Generation of Efficient Programs for Solving Maximum Multi-Marking Problems

Isao Sasano*, Zhenjiang Hu, and Masato Takeichi

Department of Information Engineering, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, JAPAN
{sasano,hu,takeichi}@ipl.t.u-tokyo.ac.jp
http://www.ipl.t.u-tokyo.ac.jp/~{sasano,hu,takeichi}

Abstract. Program generation has seen an important role in a wide range of software development processes, where effective calculation rules are critical. In this paper, we propose a more general calculation rule for generation of efficient programs for solving maximum marking problems. Easy to use and implement, our new rule gives a significant extension of the rule proposed by Sasano *et al.*, allowing multiple kinds of marks as well as more general description of the property of acceptable markings. We illustrate its effectiveness using several interesting problems.

Keywords: Program Generation Rule, Optimization Problem, Maximum Marking Problem, Functional Programming, Algorithm Synthesis.

1 Introduction

Program generation has seen an important role in a wide range of software development processes. A successful program generation system requires not only a powerful language supporting coding of program generation, but also a set of effective transformation rules for the generation of programs. An example, which convincingly shows the importance of the design of effective transformation rules, is the well-known *fold-build* rule [1] for fusing composition of functions in Glasgow Haskell Compiler (GHC). It is this general, concise and cheap calculation rule that makes it possible for GHC to *practically* generate from *large-scale* programs efficient programs without unnecessary intermediate data structures. Generally, the effective rules for program generation should meet several requirements.

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

* Isao Sasano is supported by JSPS Research Fellowships for Young Scientists.

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

$$mmm\ p\ wf\ k \ = \ \uparrow_{wf}/ \ \circ \ \mathit{filter}\ p \ \circ \ \mathit{gen}\ k,$$

with which one can straightforwardly specify solutions for the *maximum marking problems* [2] (sometimes also called *maximum weightsum problems* [3]): from the elements of a data structure, find a subset which satisfies a certain property p and whose weightsum is maximum. Informally speaking, this program pattern generates all the possible ways of marking the elements in the input data using the generation function *gen*, keeps those markings satisfying the property p, and finally selects one which has the maximum value with respect to the weight function *wf*. More formal explanation can be found in Section 3.

The maximum marking problems are interesting because they encompass a very large class of optimization problems [4, 5], and they have attracted many researchers. Based on the algebraic laws of programs [6, 7], Bird successfully derived a linear algorithm to solve the maximum segment sum problem [6], which is a maximum weightsum problem on lists. Bird *et al.* [7] demonstrated the derivation of many kinds of maximum marking problems. However, the success of derivation not only depends on a powerful calculation theorem but also involves careful and insightful justification to meet the conditions of the theorem, which makes it difficult for the theorem to be used for mechanical program generation. On the other hand, it has been shown for decades [4, 5], that if specified by so-called *regular predicates* [5], the maximum marking problems are linear time solvable and such linear time programs can be automatically generated. Though being systematic and constructive, the generated linear time programs suffer from an impractically large table [5], which actually prevents them from practical use. To resolve this problem, Sasano *et al.* gave a new approach [3] to generating practical linear time algorithms for the maximum marking problems over data structures such as lists, trees, and decomposable graphs. The key point there is to express the property p by a *recursive* function of certain form, and to apply program transformation techniques for program optimization.

However, there still remain several limitations. First, the number of kinds of marks are basically restricted to two. Although by using two marks (marking an element or not) one can describe many combinatorial optimization problems such as the 0-1 knapsack problem, it is difficult to handle the optimization problems where more states on elements are required, as will be seen later. Second, the property p is restrictive in the sense it must be a function without the use of accumulating parameters, which makes it hard to specify history-sensitive properties. Third, the weight function *wf* is restricted to only the sum of marked elements.

To remedy this situation, we extend the work in Sasano *et al.* [3], giving a calculation rule for generating efficient programs for solving *maximum multi-marking problems*. Our main contributions can be summarized as follows.

– We propose a new calculation rule (Section 4) for generating efficient algorithms for solving maximum multi-marking problems. It can efficiently

handle multi-marking, general weight functions, and property description with an accumulating parameter, leading to a more *general* framework for solving a wider class of combinatorial optimization problems, covering those in the previous work [2–5].
- We demonstrate, with several non-trivial example problems, that our calculation rule provides a practical and friendly interface for people both to specify those problems and to generate efficient programs automatically. Surprisingly, as partly shown in Section 5, our approach can deal with many optimization problems in Bird *et al.* [7]. By contrast, the derivation process in Bird *et al.* [7] is difficult to be mechanized.
- We show that our calculation rule can be easily implemented by using the existing transformation systems like MAG [8], and efficient programs can be obtained in a fully automatic way.

The organization of this paper is as follows. In Section 2, we briefly review the previous work on the maximum marking problems and explain limitations. In Section 3, we give a formal definition of the maximum multi-marking problems. In Section 4, we propose our optimization theorem which gives the rule for generation of efficient programs for maximum multi-marking problems. In Section 5, we show the effectiveness of our approach by deriving efficient algorithms for solving several interesting problems. Related work is discussed in Section 6, and the conclusion is made in Section 7.

2 Maximum Marking Problems

In this section, we briefly review the previous work on maximum marking problems and the results obtained in Sasano *et al.* [3]. The maximum marking problems are a special and simpler case of the maximum multi-marking problems.

