MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

Generator-based GG Fortress Library —Collection of GGs and Theories—

Kento EMOTO, Zhenjiang HU, Kazuhiko KAKEHI,
Kiminori MATSUZAKI and Masato TAKEICHI

METR 2008–17 March 2008

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

Generator-based GG Fortress Library —Collection of GGs and Theories—

Kento Emoto[†], Zhenjiang Hu[†], Kazuhiko Kakehi[‡], Kiminori Matsuzaki[†] and Masato Takeichi[†]

[†]Graduate School of Information Science and Technology, University of Tokyo
[‡]Division of University Corporate Relations (DUCR), University of Tokyo

{emoto,kmatsu,kaz}@ipl.t.u-tokyo.ac.jp
{hu,takeichi}@mist.i.u-tokyo.ac.jp

Abstract

We have proposed a novel library called "GG Library" on Fortress in the previous report. The library supports easy development of correct and efficient parallel programs, allowing users to write naive generate-and-test programs easily and uniformly with generator-of-generators that abstract generation of nested data structures. The library has an automatic optimization mechanism by dispatching efficient implementation to a user program written with generator-of-generators based on a collection of theories. To enrich the power of GG library, we need to make collections of generator-of-generators and their theories for optimization. This report collects generator-of-generators and their theories for optimization with formal discussion, as well as their implementations in GG library.

Contents

1 Introduction

In the previous report [EHK+08], we have proposed a novel library called "GG Library" (GG stands for Generator-of-Generators) for parallel computation on Fortress [ACH+]. The library supports easy development of correct and efficient parallel programs, and the library itself can grow up to cover a wider range of problems and to achieve better optimization power. These points match to the spirit of Fortress.

Features of the library are summarized as follows.

- Support for easy program development by generate-and-test specification
  Users can write naive generate-and-test programs easily and uniformly with *GGenerators* (generator-of-generators) that abstract generation of nested data structures. This generate-and-test specification covers wide range of problems, and users can make their programs by changing parameters of the specification, such as GGenerators, binary operators, functions and predicates to filter elements.

- Automatic optimization by dispatching correct and efficient implementation
  The library automatically dispatches efficient implementation to a user program written with GGenerators based on a collection of theories. Each GGenerator has its collection of theories and accompanying efficient implementations. If the library detects that the given user program satisfies conditions to apply some efficient implementation given by theories, then the library dispatches the efficient implementation to the user program. Properties of user programs such as distributivity of operators should be explicitly given by users when user-defined functions and operators are used in their programs.

- Growing Library
  The library grows in two directions: expressiveness and optimization power. The expressiveness of the library easily grows by extending the specification supported by the library. For example, adding a new GGenerator we can extend the specification to cover a wider range of problems. The power of optimization of the library easily grows by adding new knowledge of theories.

To enrich the power of GG library, we need to make collections of GGenerators and their theories for optimization.

This report collects GGenerators and their theories for optimization with formal discussion, as well as their implementations in GG library. Section 2 shows an overview of GG library. Section 3 shows a collection of GGenerators with some example problems. Section 4 formalizes a collection of theories for optimization. Section 5 shows implementation of GGenerators and knowledge of their theories in GG library, as well as the core of GG library. Finally, Section 6 concludes this report. Note that, in Section 3, Section 4 and Section 5, subsections of the same subsection-number correspond to the same target. For example, for GGenerator inits, Section 3.3, Section 4.3 and Section 5.3 give example problems, theories, and implementation in GG library, respectively.

2 Overview of GG Library

This section briefly reviews our proposed GG library [EHK+08].

2.1 Structure of GG Library

Figure 1 illustrates the structure of GG Library. There are two kinds of collections in the library. One is a collection of GGeneratorsthat are used to describe specification of problems. Each GGeneratorabstracts generation of a nested data structure. The collection of GGeneratorsprovides an interface for easy and uniform description of user programs. The collection of GGeneratorswith their examples is shown in Section 3.

The other is a collection of theories and accompanying efficient implementations dispatched to user programs. Basically, each GGenerator has its own collection of theories and efficient implementations. Given a user program written with GGenerators, the library checks applicable conditions of theories against the given program, and then if the program satisfies the condition, the library dispatches the accompanying efficient implementation. The collection of theories is shown in Section 4.

Besides those two specific collections, the library has a collection of traits to describe mathematical properties of user programs. Those traits are used to determine whether a given user program satisfies applicable conditions of theories.

Figure 1 also illustrates how GG library grows in two directions: expressiveness and optimization power. The expressiveness of the library grows by adding new GGenerators to support a wider range of problem specifications. The power of optimization grows by adding new knowledge of theories to dispatch more efficient implementation to more user programs.

2.2 Behavior of GG Library

Figure 2 shows the behavior of our GG Library with concrete examples for a program to solve "Maximum Prefix Sum Problem" (Section 3.3.2.) The objective of the program is to find the maximum sum of a prefix of the input sequence. The behavior of GG Library for dispatching implementation to a user program is separated into the following two phases.

Phase 1. *Desugaring a user program into invocations of method generate2 of GGenerators*
In the first phase, the library desugars a user program written with for-loops or comprehensions into invocations of method *generate2* of GGenerators (Section 5.1.1) used in the program. Method *generate2* performs nested reductions on generated nested data structures.

Figure 2-(a) shows an example of such user programs written with comprehensions. This program uses GGenerator *inits* (Sections 3.3, 4.3, 5.3) to generate prefix segments of the input sequence x. The desugaring process transforms the user program into the program shown in Figure 2-(b). Two reduction operators + and `MAX` are given to method *generate2* as its arguments enclosed in objects: SumReductionZZ32 and MaxReductionZZ32. The other arguments of method *generate2* are default values such as IdFunction (Section 5.2.2) and TrueListPredicate (Section 5.2.3).

Phase 2. *Dispatching implementation within the invocation of method generate2*
After the desugaring process, the library dispatches implementation within the invocation of *generate2*. In this phase, the library checks whether properties of the given arguments of method *generate2* satisfy the applicable condition of each theory of the GGenerator. And then, if it is found that the applicable condition is satisfied, the library performs computation of the nested reductions by corresponding efficient implementation.

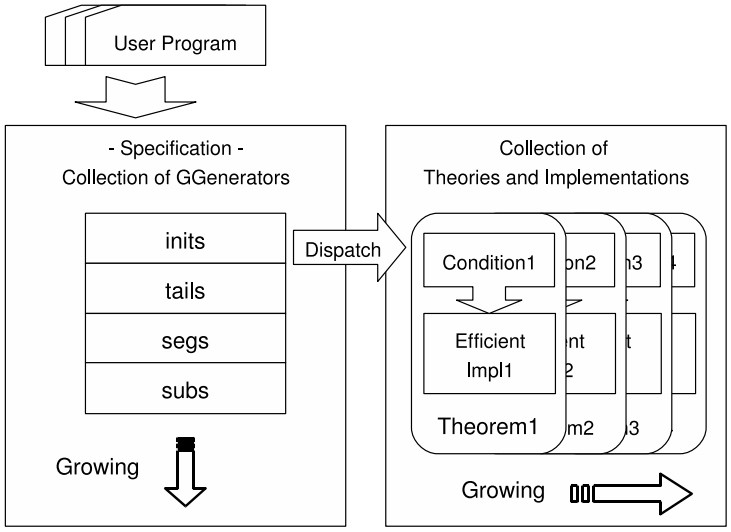
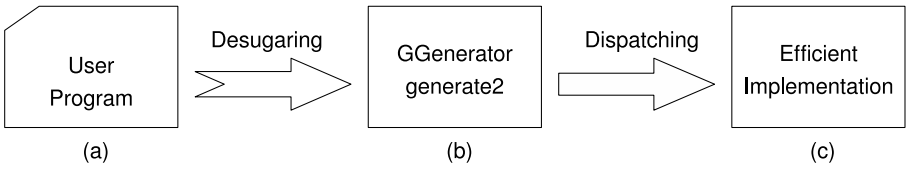


Figure 1: Structure of GG library



$$\texttt{BIG MAX}\,[\,\textstyle\sum[\,a \mid a \leftarrow y\,] \mid y \leftarrow \mathit{inits}\,x\,]$$

