficient yet, and there would be a room for further improvements. Recently, Puchinger and Stuckey [PS08] introduced a method that automatically derives efficient branch-and-bound procedure from dynamic programming algorithms. We feel that combination of our method and theirs is effective. Our method automatically derives dynamic programming algorithms by gathering some information from programs, and we can supply such information to further

improvements.

Acknowledgement

This paper is submitted to METR on communication with Professor Masato Takeichi. The author is supported by the Grant-in-Aid for JSPS research fellows $20 \cdot 2411$.

References

[ABH03] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, January 15-17, 2003*, pages 14–25, 2003.

[ALS91] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991.

[BdM96] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1996.

[Ben86] Jon Bentley. *Programming Pearls*. ACM, New York, NY, USA, 1986.

[Bir89] Richard S. Bird. Algebraic identities for program calculation. *Computer Journal*, 32(2):122–126, 1989.

[Bir01] Richard S. Bird. Maximum marking problems. *Journal of Functional Programming*, 11(4):411–424, 2001.

[BPT92] Richard B. Borie, R. Gary Parker, and Craig A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7(5&6):555–581, 1992.

[Chi99] Olaf Chitil. Type inference builds a short cut to deforestation. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming, ICFP'99, Paris, France*, volume 34(9), pages 249–260. ACM, New York, 1999.

[CKJ06] Wei-Ngan Chin, Siau-Cheng Khoo, and Neil Jones. Redundant call elimination via tupling. *Fundamenta Informaticae*, 69(1-2):1–37, 2006.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to algorithms, Second edition*. MIT Press, Cambridge, MA, USA, 2001.

[dM92] Oege de Moor. *Categories, Relations and Dynamic Programming*. PhD thesis, Oxford University Computing Laboratory, 1992. *Technical Monograph PRG*-98.

[Gil96] Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Department of Computing Science, Glasgow University, 1996.

[GLJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture. Copenhagen, Denmark, 9-11 June 1993*, pages 223–232. ACM, 1993.

[GMS04] Robert Giegerich, Carsten Meyer, and Peter Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.

[Hen08] Fritz Henglein. Generic discrimination: sorting and paritioning unshared data in linear time. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 91–102. ACM, 2008.

[HHJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[HITT97] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming, ICFP'97, Amsterdam, The Netherlands, June 9-11, 1997*, pages 164–175. ACM, 1997.

[KGK01] Armin Kühnemann, Robert Glück, and Kazuhiko Kakehi. Relating accumulative and non-accumulative functional programs. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications ,RTA'01*, pages 154–168. Springer-Verlag, 2001.

[KT05] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Conference Record of FPCA'95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture. La Jolla, CA, USA, 25-28 June 1995*, pages 314–323. ACM Press, New York, 1995.

[LS03] Yanhong A. Liu and Scott D. Stoller. Dynamic programming via static incrementalization. *Higher-Order and Symbolic Computation*, 16(1–2):37–62, 2003.

[LSLR05] Yanhong A. Liu, Scott D. Stoller, Ning Li, and Tom Rothamel. Optimizing aggregate array computations in loops. *ACM Transactions on Programming Languages and Systems*, 27(1):91–125, 2005.

[Mor09] Akimasa Morihata. *Calculational Approach to Automatic Algorithm Construction*. PhD thesis, Department of Mathematical Informatics, University of Tokyo, 2009.

[Mu08] Shin-Cheng Mu. Maximum segment sum is back: deriving algorithms for two segment problems with bounded lengths. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*, pages 31–39. ACM, 2008.

[Pey03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[PS08] Jakob Puchinger and Peter J. Stuckey. Automating branch-and-bound for dynamic programs. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*, pages 81–89. ACM, 2008.

[PTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Proceedings of 2001 ACM SIGPLAN Haskell Workshop (HW'2001), Firenze, Italy, 2nd September 2001. Technical Report UU-CS-2001-23*, pages 203–233. Institute of Information and Computing Sciences Utrecht University, 2001.

[SHTO00] Isao Sasano, Zhenjiang Hu, Masato Takeichi, and Mizuhito Ogawa. Make it practical: a generic linear-time algorithm for solving maximum-weightsum problems. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 137–149. ACM, 2000.

[SOH05] Isao Sasano, Mizuhito Ogawa, and Zhenjiang Hu. Maximum marking problems with accumulative weight functions. In *Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Hanoi, Vietnam, October 17-21, 2005, Proceedings*, pages 562–578. Springer, 2005.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conference Record of FPCA '95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture. La Jolla, CA, USA, 25-28 June 1995*, pages 306–313. ACM Press, New York, 1995.

[Voi02] Janis Voigtländer. Concatenate, reverse and map vanish for free. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 14–25, 2002.

[Wad89] Philip Wadler. Theorems for free! In *FPCA '89 Conference on Functional Programming Languages and Computer Architecture. Imperial College, London, England, 11-13 September 1989*, pages 347–359. ACM, New York, 1989.

[YHT05] Tetsuo Yokoyama, Zhenjiang Hu, and Masato Takeichi. Calculation rules for warming-up in fusion transformation. In *the 2005 Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*, pages 399–412, 2005.