We assume that the readers are familiar with the functional language Haskell [9], whose notation will be used throughout this paper.

2.1 Overview

Given a data structure xs (of type $D\ \alpha$), a *maximum marking problem* is to find a marking of xs's elements such that the marked data structure xs^* (of type $D\ \alpha^*$) satisfies a certain property p, and that the sum of the marked elements in xs^* is maximum. Here the "maximum" means that no other marking of xs satisfying p can produce a larger weightsum.

As an example, consider the maximum independent sublist sum problem (*mis* for short) [3], which is to compute a way of marking of the elements in a list xs, such that no two marked elements are adjacent and the sum of the marked elements are maximum. For instance, for

$$xs = [1, 2, 3, 4, 5]$$

the result is the marking of

$$[1^*, 2, 3^*, 4, 5^*],$$

which gives the maximum sum of 9 among all the feasible marking of xs. One can check that any other way of feasible marking cannot give a larger sum.

A straightforward solution for the general maximum marking problem, whose complexity is exponential to the number of elements in xs, can be defined precisely as follows:

$$
\begin{array}{lcl}
mws & : & (D\ \alpha^* \rightarrow Bool) \rightarrow D\ \alpha \rightarrow D\ \alpha^* \\
mws\ p & = & \uparrow_{wsum} / \ \circ\ filter\ p\ \circ\ gen.
\end{array}
$$

The function *gen* generates all the possible markings of input data, and from those which satisfy the property p the function $\uparrow_{wsum}$ / selects one whose weightsum of marked elements is maximum. The operator / is called *reduce* [10] and is defined as follows:

$$\oplus/[x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

where $\oplus$ is an associative operator. The operator $\uparrow_f$ is called *selection* [10] and is defined as follows:

$$
\begin{array}{rll}
x \uparrow_f y = & x, & \textbf{if } f\ x \geq f\ y \\
= & y, & \textbf{otherwise}.
\end{array}
$$

Using mws, one can specify many interesting optimization problems by giving different definition for the property p [3]. For example, the property p for the above *mis* problem can be naturally defined as follows:

$$
\begin{array}{ll}
p\ [\,] & = True \\
p\ (x : xs) & = \textbf{if } marked\ x \textbf{ then } p_1\ xs \textbf{ else } p\ xs \\
 & \\
p_1\ [\,] & = True \\
p_1\ (x : xs) & = not\ (marked\ x)\ \wedge\ p\ xs.
\end{array}
$$

Here, the function *marked* takes as its argument an element x and returns *True* when x is marked and returns *False* otherwise. The calculation rule already proposed in Sasano *et al.* [3] says that if the property is described in a *mutumorphic* form (which can be considered as a mutually recursive version of *fold*), then a linear program can be automatically generated. So for the *mis* problem, since the property p is already described in a mutumorphic form, we can conclude that a linear program can be obtained.

2.2 Limitations

To see the limitations of the existing approach, consider the following coloring problem, a simple extension of the *mis* problem. Suppose there are three marks: red, blue, and yellow. The problem is to find a way of marking all the elements such that each sort of mark does not appear continuously, and that the sum of the elements marked in red minus the sum of the elements marked in blue is maximum. To obtain an efficient algorithm for this problem by using the existing approach [3], we intend to specify this coloring problem using the following program pattern:

$$mws\ p \;=\; \uparrow_{wsum} / \;\circ\; filter\ p \;\circ\; gen.$$

Unfortunately, there are several problems preventing us from doing so. First, the existing generation function *gen* generates all the possible markings with just a single kind of mark, but a single kind of mark is not enough for this problem. That means we need to extend the generation function *gen* so that it can generate all the possible markings with multiple kinds of marks. A generation function for the coloring problem may be written as follows:

$$\begin{array}{l}
gen\ [\,] \qquad\quad = [[\,]] \\
gen\ (x : xs) = [(x, m) : ys \mid m \leftarrow [Red, Blue, Yellow], ys \leftarrow gen\ xs].
\end{array}$$

Second, the property description p for the coloring problem can be naturally specified as follows:

$$\begin{array}{ll}
indep\ xs & = indep'\ xs\ Neutral \\
indep'\ [\,]\ color & = True \\
indep'\ (x : xs)\ color & = markKind\ x \neq color \wedge indep'\ xs\ (markKind\ x).
\end{array}$$

But this is not in a required mutumorphic form such that the rule in Sasano *et al.* [3] can be applied, because *indep*$'$ has an additional accumulating parameter *color*. Here, *Neutral* is used as the initial value of the accumulating parameter, which is different from all the colors used for coloring the elements. The function *markKind* takes as its argument a marked element and returns the kind of mark of the element. If we insist on specifying *indep* in a mutumorphic form, we would have to instantiate all the possible values of *color* used by *indep*$'$, and could reach the following complicated definition:

$$\begin{array}{ll}
indep\ [\,] & = True \\
indep\ (x : xs) & = \mathbf{case}\ markKind\ x\ \mathbf{of} \\
& \quad Red \rightarrow indep_R\ xs \\
& \quad Blue \rightarrow indep_B\ xs \\
& \quad Yellow \rightarrow indep_Y\ xs
\end{array}$$