(a) User program of nested reductions with comprehensions

$$\begin{aligned} \mathit{mis\_xs} = \mathit{inits}(x).\mathit{generate}_2[\![\,&\mathrm{MaxReductionZZ32}, \mathrm{IdFunction}[\![\mathbb{Z}32]\!],\\ &\mathrm{SumReductionZZ32}, \mathrm{IdFunction}[\![\mathbb{Z}32]\!],\\ &\mathrm{TrueListPredicate}[\![\mathbb{Z}32]\!], \mathbb{Z}32, \mathbb{Z}32]\!]\\ &\quad(\mathrm{MaxReductionZZ32}, \mathrm{IdFunction}[\![\mathbb{Z}32]\!],\\ &\quad\ \mathrm{SumReductionZZ32}, \mathrm{IdFunction}[\![\mathbb{Z}32]\!],\\ &\quad\ \mathrm{TrueListPredicate}[\![\mathbb{Z}32]\!]) \end{aligned}$$

(b) Desugared program

$$\begin{aligned} &\texttt{opr MPS}(a, b) = \texttt{do}\\ &\quad (m_1, s_1) = a\\ &\quad (m_2, s_2) = b\\ &\quad (m_1\ \texttt{MAX}\ (s_1 + m_2), s_1 + s_2)\\ &\texttt{end}\\ &\texttt{object}\ \mathrm{MPSReduction}\ \texttt{extends}\ \mathrm{Reduction}[\![(\mathrm{Number}, \mathrm{Number})]\!]\\ &\quad \mathit{empty}(): (\mathrm{Number}, \mathrm{Number}) = (-\mathit{infinity}, 0)\\ &\quad \mathit{join}(a : (\mathrm{Number}, \mathrm{Number}), b : (\mathrm{Number}, \mathrm{Number})): (\mathrm{Number}, \mathrm{Number}) = a\ \texttt{MPS}\ b\\ &\texttt{end}\\ &\texttt{opr BIG MPS}\ [\![E]\!](g : (\mathrm{Reduction}[\![E]\!], E \to E) \to E): E = g(\mathrm{MPSReduction}, \texttt{fn}\ a \Rightarrow a)\\ &(r_1, r_2) = \texttt{BIG MPS}\ [\,(a, a) \mid a \leftarrow x]\\ &r_1 \end{aligned}$$

(c) Efficient implementation for distributive operators

Figure 2: Two-phase behavior of GG library (with examples for "Maximum Prefix Sum")

To tell properties of the arguments to the library, the operator/predicate/function should extend a set of suitable traits when it has some mathematical properties. For example, usual plus operator + has distributivity over the maximum operator `MAX`, so SumReduction should extend trait LeftDistributiveOver⟦MaxReduction⟧ to tell its distributivity (Section 5.2.1). Since properties of the given arguments are specified by traits, checking of applicable conditions of theories are performed by checking types of the arguments (Section 5.1.2).

For example, for the desugared program shown in Figure 2-(b), the library checks whether the reduction operator has distributivity. In this case, the library finds SumReductionZZ32 (operator +) distributes over MaxReductionZZ32 (operator `MAX`), since SumReductionZZ32 extends LeftDistributiveOver⟦MaxReductionZZ32⟧. Thus, the library uses the efficient implementation shown in Figure 2-(c) to perform the nested reduction of the program. This efficient implementation is given by Theorem 15 in Section 4.

3 A Collection of GGenerators with Example Problems

This section gives a collection of GGenerators and their example applications. More examples may be found in [Bir01, SHTO00, SHT01, Zha02, Zan92, Jeu93].

Notation in this section follows that of Haskell [Bir98].

Note that not all of GGenerators shown in this section are currently implemented in GG library.

3.1 Preliminaries

In this section, some functions are defined to describe example problems, and a class of problem specifications called "generate-and-test" is introduced. Also, some predicates used in the following examples are defined in this section.

3.1.1 Basic Functions

We use the following functions to define specification of problems.

$$
\begin{array}{lcl}
\mathsf{map}\, f\, [x_1, x_2, \ldots, x_n] & = & [f\ x_1, f\ x_2, \ldots, f\ x_n] \\
\mathsf{reduce}\, (\oplus)\, [x_1, x_2, \ldots, x_n] & = & x_1 \oplus x_2 \oplus \cdots \oplus x_n \\
\mathsf{hom}\, (\oplus, f)\, [x_1, x_2, \ldots, x_n] & = & f\ x_1 \oplus f\ x_2 \oplus \cdots \oplus f\ x_n \\
\mathsf{last}\ [x_1, x_2, \ldots, x_n] & = & x_n \\
\mathsf{head}\ [x_1, x_2, \ldots, x_n] & = & x_1 \\
\mathsf{length}\ [x_1, x_2, \ldots, x_n] & = & n \\
\mathsf{filter}\, p\, [x_1, x_2, \ldots, x_n] & = & [x_{i_1}, x_{i_2}, \ldots, x_{i_m}]\ \ \textbf{where}\ p\ (x_{i_j}) = \mathsf{True}
\end{array}
$$

Function $\mathsf{map}\, f$ applies the given function f to every element of the input list. Function $\mathsf{reduce}\, (\oplus)$ performs 'summation' of the input list with the given associative binary operator $\oplus$. Function $\mathsf{hom}\, (\oplus, f)$ performs $\mathsf{reduce}\, (\oplus)$ after $\mathsf{map}\, f$, i.e., $\mathsf{hom}\, (\oplus, f) = \mathsf{reduce}\, (\oplus) \circ \mathsf{map}\, f$. Function last takes the last element of the input, while function head takes the head. Function length returns the length of the input. Function $\mathsf{filter}\, p$ removes every element that does not satisfy the given predicate p.

Maximum and minimum are denoted by the following arrows.

$$
\begin{array}{rcll}
a \uparrow b & = & a & \text{if } a \geq b \\
 & = & b & \text{otherwise} \\
a \downarrow b & = & a & \text{if } a \leq b \\
 & = & b & \text{otherwise}
\end{array}
$$

For a function f, $\uparrow_f$ and $\downarrow_f$ are defined as follows.

$$
\begin{array}{rcll}
a \uparrow_f b & = & a & \text{if } f\ a \geq f\ b \\
 & = & b & \text{otherwise} \\
a \downarrow_f b & = & a & \text{if } f\ a \leq f\ b \\
 & = & b & \text{otherwise}
\end{array}
$$

For a tuple of any length, projection functions π_i are defined as follows.

$$\pi_i\ (a_1, \ldots, a_i, \ldots, a_n) \quad = \quad a_i$$

For instance, $\pi_1\ (a, b) = a$ and $\pi_2\ (a, b) = b$.

3.1.2 GGenerators and Generate-and-test Specification

A GGenerator is a function to generate a nested list. For example, inits shown in the next section generates a nested list of which element is a prefix of an input list, as shown in Section 3.3 and Section 4.3.1.

Fortress' generator abstracts generation of a flat data structure, i.e., a flat list, and reduction on the flat structure. Our GGenerator extends the generator to abstract generation of a nested data structure, i.e., a list of lists, and nested reductions on the nested data structure. Of course, Fortress' generator can perform nested reductions by nested use of Generators. The advantage of our GGenerator's nested reductions is that a GGenerator can use relationship between the pair of reductions to perform the reductions with specialized efficient implementation given by a collection of theories on GGenerators.

We focus on problems described with generate-and-test specification. The general form of the generate-and-test specification with GGenerators is shown below.

$$\mathsf{hom}\,(\oplus, f) \circ \mathsf{map}\,(\mathsf{hom}\,(\otimes, g)) \circ \mathsf{filter}\,p \circ gg$$

Here, gg is one instance of GGenerators, which generates a nested list, and the generated nested list is consumed by the nested reductions, $\mathsf{hom}\,(\oplus, f)$ for the outer list and $\mathsf{hom}\,(\otimes, g)$ for the inner lists, after filtered with predicate p.

For example, simple nested reductions with two binary operators $\oplus$ and $\otimes$ for a GGenerator gg can be described in the generate-and-test specification as follows.

$$\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ gg = \mathsf{hom}\,(\oplus, \mathsf{id}) \circ \mathsf{map}\,(\mathsf{hom}\,(\otimes, \mathsf{id})) \circ \mathsf{filter}\,\mathit{true} \circ gg$$

Here, *true* is the predicate returning always True. The equality of the left hand side (the simple nested reductions) and the right hand side (the generate-and-test specification) is easily shown. First, by the definition of $\mathsf{reduce}\,(\oplus)$, we get $\mathsf{hom}\,(\oplus, \mathsf{id}) = \mathsf{reduce}\,(\oplus)$ and $\mathsf{hom}\,(\otimes, \mathsf{id}) = \mathsf{reduce}\,(\otimes)$. Next, by the definition of $\mathsf{filter}\,p$, $\mathsf{filter}\,\mathit{true} = \mathsf{id}$ since *true* returns always True.