$$\begin{array}{ll}
indep_R\ [\,] & = True \\
indep_R\ (x : xs) & = \mathbf{case}\ markKind\ x\ \mathbf{of} \\
& \quad Red \rightarrow False \\
& \quad Blue \rightarrow indep_B\ xs \\
& \quad Yellow \rightarrow indep_Y\ xs
\end{array}$$

$$\begin{array}{ll}
indep_B\ [\,] & = True \\
indep_B\ (x : xs) & = \mathbf{case}\ markKind\ x\ \mathbf{of} \\
& \quad Red \rightarrow indep_R\ xs \\
& \quad Blue \rightarrow False \\
& \quad Yellow \rightarrow indep_Y\ xs
\end{array}$$

$$
\begin{aligned}
indep_Y\ [\,] &= True\\
indep_Y\ (x:xs) &= \mathbf{case}\ markKind\ x\ \mathbf{of}\\
&\qquad Red \to indep_R\ xs\\
&\qquad Blue \to indep_B\ xs\\
&\qquad Yellow \to False.
\end{aligned}
$$

In fact, this instantiation not only leads to a complicated definition, but also makes the generated program less efficient than that generated in Section 4.

Finally, the weight function *wsum* fixed in the program pattern *mws p* is rather restrictive. For the coloring problem, we may hope to use the following weight function:

$$
\begin{aligned}
wf &= +/ \circ map\ f\\
&\quad \mathbf{where}\ f\ x = \mathbf{case}\ markKind\ x\ \mathbf{of}\\
&\qquad\qquad Red \to weight\ x\\
&\qquad\qquad Blue \to -(weight\ x)\\
&\qquad\qquad Yellow \to 0,
\end{aligned}
$$

which is clearly not a simple *wsum*.

To overcome these limitations, in this paper we will give a more general program pattern for specifying the maximum marking problems with multiple marks, while guaranteeing that an efficient program can be automatically generated from this program pattern.

3 Maximum Multi-Marking Problems

In this section, we give a formal definition of the maximum multi-marking problems. To simplify our presentation, we focus on the problems on lists in this paper.

A maximum multi-marking problem can be specified as follows. Given a list xs, the task is to find a way to mark each element in xs such that the marked data structure xs, say xs^*, satisfies a certain property p, and the value of weight function *wf* of marked list xs^* is maximum. A straightforward program *mmm* to solve this problem is

$$
\begin{aligned}
mmm\quad &: ([\alpha^*] \to Bool) \to ([\alpha^*] \to Weight) \to Int \to [\alpha] \to [\alpha^*]\\
mmm\ p\ wf\ k &= \uparrow_{wf} / \circ filter\ p \circ gen\ k.
\end{aligned}
$$

We use *gen* k, which is different from that in Section 2, to generate all the possible markings of an input list with k kinds of marks, and from those which satisfy the property p we use $\uparrow_{wf} /$ to select one with maximum value of the weight function *wf*. Many optimization problems, including the coloring problem in Section 2, can be expressed using *mmm* by giving a suitable property p and weight function *wf*.

Before defining *gen* and *wf*, we explain some of our notation for marking. For a data type of α, we use α^* to extend α with marking information. It can be defined more precisely as follows:

$$\alpha^* = (\alpha, Mark)$$

where *Mark* is the type of marks. We use integers from 1 to k as marks where k is the integer which is given as the third argument of mmm. Of course it is possible to use any set holding k elements for *Mark*. Accordingly, we use a^*, $b^*, \ldots, x^*$ to denote variables of the type α^* and use $xs^*, ys^*, \ldots$ to denote variables of the type $[\alpha^*]$.

The function *gen*, exhaustively enumerating all the possible ways of marking elements using marks from 1 to k, can be recursively defined as follows:

$$\begin{array}{ll} \mathit{gen} & : \ \mathit{Int} \to [\alpha] \to [\alpha^*] \\ \mathit{gen}\ k\ [\,] & = [[\,]] \\ \mathit{gen}\ k\ (x:xs) & = [\,x^* : xs^* \mid x^* \leftarrow \mathit{mark}\ x, xs^* \leftarrow \mathit{gen}\ k\ xs\,] \end{array}$$

where *mark* is a function for marking which is defined as follows:

$$\mathit{mark}\ x = [\,(x,m) \mid m \leftarrow [1\ ..\ k]\,].$$

In addition, in order to get the mark from a marked element, we define the function *markKind* as follows:

$$\mathit{markKind}\ (x,m) = m.$$

Using mmm we can express various kinds of problems. For example, the coloring problem in Section 2 can be specified as follows:

$$\begin{array}{l} \mathit{coloring} \ = \ \mathit{mmm}\ \mathit{indep}\ \mathit{wf}\ 3 \\ \\ \mathit{indep}\ xs \ = \ \mathit{indep}'\ xs\ 0 \\ \mathit{indep}'\ [\,]\ \mathit{color} \ = \ \mathit{True} \\ \mathit{indep}'\ (x:xs)\ \mathit{color} \ = \ \mathit{markKind}\ x \neq \mathit{color} \wedge \mathit{indep}'\ xs\ (\mathit{markKind}\ x) \\ \\ \mathit{wf} \ = \ +/ \circ \mathit{map}\ f \\ \qquad \textbf{where}\ f\ e^* \ = \ \textbf{case}\ \mathit{markKind}\ e^*\ \textbf{of} \\ \qquad\qquad\qquad\qquad 1 \to \mathit{weight}\ e^* \\ \qquad\qquad\qquad\qquad 2 \to -(\mathit{weight}\ e^*) \\ \qquad\qquad\qquad\qquad 3 \to 0. \end{array}$$

Of course, this definition of mmm is terribly inefficient, though it is straightforward. In the next section, we would like to show that they can be automatically transformed to an efficient linear one.