Examples shown in the following sections are instances of the generate-and-test specification with GGenerators.

3.1.3 Predicates for Examples

The following predicates on lists are used in examples shown in the following sections.

Predicate *ascending* returns True if the given list is ascending (i.e., each element is smaller than the following element). For example, applying *ascending* to a sorted list $[-2, 4, 5]$ we get $\mathit{ascending}([-2, 4, 5]) = \mathsf{True}$, while $\mathit{ascending}([3, 6, 2]) = \mathsf{False}$ since 6 is not smaller than 2.

Predicate *descending* returns True if the given list is descending (i.e., each element is bigger than the following element). For example, applying *descending* to a descendingly sorted list $[5, 3, 0]$ we get $\mathit{descending}([5, 3, 0]) = \mathsf{True}$, while $\mathit{ascending}([1, 9, 7]) = \mathsf{False}$ since 1 is not bigger than 9.

Predicate smooth_c returns True if a difference between any successive elements in the given list is less than or equal to c. For example, $\mathit{smooth}_2([6, 5, 7]) = \mathsf{True}$, since both of differences $1 = |6 - 5|$ and $2 = |5 - 7|$ are less than or equal to 2. But, $\mathit{smooth}_2([6, 5, 8]) = \mathsf{False}$ because the difference $3 = |5 - 8|$ is greater than 2.

Predicate *high* returns True if the maximum element of the given list is greater than the length of the list. For example, $\mathit{high}([1, 4, 3]) = \mathsf{True}$, since the maximum element 4 is greater than the length of the list, i.e., 3. But, $\mathit{high}([1, 4, 3, 2]) = \mathsf{False}$ because the maximum element 4 is not greater than the length of the list 4.

3.3 GGenerator inits

GGenerator inits generates prefixes of an input list. For example, applying inits to $[1, 3, 1, -7, 2, 4]$ we get the following nested list.

$$\mathsf{inits}\ [1, 3, 1, -7, 2, 4] = [[1], [1, 3], [1, 3, 1], [1, 3, 1, -7], [1, 3, 1, -7, 2], [1, 3, 1, -7, 2, 4]]$$

Here, each element of the resulting list is a prefix of the given list.

3.3.1 Prefix Sums

The most famous application of inits is prefix sums, which has many applications [Ble90]. The statement is as follows.

Example 1 (Prefix Sums)**.** *Given a list and an associative binary operator, find sums for all prefixes of the given list.*

For example, prefix sums of $[1, 3, 1, -7, 2, 4]$ with operator $\oplus$ is $[1, 1 \oplus 3, 1 \oplus 3 \oplus 1, 1 \oplus 3 \oplus 1 \oplus -7, 1 \oplus 3 \oplus 1 \oplus -7 \oplus 2, 1 \oplus 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4]$. Using inits, prefix sums are easily obtained by applying reduction with $\oplus$ to all prefixes generated by inits.

$$\begin{aligned}
&\mathsf{map}\,(\mathsf{reduce}\,(\oplus))\,(\mathsf{inits}\ [1, 3, 1, -7, 2, 4]) \\
&= \mathsf{map}\,(\mathsf{reduce}\,(\oplus))\,[[1], [1, 3], [1, 3, 1], [1, 3, 1, -7], [1, 3, 1, -7, 2], [1, 3, 1, -7, 2, 4]] \\
&= [1, 1 \oplus 3, 1 \oplus 3 \oplus 1, 1 \oplus 3 \oplus 1 \oplus -7, 1 \oplus 3 \oplus 1 \oplus -7 \oplus 2, 1 \oplus 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4]
\end{aligned}$$

Sample fortress code for prefix sums with the usual plus operator + is shown below.

$$\begin{aligned}
&x = \langle\, 1, 3, 1, -7, 2, 4 \,\rangle \\
&\mathit{prefix_sums} = \langle\, \textstyle\sum y \mid y \leftarrow \mathit{inits}\ x \,\rangle
\end{aligned}$$

A related theory is found in Section 4.3.1. Related implementation on GG library is found in Section 5.3.1.

3.3.2 Maximum Prefix Sum

The maximum prefix sum problem is one of optimization problems on sequences. Its statement is as follows.

Example 2 (Maximum Prefix Sum)**.** *Given a list, find the maximum sum of a prefix of the given list.*

For example, the maximum prefix sum of $[1, 3, 1, -7, 2, 4]$ is 5 since its prefix sums (with the usual plus operator +) are $[1, 4, 5, -2, 0, 4]$. So, using inits we can get the maximum prefix sum easily by applying reduction with the maximum operator $\uparrow$ for the prefix sums.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{inits}\ [1, 3, 1, -7, 2, 4])) \\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[1], [1, 3], [1, 3, 1], [1, 3, 1, -7], [1, 3, 1, -7, 2], [1, 3, 1, -7, 2, 4]]) \\
&= \mathsf{reduce}\,(\uparrow)\,([1, 1+3, 1+3+1, 1+3+1+-7, 1+3+1+-7+2, 1+3+1+-7+2+4]) \\
&= 5
\end{aligned}$$

Sample fortress code for the maximum prefix sum is shown below.

$$\begin{aligned}
&x = \langle\, 1, 3, 1, -7, 2, 4 \,\rangle \\
&(* \text{ with complemension } *) \\
&\mathit{mis} = \texttt{BIG MAX}\ \langle\, \textstyle\sum y \mid y \leftarrow \mathit{inits}\ x \,\rangle \\
&(* \text{ with for-loop } *) \\
&m : \mathbb{Z}32 = -\mathit{infinity} \\
&\texttt{for}\ y \leftarrow \mathit{inits}\ x\ \texttt{do} \\
&\quad s = \textstyle\sum y \\
&\quad \texttt{atomic}\ m\texttt{MAX} = s \\
&\texttt{end}
\end{aligned}$$

A related theory is found in Section 4.3.2. Related implementation on GG library is found in Section 5.3.2.

3.3.3 Maximum p-Prefix Sum

This is a variant of the maximum prefix sum problem. The statement is as follows.

Example 3 (Maximum p-Prefix Sum)**.** *Given a list and a predicate p, find the maximum sum of its prefix satisfying the given predicate p.*

For example, the maximum ascending-prefix sum of $[1, 3, 1, -7, 2, 4]$ is 4 since its ascending prefixes are $[1]$ and $[1, 3]$. Using inits and filter, we can get the maximum ascending-prefix sum easily as follows.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{filter}\;\mathit{ascending}\,(\mathsf{inits}\;[1,3,1,-7,2,4])))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{filter}\;\mathit{ascending}\,[[1],[1,3],[1,3,1],[1,3,1,-7],\\
&\qquad\qquad [1,3,1,-7,2],[1,3,1,-7,2,4]]))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[1],[1,3]])\\
&= \mathsf{reduce}\,(\uparrow)\;\;[1, 1+3]\\
&= 4
\end{aligned}$$

Sample fortress code for the maximum ascending-prefix sum is shown below.

$$\begin{aligned}
&x = \langle\, 1, 3, 1, -7, 2, 4 \,\rangle\\
&\mathit{mais} = \mathtt{BIG\,MAX}\,\langle\, \textstyle\sum y \mid y \leftarrow \mathit{inits}\; x, \mathit{ascending}(y) \,\rangle
\end{aligned}$$

A related theory is found in Section 4.3.3. Related implementation on GG library is found in Section 5.3.3.

3.4 GGenerator tails

GGenerator tails generates suffixes of an input list. For example, applying tails to $[1, 3, 1, -7, 2, 4]$ we get the following nested list.

$$\mathsf{tails}\;[1,3,1,-7,2,4] = [[1,3,1,-7,2,4],[3,1,-7,2,4],[1,-7,2,4],[-7,2,4],[2,4],[4]]$$

Here, each element of the resulting list is a suffix of the given list.

3.4.1 Suffix Sums

The most famous application of tails is suffix sums, which is converse computation of prefix sums shown in Section 3.3.1. The statement is as follows.

Example 4 (Suffix Sums)**.** *Given a list and an associative binary operator, find sums for all suffixes of the given list.*

For example, suffix sums of $[1, 3, 1, -7, 2, 4]$ with operator $\oplus$ is $[1 \oplus 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4, 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4, 1 \oplus -7 \oplus 2 \oplus 4, -7 \oplus 2 \oplus 4, 2 \oplus 4, 4]$. Using tails, we can get suffix sums easily by applying reduction with $\oplus$ to all suffixes generated by tails.

$$\begin{aligned}
&\mathsf{map}\,(\mathsf{reduce}\,(\oplus))\,(\mathsf{tails}\;[1,3,1,-7,2,4])\\
&= \mathsf{map}\,(\mathsf{reduce}\,(\oplus))\,[[1,3,1,-7,2,4],[3,1,-7,2,4],[1,-7,2,4],[-7,2,4],[2,4],[4]]\\
&= [1 \oplus 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4, 3 \oplus 1 \oplus -7 \oplus 2 \oplus 4, 1 \oplus -7 \oplus 2 \oplus 4, -7 \oplus 2 \oplus 4, 2 \oplus 4, 4]
\end{aligned}$$

Sample fortress code for suffix sums with the usual plus operator + is shown below.

$$\begin{aligned}
&x = \langle\, 1, 3, 1, -7, 2, 4 \,\rangle\\
&\mathit{suffix_sums} = \langle\, \textstyle\sum y \mid y \leftarrow \mathit{tails}\; x \,\rangle
\end{aligned}$$

A related theory is found in Section 4.4.1. Related implementation on GG library is found in Section 5.4.1.

3.4.2 Maximum Suffix Sum

The maximum suffix sum problem is one of optimization problems on sequences. Its statement is as follows.

Example 5 (Maximum Suffix Sum)**.** *Given a list, find the maximum sum of a suffix of the given list.*

For example, the maximum suffix sum of $[1, 3, 1, -7, 2, 4]$ is 6 since its suffix sums (with the usual plus operator $+$) are $[4, 3, 0, -1, 6, 4]$. So, using tails we can get the maximum suffix sum easily by applying reduction with the maximum operator $\uparrow$ for suffix sums.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{tails}\ [1,3,1,-7,2,4])) \\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[1,3,1,-7,2,4],[3,1,-7,2,4],[1,-7,2,4],[-7,2,4],[2,4],[4]] \\
&= \mathsf{reduce}\,(\uparrow)\,[1+3+1+-7+2+4, 3+1+-7+2+4, 1+-7+2+4, -7+2+4, 2+4, 4] \\
&= 6
\end{aligned}$$

Sample fortress code for the maximum suffix sum is shown below.

$$\begin{aligned}
&x = \langle\, 1, 3, 1, -7, 2, 4 \,\rangle \\
&(* \text{ with complemension } *) \\
&mts = \texttt{BIG MAX}\ \langle\, \textstyle\sum y \mid y \leftarrow \mathit{tails}\ x \,\rangle \\
&(* \text{ with for-loop } *) \\
&m : \mathbb{Z}32 = -\mathit{infinity} \\
&\texttt{for}\ \ y \leftarrow \mathit{tails}\ x\ \texttt{do} \\
&\quad s = \textstyle\sum y \\
&\quad \texttt{atomic}\ m\texttt{MAX} = s \\
&\texttt{end}
\end{aligned}$$

A related theory is found in Section 4.4.2. Related implementation on GG library is found in Section 5.4.2.

3.4.3 Maximum p-Suffix Sum

This is a variant of the maximum suffix sum problem. The statement is as follows.

Example 6 (Maximum p-Suffix Sum)**.** *Given a list and a predicate p, find the maximum sum of its suffix satisfying the given predicate p.*

For example, the maximum ascending-suffix sum of $[1, 3, 1, -7, 2, 4]$ is 6 since its ascending suffixes are $[-7, 2, 4]$, $[2, 4]$ and $[4]$. Using tails and filter , we can get the maximum ascending-suffix sum easily as follows.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{filter}\ \mathit{ascending}\,(\mathsf{tails}\ [1,3,1,-7,2,4]))) \\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{filter}\ \mathit{ascending}\,[[1,3,1,-7,2,4],[3,1,-7,2,4], \\
&\qquad\qquad [1,-7,2,4],[-7,2,4],[2,4],[4]])) \\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[-7,2,4][2,4],[4],]) \\
&= \mathsf{reduce}\,(\uparrow)\,[-7+2+4, 2+4, 4] \\
&= 6
\end{aligned}$$

Sample fortress code for the maximum ascending-suffix sum is shown below.

$$\begin{aligned}
&x = \langle\, 1, 3, 1, -7, 2, 4 \,\rangle \\
&mats = \texttt{BIG MAX}\ \langle\, \textstyle\sum y \mid y \leftarrow \mathit{tails}\ x, \mathit{ascending}(y) \,\rangle
\end{aligned}$$

A related theory is found in Section 4.4.3. Related implementation on GG library is found in Section 5.4.3.

3.5 GGenerator segs

GGenerator segs generates segments (continuous subsequences) of an input list. The segments are listed in the lexicographic order. For example, applying segs to $[3, 2, -7, 4, 2]$ we get the following nested list.

$$\begin{aligned}\mathsf{segs}\ [3,2,-7,4,2] \quad = \quad & [[3],[3,2],[3,2,-7],[3,2,-7,4],[3,2,-7,4,2],[2],[2,-7],\\ & [2,-7,4],[2,-7,4,2],[-7],[-7,4],[-7,4,2],[4],[4,2],[2]]\end{aligned}$$

3.5.2 Maximum Segment Sum

The most famous application of segs is maximum segment sum problem [Bir87,Jeu93,SHTO00], which is one of optimization problems on sequences. The statement is as follows.

Example 7 (Maximum Segment Sum)**.** *Given a list, find the maximum sum of a segment of the given list.*

For example, the maximum segment sum of $[3, 2, -7, 4, 2]$ is 6 that is the sum of its segment $[4, 2]$. We can get the maximum segment sum of the given list using segs as follows. First, we generate all segments of the given list by segs. Then, we apply reduction with the usual plus operator $+$ to generated segments to get all segment sums. Finally, applying reduction with maximum-operator to those segment sums, we get the maximum segment sum of the given list.

$$\begin{aligned}&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{segs}\ [3,2,-7,4,2]))\\ &= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[3],[3,2],[3,2,-7],[3,2,-7,4],[3,2,-7,4,2],[2],[2,-7],\\ &\qquad\qquad [2,-7,4],[2,-7,4,2],[-7],[-7,4],[-7,4,2],[4],[4,2],[2]])\\ &= \mathsf{reduce}\,(\uparrow)\,[3,5,-2,2,4,2,-5,-1,1,-7,-3,-1,4,6,2]\\ &= 6\end{aligned}$$

Sample fortress code for the maximum prefix sum is shown below.

```
x = ⟨ 3, 2, −7, 4, 2 ⟩
(∗ with complemension ∗)
mss = BIG MAX ⟨ ∑ y | y ← segs x ⟩
(∗ with for-loop ∗)
m : ℤ32 = −infinity
for y ← segs x do
  s = ∑ y
  atomic mMAX = s
end
```

A related theory is found in Section 4.5.2. Related implementation on GG library is found in Section 5.5.2.

3.5.3 Maximum p-Segment Sum

This is a variant of the maximum segment sum problem. The statement is as follows.