4 Program Generation

In this section, we propose a theorem for generating efficient algorithms for solving maximum multi-marking problems, and highlight how this theorem can be easily implemented for automatic generation of efficient executable programs.

4.1 Generation Rule

We shall propose a theorem for generating efficient algorithms for solving maximum multi-marking problems. The theorem gives a significant extension of the theorem proposed in Sasano *et al.* [3]. It can efficiently handle multi-marking, general weight functions, and property description with an accumulating parameter. Before giving the theorem, we should be more precise about the requirement of the weight function *wf* and the property p used for specifying a maximum multi-marking problem *mmm* in Section 3.

The previous work, including ours, restricted the weight function *wf* to be just the sum of weight of marked elements [2–5]. As seen in the coloring problem in Section 2, we often need to use a more general weight function. For this purpose, we define the following general form, a kind of list homomorphism [10], which we call *homomorphic weight function*.

Definition 1 (Homomorphic Weight Function). *A function wf is a homomorphic weight function if it is defined as follows:*

$$\begin{array}{rcl} wf & : & [\alpha^*] \to \mathit{Weight} \\ wf & = & \oplus/ \circ \ \mathit{map}\ f \end{array}$$

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

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

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

For the property p which is to specify the feasible markings with multiple kinds of marks, the existing approach [3] (as seen in the definition of *indep* in Section 2) only allows p to be defined in a *mutumorphic* form with several other functions, say $p_1, \ldots, p_n$, whose ranges are finite.

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

If p is defined in the mutumorphic form, by applying the tupling transformation [11, 12], we can always come up with the following definition for p, a composition of a project function with a *foldr*:

$$\begin{array}{l} p = \mathit{fst}\ \circ\ \mathit{foldr}\ \psi\ e' \\ \quad\ \textbf{where}\ \psi\ x\ es = (\phi\ x\ es, \phi_1\ x\ es, \ldots, \phi_n\ x\ es) \\ \qquad\qquad e' = (e, e_1, \ldots, e_n). \end{array}$$

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

$$\begin{array}{l} foldr_h\ (\phi_1, \phi_2)\ \delta\ []\ e \qquad\quad = \phi_1\ e \\ foldr_h\ (\phi_1, \phi_2)\ \delta\ (x : xs)\ e = \phi_2\ x\ e\ (foldr_h\ (\phi_1, \phi_2)\ \delta\ xs\ (\delta\ x\ e)). \end{array}$$

Using this function $foldr_h$, we define the following form, which we call *finite accumulative property.*

Definition 2 (Finite Accumulative Property). *A property p is a finite accumulative property if it is defined as follows:*

$$\begin{array}{l} p : [\alpha^*] \to Bool \\ p\ xs = g\ (foldr_h\ (\phi_1, \phi_2)\ \delta\ xs\ e_0) \end{array}$$

where the domain of g and range of δ is finite.

Now we propose our main theorem.

Theorem 1 (Generation Rule). *Suppose a specification of a maximum multi-marking problem is given as*

$$mmm\ p\ wf\ k \ = \ \uparrow_{wf} / \ \circ\ filter\ p\ \circ\ gen\ k.$$

If wf is a homomorphic weight function

$$wf \ = \ \oplus / \ \circ\ map\ f$$

and p is a finite accumulative property

$$p\ xs \ = \ g\ (foldr_h\ (\phi_1, \phi_2)\ \delta\ xs\ e_0),$$

then the maximum multi-marking problem ($mmm\ p\ wf\ k$) can be solved by

$$opt\ k\ (\lambda(c, e)\ .\ g\ c\ \wedge\ e == e_0)\ (f, \oplus, \iota_\oplus)\ \phi_1\ \phi_2\ \delta.$$

The definition of opt is given in Figure 1.

This theorem has a form similar to that in Sasano *et al.* [3] except for using array in the definition of *opt* for efficiency, and it can be proved by induction on the input list. We omit the detailed proof in this paper, due to the space limitation. One remark worth making is about the cost of the derived program. Assuming that δ and g have the types

$$\begin{array}{l} \delta : \alpha^* \to Acc \to Acc \\ g : Class \to Bool, \end{array}$$

we can conclude that the generated program using *opt* can be computed in $O(|Acc| \cdot |Class| \cdot k \cdot n)$ time, where n is the length of input list, k is the number of marks, and $|Acc|$ and $|Class|$ denote the size of the type Acc and the type $Class$ respectively. That means that our approach is applicable only when the domain of g and the range of δ is finite. If our approach is applicable, our generated program is much more efficient than the initial specification program $mmm\ p\ wf\ k$, which is exponential.

$$
\begin{array}{l}
opt\ k\ accept\ (f, \oplus, \iota_{\oplus})\ \phi_1\ \phi_2\ \delta\ xs = \\
\quad \textbf{let}\ opts = foldr\ \psi_2\ \psi_1\ xs \\
\quad \textbf{in}\ snd\ (\uparrow_{fst}\ /\ [\,(w, r^*) \mid Just\ (w, r^*) \leftarrow [\,opts!i \mid i \leftarrow range\ bnds, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad opts!i \neq Nothing, accept\ i\,]\,]) \\
\quad \textbf{where}\ \psi_1 = array\ bnds\ [\,(i, g\ i) \mid i \leftarrow range\ bnds] \\
\qquad\qquad \psi_2\ x\ cand = accumArray\ h\ Nothing\ bnds \\
\qquad\qquad\qquad\qquad [\,((\phi_2\ x^*\ e\ c,\ e),\ (f\ x^* \oplus w,\ x^* : r^*)) \\
\qquad\qquad\qquad\qquad \mid x^* \leftarrow [\,(x,1), (x,2), \ldots, (x,k)\,], \\
\qquad\qquad\qquad\qquad\ \ e \leftarrow acclist, \\
\qquad\qquad\qquad\qquad\ \ ((c, \_), Just\ (w, r^*)) \leftarrow \\
\qquad\qquad\qquad\qquad\qquad [\,(i, cand!i) \mid i \leftarrow [\,(c', \delta\ x^*\ e) \mid c' \leftarrow classlist], \\
\qquad\qquad\qquad\qquad\qquad\qquad inRange\ bnds\ i, \\
\qquad\qquad\qquad\qquad\qquad\qquad cand!i \neq Nothing\,]\,] \\
\qquad\qquad g\ (c, e) = \textbf{if}\ (c == \phi_1\ e)\ \textbf{then}\ Just\ (\iota_{\oplus}, [])\ \textbf{else}\ Nothing \\
\qquad\qquad h\ (Just\ (w_1, x_1))\ (w_2, x_2) = \textbf{if}\ w_1 > w_2\ \textbf{then}\ Just\ (w_1, x_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ Just\ (w_2, x_2) \\
\qquad\qquad h\ Nothing\ (w, x) = Just\ (w, x) \\
\qquad\qquad bnds = ((head\ classlist, head\ acclist), (last\ classlist, last\ acclist)) \\
acclist = \text{list of all the values in } Acc \\
classlist = \text{list of all the values in } Class
\end{array}
$$

Fig. 1. Optimization function *opt*.

An Example To see how the theorem works, we demonstrate how to derive a linear algorithm for the coloring problem in Section 2. Recall that the specification for the coloring problem has been given in Section 3. The weight function has been written in our required form, and the property *indep* can be easily rewritten using $foldr_h$ as follows:

$$
\begin{array}{l}
indep\ xs = id\ (foldr_h\ (\phi_1, \phi_2)\ \delta\ xs\ 0) \\
\qquad\qquad \textbf{where}\ \phi_1\ e = True \\
\qquad\qquad\qquad\quad \phi_2\ x\ e\ r = markKind\ x \neq e \wedge r \\
\qquad\qquad\qquad\quad \delta\ x\ e = markKind\ x.
\end{array}
$$

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

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

gives the result of

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

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

4.2 Implementation

Our generation rule can be implemented, so that efficient programs can be generated automatically. In this section, we highlight[1] how we can do so using MAG system [8], a transformation system with a powerful *higher order pattern matching*.

As seen in Figure 2, our obtained program can be divided into two parts: the dynamic and static parts. The dynamic part changes from problems to problems, while the static part remains the same. In Figure 2, the upper part is dynamic and the lower is static. We show how to generate the dynamic part from the specification *mmm p wf k*.

Using MAG, we may code the generation of the dynamic part from specification *mmm p wf k* as a rule called `mmmRule` as follows.

```
mmmRule: mmm p wf k
           = opt k accept (f,oplus,e) phi1 phi2 delta,
        if {
             wf = foldr (oplus) e . map f;
             p xs = g (h xs e0);
             h [] = phi1;
             h (x:xs) y = phi2 x y (h xs (delta x y));
             \(c,e) -> g c && e==e0 = accept
           };
```

Now for the coloring problem, we can apply this rule to the following specification and obtain a linear time program as in Figure 2 automatically.

```
coloring: coloring = mmm indep wf 3;
wf: wf = foldr (+) 0 . map f;
f: f = \x -> case markKind x of
                1 -> weight x
                2 -> - (weight x)
                3 -> 0;
p: p xs = p' xs 0;
p1: p' [] color = True;
p2: p' (x:xs) color = markKind x /= color && p' xs (markKind x);
classlist: classlist = [False,True];
acclist: acclist = [0..3]
```