Example 8 (Maximum p-Segment Sum)**.** *Given a list and a predicate p, find the maximum sum of its segment satisfying the given predicate p.*

For example, the maximum ascending-segment sum of $[3, 2, -7, 4, 2]$ is 4 since its ascending-segments are all singletons. The maximum descending-segment sum of $[3, 2, -7, 4, 2]$ is 6 since

its descending-segments are $[3,2]$, $[3,2,-7]$, $[2,-7]$, $[4,2]$ and singletons. Using segs and filter, we can get the maximum descending-segment sum easily as follows.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{filter}\ \mathit{descending}\,(\mathsf{segs}\ [3,2,-7,4,2])))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,(\mathsf{filter}\ \mathit{descending}\ [[3],[3,2],[3,2,-7],[3,2,-7,4],[3,2,-7,4,2],\\
&\qquad\qquad [2],[2,-7],[2,-7,4],[2,-7,4,2],[-7],[-7,4],\\
&\qquad\qquad [-7,4,2],[4],[4,2],[2]]))\\
&= \mathsf{reduce}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{reduce}\,(+))\,[[3],[3,2],[3,2,-7],[2],[2,-7],[-7],[4],[4,2],[2]])\\
&= \mathsf{reduce}\,(\uparrow)\,[3,5,-2,2,-5,-7,4,6,2]\\
&= 6
\end{aligned}$$

Sample fortress code for the maximum descending-segment sum is shown below.

$$\begin{aligned}
&x = \langle\, 3,2,-7,4,2 \,\rangle\\
&mdss = \texttt{BIG MAX}\,\langle \textstyle\sum y \mid y \leftarrow \mathit{segs}\ x, \mathit{descending}(y) \,\rangle
\end{aligned}$$

A related theory is found in Section 4.5.3. Related implementation on GG library is found in Section 5.5.3.

3.5.4 Longest-p Segment

Longest-p segment problem [Zan92] is also one of optimization problems on sequences. Its statement is as follows.

Example 9 (Longest-p Segment)**.** *Given a list and a predicate, find the longest segment (continuous subsequence) of the list that satisfies the predicate.*

For example, the longest-ascending segment of $[3,2,-7,4,2]$ is $[-7,4]$, while the longest-descending segment is $[3,2,-7]$. There are many instances of longest-p segment for various predicates. The longest segment satisfying a given predicate p is obtained by using segs and filtering. Some instances are shown below.

The longest ascending segment of an input list is obtained by using the predicate *ascending*.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,(\mathsf{filter}\ \mathit{ascending}\,(\mathsf{segs}\ [3,2,-7,4,2]))\\
&= \mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,(\mathsf{filter}\ \mathit{ascending}\,(\mathsf{segs}\ [[3],[3,2],[3,2,-7],[3,2,-7,4],[3,2,-7,4,2],\\
&\qquad\qquad [2],[2,-7],[2,-7,4],[2,-7,4,2],[-7],[-7,4],\\
&\qquad\qquad [-7,4,2],[4],[4,2],[2]]))\\
&= \mathsf{reduce}\,(\uparrow_{\mathsf{length}})\ \ [[3],[2],[-7],[-7,4],[4],[2]]\\
&= [2,5]
\end{aligned}$$

The longest smooth segment of an input list is obtained by using the predicate smooth_c.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,(\mathsf{filter}\ \mathit{smooth}_4\,(\mathsf{segs}\ [3,2,-7,4,2]))\\
&= \mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,(\mathsf{filter}\ \mathit{smooth}_4\,(\mathsf{segs}\ [[3],[3,2],[3,2,-7],[3,2,-7,4],[3,2,-7,4,2],\\
&\qquad\qquad [2],[2,-7],[2,-7,4],[2,-7,4,2],[-7],[-7,4],\\
&\qquad\qquad [-7,4,2],[4],[4,2],[2]]))\\
&= \mathsf{reduce}\,(\uparrow_{\mathsf{length}})\ \ [[3],[3,2],[2],[-7],[4],[4,2],[2]]\\
&= [3,2]
\end{aligned}$$

The longest high segment of an input list is obtained by using predicate *high*.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,(\mathsf{filter}\ \mathit{high}\,(\mathsf{segs}\ [3,2,-7,4,2]))\\
&= \mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,(\mathsf{filter}\ \mathit{high}\,(\mathsf{segs}\ [[3],[3,2],[3,2,-7],[3,2,-7,4],[3,2,-7,4,2],[2],[2,-7],\\
&\qquad\qquad [2,-7,4],[2,-7,4,2],[-7],[-7,4],[-7,4,2],[4],[4,2],[2]]))\\
&= \mathsf{reduce}\,(\uparrow_{\mathsf{length}})\ \ [[3],[3,2],[2],[4],[4,2],[2]]\\
&= [3,2]
\end{aligned}$$

Sample fortress code for the longest p-segment is shown below. The code computes the longest p-segment with its length.

$$x = \langle\, 3, 2, -7, 4, 2 \,\rangle$$
$$las = \texttt{BIG MAX} \,\langle\, (length\ y, y) \mid y \leftarrow segs\ x, ascending(y) \,\rangle$$
$$lss = \texttt{BIG MAX} \,\langle\, (length\ y, y) \mid y \leftarrow segs\ x, smooth(4, y) \,\rangle$$
$$lhs = \texttt{BIG MAX} \,\langle\, (length\ y, y) \mid y \leftarrow segs\ x, high\ y \,\rangle$$

3.6 GGenerator subs

GGenerator subs generates all subsequences (sub-lists) of an input list. Generated subsequences are listed in the lexicographic order. For example, applying subs to $[2, 5, -3, 4]$ we get the following nested list.

$$\begin{aligned}\mathsf{subs}\ [2, 3, -3, 4] = [[2], [2, 3], [2, 3, -3], [2, 3, -3, 4], [2, 3, 4], [2, -3], [2, -3, 4], [2, 4],\\ [3], [3, -3], [3, -3, 4], [3, 4], [-3], [-3, 4], [4]]\end{aligned}$$

3.6.1 Longest-p Subsequence

Longest-p subsequence problem is one of optimization problems on sequences [Jeu93].

Example 10 (Longest-p Subsequence)**.** *Given a list and a predicate, find the longest subsequence of the list that satisfies the predicate.*

For example, the longest-ascending subsequnce of $[2, 3, -3, 4]$ is $[2, 3, 4]$, while the longest-descending subsequnce is $[2, -3]$. There are many instances of longest-p subsequnce for various predicates. The longest subsequnce satisfying a given predicate p is obtained by using subs and filter. Some instances are shown blow.

The longest ascending subsequence of an input list is obtained by using subs and the predicate *ascending*.

$$\begin{aligned}&\mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,(\mathsf{filter}\ ascending\ (\mathsf{subs}\ [2, 3, -3, 4]))\\ &\mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,(\mathsf{filter}\ ascending\ [[2], [2, 3], [2, 3, -3], [2, 3, -3, 4], [2, 3, 4], [2, -3], [2, -3, 4], [2, 4],\\ &\qquad\qquad [3], [3, -3], [3, -3, 4], [3, 4], [-3], [-3, 4], [4]])\\ &= \mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,[[2], [2, 3], [2, 3, 4], [2, 4], [3], [3, 4], [-3], [-3, 4], [4]]\\ &= [2, 3, 4]\end{aligned}$$

The longest descending subsequence of an input list is obtained by using subs and the predicate *descending*.

$$\begin{aligned}&\mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,(\mathsf{filter}\ descending\ (\mathsf{subs}\ [2, 3, -3, 4]))\\ &\mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,(\mathsf{filter}\ descending\ [[2], [2, 3], [2, 3, -3], [2, 3, -3, 4], [2, 3, 4], [2, -3], [2, -3, 4],\\ &\qquad\qquad [2, 4], [3], [3, -3], [3, -3, 4], [3, 4], [-3], [-3, 4], [4]])\\ &= \mathsf{reduce}\,(\uparrow_{\mathsf{length}})\,[[2], [2, -3], [3], [3, -3], [-3], [4]]\\ &= [2, -3]\end{aligned}$$

Sample fortress code for longest-ascending subsequence is shown below. The code computes the longest p-subsequence with its length.

$$x = \langle 2, 5, -3, 4 \,\rangle$$
$$las = \texttt{BIG MAX} \,\langle\, (length\ y, y) \mid y \leftarrow subs\ x, ascending\ y \,\rangle$$
$$lds = \texttt{BIG MAX} \,\langle\, (length\ y, y) \mid y \leftarrow subs\ x, descending\ y \,\rangle$$

3.6.2 0-1 Knapsack Problem

Example 11 (0-1 Knapsack Problem)**.** *Given a list of items (pairs of value and weight) and a knapsack of fixed capacity, find a subset of items that has the maximum sum of values and its sum of weight is less than or equals to the capacity.*

For example, given four items $[(2,1),(5,3),(1,1),(4,2)]$ (first element of a pair is value and the second is weight) and a knapsack of capacity 4, the solution (a set of items to be put into the knapsack) is $[(2,1),(5,3)]$. This solution is given as follows. First, we generate every combination of items by subs. Then, using filtering, we throw away combinations of which weight is greater than the capacity. Finally, we take the combination that have the maximum sum of values.

$$\begin{aligned}
&\mathsf{reduce}\,(\uparrow_{\mathsf{reduce}\,(+)\,\circ\,\mathsf{map}\,\pi_1})\,(\mathsf{filter}\,((\leq 4)\circ(\mathsf{reduce}\,(+))\circ(\mathsf{map}\,\pi_2))\,(\mathsf{subs}\,[(2,1),(5,3),(1,1),(4,2)]))\\
&= \mathsf{reduce}\,(\uparrow_{\mathsf{reduce}\,(+)\,\circ\,\mathsf{map}\,\pi_1})\,[[(2,1)],[(2,1),(5,3)],[(2,1),(1,1)],[(2,1),(1,1),(4,2)],\\
&\qquad\qquad [(2,1),(4,2)],[(5,3)],[(5,3),(1,1)],[(1,1)],[(1,1),(4,2)],[(4,2)]]\\
&= [(2,1),(5,3)]
\end{aligned}$$

Sample fortress code for 0-1 knapsack problem is shown below.

$$\begin{aligned}
&x = \langle (2,1),(5,3),(1,1),(4,2)\rangle\\
&capa = 4\\
&pf = \texttt{BIG MAX}\,\langle\,(\textstyle\sum\langle\, first\ a \mid a \leftarrow y\rangle, y) \mid y \leftarrow subs\ x, \sum\langle\,\mathrm{second}\ a \mid a \leftarrow y\rangle \leq capa\rangle
\end{aligned}$$

3.7 GGenerator parts

GGenerator parts generates all partitions of an input list. For example, applying parts to $[2,5,-3]$ we get the following nested list.

$$\mathsf{parts}\ [2,5,-3] = [[[2,5,-3]],[[2,5],[-3]],[[2],[5,-3]],[[2],[5],[-3]]]$$

3.7.1 Smallest All Ascending Partition

Example 12 (Smallest All Ascending Partition)**.** *Given a list, find the smallest partition in which each part is ascending.*

For example, smallest all ascending partition of $[2,5,-3]$ is $[[2,5],[-3]]$. This solution is obtained by using parts and filter as shown below.

$$\begin{aligned}
&\mathsf{reduce}\,(\downarrow_{\mathsf{length}})\,(\mathsf{filter}\,((\mathsf{reduce}\,(\wedge))\circ(\mathsf{map}\ ascending))\,(\mathsf{parts}\ [2,5,-3]))\\
&\mathsf{reduce}\,(\downarrow_{\mathsf{length}})\,(\mathsf{filter}\,((\mathsf{reduce}\,(\wedge))\circ(\mathsf{map}\ ascending))\,[[[2,5,-3]],[[2,5],[-3]],[[2],[5,-3]],\\
&\qquad\qquad [[2],[5],[-3]]])\\
&= \mathsf{reduce}\,(\downarrow_{\mathsf{length}})\ \ [[[2,5],[-3]],[[2],[5],[-3]]]\\
&= [[2,5],[-3]]
\end{aligned}$$

Sample fortress code for all ascending partition is shown below. The code computes shortest-partition with its length.

$$\begin{aligned}
&x = \langle 2,5,-3\rangle\\
&las = \texttt{BIG MIN}\,\langle\,(length\ y, y) \mid y \leftarrow parts\ x, \texttt{BIG}\wedge\langle\ ascending\ a \mid a \leftarrow y\,\rangle\,\rangle
\end{aligned}$$

3.7.2 Paragraph Formatting

Example 13 (Paragraph Formatting)**.** *Given a list of lengths of words and a length of a line, find the minimum partition of the list so that a sum of each part is less or equals to the given length of a line.*

For example, lengths of words are $[2,5,3,4]$ and the length of the line is 7, the minimum partition is $[[2,5],[3,4]]$, where sum of each part is equal to 7. This solution is obtained by using parts, filtering and reduction.

$$\begin{aligned}
&\mathsf{reduce}\,(\downarrow_{\mathsf{length}})\,(\mathsf{filter}\,((\leq 7)\circ(\mathsf{reduce}\,(+))\,(\mathsf{parts}\ [2,5,3,4]))\\
&= \mathsf{reduce}\,(\downarrow_{\mathsf{length}})\ \ [[[2,5],[3,4]],[[2,5],[3],[4]],[[2],[5],[3,4]],[[2],[5],[3],[4]]]\\
&= [[2,5],[3,4]]
\end{aligned}$$

Sample fortress code for paragraph formatting is shown below.

$x = \langle 2, 5, 3, 4 \rangle$
$line\_width = 7$
$pf = \texttt{BIG MIN} \langle\, (length\ y, y) \mid y \leftarrow parts\ x, \texttt{BIG} \wedge \langle \textstyle\sum a \leq line\_width \mid a \leftarrow y \,\rangle \,\rangle$

3.8 GGenerator perm

GGenerator perm generates all permutations of an input list. For example, applying perm to $[2, 5, -3]$ we get the following nested list.

$$\mathsf{perm}\ [2, 5, -3] = [[2, 5, -3], [2, -3, 5], [5, 2, -3], [5, -3, 2], [-3, 2, 5], [-3, 5, 2]]$$

3.8.1 Sorting

Sorting can be performed by using perm as follows.

$\mathsf{head}\ (\mathsf{filter}\ ascending\ (\mathsf{perm}\ [2, 5, -3]))$
$= \mathsf{head}\ (\mathsf{filter}\ ascending\ [[2, 5, -3], [2, -3, 5], [5, 2, -3], [5, -3, 2], [-3, 2, 5], [-3, 5, 2]])$
$= \mathsf{head}\ [[-3, 2, 5]]$
$= [-3, 2, 5]$

The program $\mathsf{head} \circ \mathsf{filter}\ ascending \circ \mathsf{perm}$ describes clearly the specification of sorting, although this program is, of course, extremely inefficient by naive execution.

Sample fortress code for the sorting is shown below.

$x = \langle 2, 5, -3 \rangle$
$sorted\_x = head\langle\, ys \mid ys \leftarrow perms\ x, ascending\ ys \,\rangle$

4 A Collection of Theories

Theories given in this section are used to dispatch efficient implementation against user programs of the generate-and-test specification. Preliminaries for formal discussion of theories are shown at the beginning of this section. The collection of theories of GGeneratorsfollows it.

Notation in this section follows that of Haskell [Bir98] and BMF (Bird-Meertens formalism) [Bir87, Ski90].

4.1 Preliminaries

This section gives formal definitions and concise notations of functions for formal manipulation of specifications. Intuitive definitions of those functions are found in Section 3.1, in which some of them have more human-readable names.

We assume that binary operators in this section, such as $\oplus$ and $\otimes$, are associative, unless otherwise noted. The identity of binary operator $\oplus$ is denoted by $\imath_\oplus$.

A list is a sequence of elements $a_1, a_2, \ldots, a_n$ denoted by $[a_1, a_2, \ldots, a_n]$. Function $[\cdot]$ takes an element to make a singleton list, i.e., $[\cdot]\ a = [a]$. Concatenation of two lists is denoted by binary operator $+\!\!+$. An empty list, which is the identity of $+\!\!+$, is denoted by $[\,]$.

Binary operators $\gg$ and $\ll$ used in the following sections are defined as follows.

$$a \gg b = b, \quad a \ll b = a$$

4.1.1 Homomorphism and Auxiliary Functions

Homomorphisms are basic functions defined on lists as follows.

$$\begin{array}{lcl} (\!|\oplus, f|\!)\ (x +\!\!+ y) & = & (\!|\oplus, f|\!)\ x \oplus (\!|\oplus, f|\!)\ y \\ (\!|\oplus, f|\!)\ [a] & = & f\ a \end{array}$$

Homomorphism $(\!|\oplus, f|\!)$ applies the given function f to every element of the input list, and then combines the results using the given associative binary operator $\oplus$. Since a homomorphism is completely specified by function f and associative binary operator $\oplus$, a homomorphism is denoted by $(\!|\oplus, f|\!)$. Homomorphism is the basis of parallel computation, since associativity of the operator $\oplus$ gives us correct balanced divide-and-conquer parallel computation of homomorphism.