[1] Although actually we cannot do it because of several restrictions of the MAG system, we are only showing the flavor of the implementation.

```
coloring = opt 3 accept (f, (+), 0) phi1 phi2 delta
acclist = [0..3]
classlist = [False, True]
accept (c,e) = c && e==0
f = \x -> case markKind x of
              1 -> weight x
              2 -> - (weight x)
              3 -> 0
phi1 e = True
phi2 x e c = markKind x /= e && c
delta x e = markKind x
markKind (_,m) = m
weight (x,_) = x
-------------------------------------------------------------------
opt k accept (f, oplus, id_oplus) phi1 phi2 delta xs =
  let opts = foldr psi2 psi1 xs
  in snd (getmax [(w,r) | Just (w,r) <- [ opts!i
                                        | i <- range bnds,
                                          opts!i /= Nothing,
                                          accept i]])
  where psi1 = array bnds [(i, g i) | i <- range bnds]
        psi2 x cand = accumArray h Nothing bnds
                        [((phi2 xm e c, e),
                          (f xm `oplus` w, xm:r))
                        | xm <- [(x,m) | m <- [1..k]],
                          e <- acclist,
                          ((c,_),Just (w,r)) <-
                             [ (i,cand!i)
                             | i <- [ (c',delta xm e)
                                    | c' <- classlist],
                               inRange bnds i,
                               cand!i /= Nothing]]
        g (c,e) = if (c == phi1 e) then Just (id_oplus, [])
                  else Nothing
        h (Just (w1,x1)) (w2,x2) = if w1 > w2 then Just (w1,x1)
                                   else Just (w2,x2)
        h Nothing (w,x) = Just (w,x)
        bnds = ((head classlist,head acclist),
                (last classlist,last acclist))
getmax [] = error "No solution."
getmax xs = foldr1 f xs
            where f (w1,cand1) (w2,cand2)
                    = if w1>w2 then (w1,cand1) else (w2,cand2)
```

Fig. 2. A linear-time Haskell program for the coloring problem.

With these, the MAG system can produce the linear time program as given in Figure 2. Note that the current version of MAG system has several restrictions such as not allowing `case` expression, so MAG system needs extension for our purpose.

5 More Examples

In this section, we give more examples, showing that our proposed gerneration rule is quite general and powerful.

5.1 Paragraph Formatting Problem

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

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

$$mmm\ p\ wf\ k.$$

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

$$
\begin{array}{l}
p\ xs = p'\ xs\ (0,2) \\
p'\ [\,]\ (pos, mk) = mk \neq 1 \\
p'\ (x : xs)\ (pos, mk) = \\
\quad \textbf{case}\ mk\ \textbf{of} \\
\quad\quad 1 \rightarrow \textbf{case}\ \mathit{markKind}\ x\ \textbf{of} \\
\qquad\qquad\quad 1 \rightarrow pos + l\ x + 1 \leq m\ \wedge\ p'\ xs\ (pos + l\ x + 1, 1) \\
\qquad\qquad\quad 2 \rightarrow pos + l\ x + 1 \leq m\ \wedge\ p'\ xs\ (0, 2) \\
\qquad\qquad\quad 3 \rightarrow \mathit{False}
\end{array}
$$

$$
\begin{array}{l}
2 \to \textbf{case } \mathit{markKind}\ x \textbf{ of} \\
\qquad 1 \to \mathit{pos} + l\ x \le m\ \wedge\ p'\ xs\ (\mathit{pos} + l\ x, 1) \\
\qquad 2 \to \mathit{pos} + l\ x \le m\ \wedge\ p'\ xs\ (0, 2) \\
\qquad 3 \to \mathit{pos} + l\ x \le m\ \wedge\ p'\ xs\ (\mathit{pos} + l\ x, 3) \\
3 \to \textbf{case } \mathit{markKind}\ x \textbf{ of} \\
\qquad 1 \to \mathit{False} \\
\qquad 2 \to \mathit{False} \\
\qquad 3 \to \mathit{pos} + l\ x + 1 \le m\ \wedge\ p'\ xs\ (\mathit{pos} + l\ x + 1, 1),
\end{array}
$$

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

Next, we have to describe the weight function *wf*. We want to minimize the sum of the number of blanks except for the last line. The function *white* which returns the sum can be written as follows:

$$
\begin{array}{l}
\mathit{white} = +/ \circ \mathit{map}\ f \\
\qquad \textbf{where } f\ x = \textbf{case } \mathit{markKind}\ x \textbf{ of} \\
\qquad\qquad\qquad 1 \to -l\ x \\
\qquad\qquad\qquad 2 \to m - l\ x \\
\qquad\qquad\qquad 3 \to 0.
\end{array}
$$

Using this function, we can define the weight function *wf* as follows:

$$
\mathit{wf}\ x = -(\mathit{white}\ x).
$$

This can be easily transformed into the following form:

$$
\begin{array}{l}
\mathit{wf} = +/ \circ \mathit{map}\ f \\
\qquad \textbf{where } f\ x = \textbf{case } \mathit{markKind}\ x \textbf{ of} \\
\qquad\qquad\qquad 1 \to l\ x \\
\qquad\qquad\qquad 2 \to -m + l\ x \\
\qquad\qquad\qquad 3 \to 0.
\end{array}
$$

Now the paragraph formatting problem is written as follows:

$$
\mathit{mmm}\ p\ \mathit{wf}\ 3.
$$

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

5.2 Security Van Problem

The security van problem can be specified as follows [7].

> *Suppose a bank has a known sequence of deposits and withdrawals. For security reasons the total amount of cash in the bank should never exceed some fixed amount N, assumed to be at least as large as any single*

transaction. To cope with demand and supply, a security van can be called upon to deliver funds to the bank or to take away a surplus. The problem is to compute a schedule under which the van visits the bank a minimum number of times.