Intuitive definition of homomorphism $(\!|\oplus, f|\!)$ is shown below.

$$(\!|\oplus, f|\!)\ [x_1, x_2, \ldots, x_n] \quad = \quad f\ x_1 \oplus f\ x_2 \oplus \cdots \oplus f\ x_n$$

Since $\oplus$ has associativity, the above result of the homomorphism can be obtained by the following computation. First, the input list $[x_1, x_2, \ldots, x_n]$ is divided into two parts $[x_1, x_2, \ldots, x_{n/2}]$ and $[x_{n/2+1}, x_{n/2+2}, \ldots, x_n]$. Next, the homomorphism $(\!|\oplus, f|\!)$ is applied to each of the parts to compute in parallel the partial results $f\ x_1 \oplus f\ x_2 \oplus \cdots \oplus f\ x_{n/2}$ and $f\ x_{n/2+1} \oplus f\ x_{n/2+2} \oplus \cdots \oplus f\ x_n$. Then, those partial results are combined by the operator $\oplus$ to get the result for the whole input. Repeatedly applying the above division and combination to computation of the partial results, we get balanced divide-and-conquer parallel computation of the homomorphism $(\!|\oplus, f|\!)$.

Two basic specializations of homomorphism are $f*$ and $\oplus/$ defined below.

$$\begin{array}{lcl} f* & = & (\!|+\!\!+, [\cdot] \circ f|\!) \\ \oplus/ & = & (\!|\oplus, \mathsf{id}|\!) \end{array}$$

Function $f*$ applies the given function f to every element of the given list. Function $\oplus/$ performs 'summation' of the given list with the given associative binary operator $\oplus$. One of the most

important relations between these specializations and homomorphism is the following equation, which shows that homomorphism $(\!|\oplus, f|\!)$ can be decomposed into $f*$ and $\oplus/$.

$$(\!|\oplus, f|\!) \quad = \quad \oplus/ \circ f*$$

Intuitive definitions of $f*$ and $\oplus/$ are shown below.

$$\begin{array}{lcl} f*\,[x_1, x_2, \ldots, x_n] & = & [f\ x_1, f\ x_2, \ldots, f\ x_n] \\ \oplus/\,[x_1, x_2, \ldots, x_n] & = & x_1 \oplus x_2 \oplus \cdots \oplus x_n \end{array}$$

Some functions are defined with homomorphism.

$$\begin{array}{lcll} \mathsf{last} & = & (\!|\gg, \mathsf{id}|\!) & \\ \mathsf{head} & = & (\!|\ll, \mathsf{id}|\!) & \\ p\triangleleft & = & (\!|+\!\!+, f|\!) & \textbf{where } f\ x = \textbf{if } p\ x \textbf{ then } [x] \textbf{ else } [\,] \end{array}$$

Function last takes the last element of the given list, while function head takes the head of the given list. Function $p\triangleleft$ filters the given list so that the resulting list contains only elements satisfying the given predicate p. Intuitive definitions of the above function are given as follows.

$$\begin{array}{lcl} \mathsf{last}\ [x_1, x_2, \ldots, x_n] & = & x_n \\ \mathsf{head}\ [x_1, x_2, \ldots, x_n] & = & x_1 \\ p\triangleleft [x_1, x_2, \ldots, x_n] & = & [x_{i_1}, x_{i_2}, \ldots, x_{i_m}] \ \textbf{where } p\ (x_{i_j}) = \mathsf{True} \end{array}$$

Correspondence between the above functions and those in Section 3.1 is shown below.

$$\mathsf{map}\, f = f*, \quad \mathsf{reduce}\,(\oplus) = \oplus/, \quad \mathsf{hom}\,(\oplus, f) = (\!|\oplus, f|\!), \quad \mathsf{filter}\, p = p\triangleleft$$

For a tuple of any length, projection functions π_n are defined as follows.

$$\pi_i\ (a_1, \ldots, a_i, \ldots, a_n) \quad = \quad a_i$$

For instance, $\pi_1\ (a, b) = a$ and $\pi_2\ (a, b) = b$.

4.1.2 GGenerators and Generate-and-test Specification

A GGenerator is a function to generate a nested list. For example, inits generates a nested list of which element is a prefix of a given list, as shown in Section 3.3 and Section 4.3.1.

We focus on problems described with generate-and-test specification. The general form of the generate-and-test specification is shown below.

$$(\!|\oplus, f|\!) \circ (\!|\otimes, g|\!)* \circ p\triangleleft \circ gg$$

Here, gg is one instance of GGenerators, which generates a nested list, and the generated nested list is consumed by the nested reductions, $(\!|\oplus, f|\!)$ for the outer list and $(\!|\otimes, g|\!)$ for the inner lists, after filtered with predicate p.

For example, simple nested reductions $\oplus/ \circ \otimes/*$ for a GGenerator gg can be described in the generate-and-test specification as follows.

$$\oplus/ \circ \otimes/* \circ gg = (\!|\oplus, \mathsf{id}|\!) \circ (\!|\otimes, \mathsf{id}|\!)* \circ \mathit{true}\triangleleft \circ gg$$

Here, *true* is the predicate returning always True. The equality of the left hand side and the right hand side is easily shown. First, by the definition of $\oplus/$, we get $(\!|\oplus, \mathsf{id}|\!) = \oplus/$ and $(\!|\otimes, \mathsf{id}|\!) = \otimes/$. Next, by the definition of $p\triangleleft$, $\mathit{true}\triangleleft = \mathsf{id}$ since *true* returns always True.

Theories shown in the following sections give efficient implementations of subclasses of the generate-and-test specification.

Related implementation on GG library is found in Section 5.1.1.

4.2 Properties on Reduction Operators and Predicates

This section defines properties on operators and predicates for deriving efficient implementation.

4.2.1 Properties on Reduction Operators

Associativity is the basis of parallel computation of homomorphism since it guarantees correctness of balanced divide-and-conquer parallel computation of homomorphism.

Definition 1 (Associativity). *Binary operator $\oplus$ is said to be associative if the following equation holds for all a, b and c.*

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c$$

Distributivity plays an important role in deriving efficient implementation for nested reductions. Basically, distributivity guarantees efficient reuse of partial results.

Definition 2 (Left-Distributivity). *Binary operator $\otimes$ is said to be left-distributive over $\oplus$ if the following equation holds for all a, b and c.*

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$

Definition 3 (Right-Distributivity). *Binary operator $\otimes$ is said to be right-distributive over $\oplus$ if the following equation holds for all a, b and c.*

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

Definition 4 (Distributivity). *Binary operator $\otimes$ is said to be distributive over $\oplus$ if $\otimes$ is left- and right-distributive over $\oplus$, i.e. the following equations hold for all a, b and c.*

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$
$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

Commutativity is used to change the order of computation to achieve good efficiency, as well as distributivity.

Definition 5 (Commutativity). *Binary operator $\oplus$ is said to be commutative if the following equation holds for all a and b.*

$$a \oplus b = b \oplus a$$

For example, well-known binary operators + (plus), $\times$ (times), $\uparrow$ (maximum), and $\downarrow$ (minimum) are all associative and commutative. Moreover, $\times$ distributes over +, and + distributes over $\uparrow$ or $\downarrow$. Concatenation operator $+\!\!+$ is associative, but not commutative. Function $f*$ is left-distributive over $+\!\!+$, since $f*(x +\!\!+ y) = (f*x) +\!\!+ (f*y)$. Distributivity of $f*$ is often used in the following derivations of efficient implementations.

Related implementation on GG library is found in Section 5.2.1.

4.2.2 Properties on Predicates

The following closure properties are defined on predicates for deriving efficient implementation [Zan92, Jeu93].

Definition 6 (Prefix-Closed). *Predicate p is said to be prefix-closed if the following equation holds for all x and y.*

$$p\ (x +\!\!+ y) \Rightarrow p\ (x)$$

Definition 7 (Suffix-Closed)**.** *Predicate p is said to be suffix-closed if the following equation holds for all x and y.*

$$p\ (x +\!\!+ y) \Rightarrow p\ (y)$$

Definition 8 (Segment-Closed)**.** *Predicate p is said to be segment-closed if p is prefix- and suffix closed, i.e. the following equation holds for all x, y and z.*

$$p\ (x +\!\!+ y +\!\!+ z) \Rightarrow p\ (y)$$

Definition 9 (Overlap-Closed)**.** *Predicate p is said to be overlap-closed if the following equation holds for all x, y and z.*

$$p\ (x +\!\!+ y) \wedge p\ (y +\!\!+ z) \wedge y \neq [\,] \Rightarrow p\ (x +\!\!+ y +\!\!+ z)$$

Prefix-closed property plays the most important role in derivation of efficient *sequential* implementation of segment problems [Zan92, Jeu93]. Other properties are used as auxiliary tools to improve the derived implementation. However, in derivation of efficient *parallel* implementation, the pair of segment-closed property and overlap-closed property plays the most important role, as shown in the following sections.