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

$$r, r + a_1, r + a_1 + a_2, \ldots, r + a_1 + \cdots + a_n$$

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

To formalize the constraint, define

$$\begin{aligned} \mathit{ceiling} &= \uparrow_{+/} / \circ \mathit{inits} \\ \mathit{floor} &= \downarrow_{+/} / \circ \mathit{inits} \end{aligned}$$

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

$$\mathit{ceiling}\ x - \mathit{floor}\ x \leq N.$$

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

$$\begin{array}{l}
p\ xs = p'\ xs\ (0,0,0) \\
p'\ [\,]\ (s,c,f) = \mathit{True} \\
p'\ (x : xs)\ (s,c,f) = \\
\quad \mathbf{let}\ (s',c',f') = (s + w\ x,\ c \uparrow_{id} s',\ f \downarrow_{id} s') \\
\quad \mathbf{in\ case}\ \mathit{markKind}\ x\ \mathbf{of} \\
\qquad 1 \rightarrow \mathbf{if}\ c' - f' \leq N\ \mathbf{then}\ p'\ xs\ (w\ x, 0 \uparrow_{id} w\ x, 0 \downarrow_{id} w\ x) \\
\qquad\qquad \mathbf{else}\ p'\ xs\ (w\ x, 0 \uparrow_{id} w\ x, 0 \downarrow_{id} w\ x) \\
\qquad 2 \rightarrow \mathbf{if}\ c' - f' \leq N\ \mathbf{then}\ p'\ xs\ (s',c',f')) \\
\qquad\qquad \mathbf{else}\ \mathit{False}
\end{array}$$

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

$$\begin{array}{l}
\mathit{times} = +/ \circ \mathit{map}\ f \\
\qquad\qquad \mathbf{where}\ f\ x = \mathbf{case}\ \mathit{markKind}\ x\ \mathbf{of} \\
\qquad\qquad\qquad\qquad 1 \rightarrow 1 \\
\qquad\qquad\qquad\qquad 2 \rightarrow 0
\end{array}$$

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

$$wf\ x = -\ (times\ x).$$

This can be easily transformed into the following form:

$$\begin{array}{l} wf = +/ \circ map\ f \\ \quad \textbf{where}\ f\ x = \textbf{case}\ markKind\ x\ \textbf{of} \\ \qquad\qquad\qquad 1 \rightarrow -1 \\ \qquad\qquad\qquad 2 \rightarrow 0. \end{array}$$

Now the security van problem is written as follows:

$$mmm\ p\ wf\ 2.$$

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

5.3 Knapsack Problem

The knapsack problem [13] is a well known combinatorial optimization problem. There are several problems called knapsack problem such as 0-1 knapsack problem, 0-1 multiple knapsack problem, multidimensional knapsack problem, and so on. Here we consider the simplest one, the 0-1 knapsack problem.

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

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

$$\begin{array}{lll} knap\ xs & = & knap'\ xs\ C \\ knap'\ [\,]\ e & = & True \\ knap'\ (x : xs)\ e & = & \textbf{case}\ markKind\ x\ \textbf{of} \\ & & \quad 1 \rightarrow \textbf{if}\ e \geq w\ x\ \textbf{then}\ knap'\ xs\ (e - w\ x)\ \textbf{else}\ False \\ & & \quad 2 \rightarrow knap'\ xs\ e \end{array}$$

Here, the function w returns the weight of the item. We want to maximize the value of selected items, so we can define the weight function wf as follows:

$$\begin{array}{l} wf = +/ \circ map\ f \\ \quad \textbf{where}\ f\ x = \textbf{case}\ markKind\ x\ \textbf{of} \\ \qquad\qquad\qquad 1 \rightarrow value\ x \\ \qquad\qquad\qquad 2 \rightarrow 0. \end{array}$$

Here, the function $value$ returns the value of the item.

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

$$mmm\ knap\ wf\ 2.$$

The weight function wf is written in the required form, and the property $knap$ can be easily rewritten into the required form, though we omit the form. By applying Theorem 1, we obtain $O(Cn)$ algorithm because $|Acc| = C + 1$, $k = 2$, and $|Class| = 2$.

5.4 Weighted Interval Selection Problem

Given a set of weighted intervals, the weighted interval selection problem is to select a maximum-weight subset such that any two selected intervals are disjoint [14]. An application of this problem is a scheduling of jobs whose start and end times are fixed and only one job can be executed at a time. We assume that start and end times are represented by integers. This assumption is natural in real-world jobs, where we mean that the time unit is a day or an hour or a minute or a second, and so on.

Suppose the job set is given as a list of jobs in the order of start time, that is, if job A starts earlier than job B, then job A appears earlier than job B in the list. We express a job by a 3-tuple of the start time, the time which it takes, and the weight of the job. Here we express start time by the difference from the previous job in the job list except for the first job. We express the time of the first job as 0. This way of expressing start time is for applying Theorem 1. For example, the list

$$jobs = [(0, 3, 2), (2, 4, 3), (3, 2, 5)].$$

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

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

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

$$\begin{array}{ll}
p\ xs & = p'\ xs\ 0 \\
p'\ [\,]\ e & = \mathit{True} \\
p'\ (x : xs)\ e & = \mathbf{case}\ \mathit{markKind}\ x\ \mathbf{of} \\
 & \qquad 1 \rightarrow \mathbf{if}\ e - s\ x > 0\ \mathbf{then}\ \mathit{False}\ \mathbf{else}\ p'\ xs\ (t\ x) \\
 & \qquad 2 \rightarrow \mathbf{if}\ e - s\ x > 0\ \mathbf{then}\ p'\ xs\ (e - s\ x)\ \mathbf{else}\ p'\ xs\ 0
\end{array}$$

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

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

$$\begin{array}{l} \mathit{wf} = +/ \circ \mathit{map}\ f \\ \qquad \textbf{where}\ f\ x = \textbf{case}\ \mathit{markKind}\ x\ \textbf{of} \\ \qquad\qquad\qquad\qquad 1 \rightarrow w\ x \\ \qquad\qquad\qquad\qquad 2 \rightarrow 0. \end{array}$$

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

Now the weighted interval selection problem is written as follows:

$$\mathit{mmm}\ p\ \mathit{wf}\ 2.$$

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

6 Related Work

In addition to the related work given in the introduction, we show some others in this section.

Bird calculated a linear-time algorithm for solving the maximum segment sum problem on lists [6], which is a kind of maximum marking problem. Bird *et al.* studied optimization problems, which include maximum marking problems, in a more general way that uses relational calculus [7]. Using relational calculus, they developed a very general framework to treat optimization problems. Their approach is called *thinning theory*, and the *thinning theorem* plays the central role. But when applying the thinning theorem, one has to find two preorders which meet prerequisites of the thinning theorem, which makes it difficult for the theorem to be used for mechanical program generation. And they didn't show the relation between the complexity of derived algorithms and specifications, in return for discussing in a very general framework. We instead focus on a useful class of optimization problems, maximum multi-marking problems, propose a very simple way to derive efficient algorithms, and assure the complexity of the derived algorithms.

Johan Jeuring proposed several fusion theorems, each of which deals with a class of optimization problems such as subsequence problems on lists, partition problems on lists, and so on [15]. In order to derive an efficient program for a problem by his method, one has to select a suitable fusion theorem, which is not necessary in our method.

De Moor considered a generic program for sequential decision processes [16] which are specified as follows.

```
listmin r . filter p . fold (choice fs) [c]
```

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

Recently, Bird showed that the maximum marking problems can be treated in the framework of thinning theory [2]. He assured the derived algorithm is a linear time algorithm, and showed the generic Haskell program for solving the maximum marking problems on polynomial data types. His method also requires that the property p should be suffix-closed.

In graph algorithms, Borie *et al.* proposed a method which enables the derivation of a linear time algorithm for solving the maximum marking problems on k-terminal graphs, a restricted class of graphs, from logical description of properties by a graph variant of monadic second order formula [5]. This graph variant of MSOL uses $\mathit{Inc}\ (v, e)$, which means a vertex v is an incident of an edge e, instead of the use of a successor function in ordinary MSOL [17]. Although appealing in theory, these methods are hardly useful in practice due to a huge constant factor for space and time.

7 Conclusions

In this paper, we propose an important theorem (generation rule) for generating efficient algorithms for solving maximum multi-marking problems, which can efficiently handle multi-marking, general weight functions, and property description with an accumulating parameter. This theorem leads to a more *general* framework for automatically generating efficient programs for solving a wider class of combinatorial optimization problems, covering those in the previous work [2–5].

Although this paper focuses on lists whereas the previous work [3] can treat any polynomial data structure, we believe that it is not difficult to extend this work to any polynomial data structure, and we leave it as our future work.

References

1. Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the 6th International Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 223–232, Copenhagen, Denmark, June 1993. ACM Press.
2. Richard Bird. Maximum marking problems, 2000. Available from http://www.comlab.ox.ac.uk/oucl/work/richard.bird/publications/mmp.ps.
3. 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, Montreal, Canada, September 2000. ACM Press.

4. Marshall W. Bern, Eugene L. Lawler, and A. L. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *Journal of Algorithms*, 8:216–235, 1987.
5. 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*, 7:555–581, 1992.
6. Richard Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.
7. Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1996.
8. Oege de Moor and Ganesh Sittampalam. Generic program transformation. In *Proceedings of the 3rd International Summer School on Advanced Functional Programming (AFP'98)*, LNCS 1608, pages 116–149, Braga, Portugal, September 1998. Springer-Verlag.
9. Richard Bird. *Introduction to Functional Programming using Haskell (second edition)*. Prentice Hall, 1998.
10. Richard Bird. An introduction to the theory of lists. In Manfred Broy, editor, *Logic of Programming and Calculi of Discrete Design*, NATO ASI Series 36, pages 5–42. Springer-Verlag, 1987.
11. Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.
12. 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)*, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.
13. Silvano Martello and Paolo Toth. *Knapsack Problems : Algorithms and Computer Implementations*. Wiley-Interscience series in discrete mathematics and optimization. John Wiley & Sons Ltd., 1990.
14. Thomas Erlebach and Frits Spieksma. Simple algorithms for a weighted interval selection problem. In *Proceedings of the 11th International Symposium on Algorithms and Computation (ISAAC'00)*, LNCS 1969, pages 228–240, Taipei, Taiwan, December 2000. Springer-Verlag.
15. Johan Jeuring. *Theories for Algorithm Calculation*. Ph.D thesis, Faculty of Science, Utrecht University, 1993.
16. Oege de Moor. A generic program for sequential decision processes. In *Proceedings of the 7th International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'95)*, LNCS 982, pages 1–23, Utrecht, the Netherlands, September 1995.
17. Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 133–192. Elsevier Science Publishers, 1990.