A candidate of segment-closed and overlap-closed predicates is given by a relation [Zan92].

Definition 10 (Relational Predicate)**.** *Given a relation R, relational predicate p_R is defined as follows.*

$$p_R\ (x) = \bigwedge\{aRb \mid [a,b] \in \mathit{segments}\ x\}$$

Here, *segments* x returns a set of segments (contiguous subsequences) of x, i.e., $\mathit{segments}\ x = \{\ y \mid u +\!\!+ y +\!\!+ v = x\ \}$. Note that $[a,b] \in \mathit{segments}\ x$ means that a and b are successive elements in x, since *segments* x generates all contiguous subsequences of x and $[a,b] \in \mathit{segments}\ x$ takes such subsequences of length two. Relational predicate p_R is true for the given list x, if all successive elements a and b in x satisfy the relation R, i.e. $aRb = \mathsf{True}$.

Zantema [Zan92] shows that a relational predicate p_R is segment-closed and overlap-closed, but we can also show the converse, i.e., a segment-closed and overlap-closed predicate is a relational predicate. The following lemma shows the relation between relational predicates and segment-closed and overlap-closed predicates.

Lemma 11 (Relational Predicate)**.** *Given predicate p that is true for all singletons and empty list, the following statements are equivalent.*

1. *p is segment-closed and overlap-closed.*
2. *p is relational.*

Proof. $2 \Rightarrow 1$) Since p is relational, there exists a relation R and the following equation holds.

$$p\ (x) = \bigwedge\{aRb \mid [a,b] \in \mathit{segments}\ x\}$$

First, we show that p is segment-closed.

$$\begin{array}{ll} & p\ (x +\!\!+ y +\!\!+ z) \\ = & \quad \{\ \text{unfolding } p\ \} \\ & \bigwedge\{aRb \mid [a,b] \in \mathit{segments}\ (x +\!\!+ y +\!\!+ z)\} \\ \Rightarrow & \quad \{\ \ \mathit{segments}\ y \subseteq \mathit{segments}\ (x +\!\!+ y +\!\!+ z)\ \} \\ & \bigwedge\{aRb \mid [a,b] \in \mathit{segments}\ y\} \\ = & \quad \{\ \text{folding } p\ \} \\ & p\ (y) \end{array}$$

Next, we show that p is overlap-closed.

$$
\begin{array}{ll}
 & p\ (x +\!\!+ y) \wedge p\ (y +\!\!+ z) \wedge y \neq [\,] \\
= & \quad \{\ \text{unfolding } p\ \} \\
 & \bigwedge\{aRb \,|\, [a,b] \in \mathit{segments}\ (x +\!\!+ y)\} \wedge \bigwedge\{aRb \,|\, [a,b] \in \mathit{segments}\ (y +\!\!+ z)\} \wedge y \neq [\,] \\
\Rightarrow & \left\{ \begin{array}{l} y \neq [\,] \Rightarrow \\ \quad \{[a,b] \,|\, [a,b] \in \mathit{segments}\ (x +\!\!+ y)\} \cup \{[a,b] \,|\, [a,b] \in \mathit{segments}\ (y +\!\!+ z)\} \\ \quad\quad = \{[a,b] \,|\, [a,b] \in \mathit{segments}\ (x +\!\!+ y +\!\!+ z)\} \end{array} \right\} \\
 & \bigwedge\{aRb \,|\, [a,b] \in \mathit{segments}\ (x +\!\!+ y +\!\!+ z)\} \\
= & \quad \{\ \text{folding } p\ \} \\
 & p\ (x +\!\!+ y +\!\!+ z)
\end{array}
$$

$1 \Rightarrow 2)$ Letting $R = \{(a,b) \,|\, p\ ([a,b])\}$, we show $p = p_R$ by induction.
For base cases, we have $p\ ([\,]) = p\ ([a]) = p_R\ ([]) = p_R\ ([a]) = \mathsf{True}$ by assumption.
For induction case, we have $p\ ([a] +\!\!+ x) = p_R\ ([a] +\!\!+ x)$ by the following calculation.

$$
\begin{array}{ll}
 & p\ ([a] +\!\!+ x) \\
= & \quad \{\ \text{segment-closed and overlap-closed}\ \} \\
 & p\ (x) \wedge p\ ([a] +\!\!+ [\mathsf{head}\ x]) \\
= & \quad \{\ \text{induction hypothesis and definition of } R\ \} \\
 & p_R\ (x) \wedge aR(\mathsf{head}\ x) \\
= & \quad \{\ \text{definition of } p_R\ \} \\
 & p_R\ ([a] +\!\!+ x)
\end{array}
$$

Thus, $p = p_R$ and p is relational. □

In the above lemma, we assumed that the predicate p is true for all singletons and empty list for simplicity. However, we can remove this assumption by letting values of relational predicate p_R in the proof be those of the given predicate p.

Examples of relational predicates are shown below.

$$
\begin{array}{lcl}
\mathit{ascending}\ (x) & = & p_<\ (x) \\
\mathit{descending}\ (x) & = & p_>\ (x) \\
\mathit{flat}\ (x) & = & p_=\ (x) \\
\mathit{smooth}_c\ (x) & = & p_{R_c}\ (x) \quad \textbf{where } aR_cb = |a-b| \leq c
\end{array}
$$

Predicate *ascending* is true when the given list is ascendingly sorted, while *descending* is true for descendingly sorted lists. Predicate *flat* is true if the all elements in the given list are the same. Predicate smooth_c is true if the maximum of differences of successive elements is less than or equal to c. Especially, $\mathit{flat} = \mathit{smooth}_0$.

It is worth mentioning about composition of predicates [Zan92]. Each closure property of prefix-closed, suffix-closed, segment-closed and overlap-closed is closed under disjunction. Each closure property of prefix-closed, suffix-closed, and segment-closed is closed also under conjunction, but overlap-closed property is not closed under conjunction. For example, *ascending* and *descending* are both overlap-closed, but $\mathit{ascending} \vee \mathit{descending}$ is not overlap-closed since we can make a counterexample: $\mathit{ascending}(x +\!\!+ y) \wedge \mathit{descending}(y +\!\!+ z) \wedge y \neq [\,]$ implies neither $\mathit{ascending}(x +\!\!+ y +\!\!+ z)$ nor $\mathit{descending}(x +\!\!+ y +\!\!+ z)$.

Related implementation on GG library is found in Section 5.2.3. Related examples are found in Section 3.1.3.

4.3 Theory of inits

This section gives formal definition and theory of GGenerator inits.

4.3.1 Formal Definition and Basic Lemmas

First of all, we give a formal definition of inits.

Definition 12 (GGenerator Inits)**.** *GGenerator* inits *is defined with homomorphism as follows.*

$$\mathsf{inits} = (\!|\oplus, [\cdot] \circ [\cdot]|\!) \quad \textbf{where } x \oplus y = x +\!\!+ ((\mathsf{last}\ x) +\!\!+) * y$$

The operator $\oplus$ in the above definition makes a list of initial segments of a list $u +\!\!+ v$ from lists (x and y in the above equation) of initial segments of u and v. Since each initial segment of u is also an initial segment of $u +\!\!+ v$, x remains in the result. Since each initial segment of v need to be concatenated with u to become an initial segment of $u +\!\!+ v$, the operator maps $((\mathsf{last}\ x) +\!\!+)$ (u is the last element of x) to y.

The following two lemmas are well-known lemmas of inits [Bir87].

Lemma 13 (Scan)**.** *For any associative binary operator $\oplus$, the following equation holds.*

$$\mathsf{scan}\ (\oplus) = \oplus/ * \circ \mathsf{inits}$$

Lemma 14 (Inits-Map Promotion)**.** *For function f, the following equation holds.*

$$f** \circ \mathsf{inits} = \mathsf{inits} \circ f*$$

First lemma gives us a way to compute prefix sums by inits. The second lemma gives us a way to promote the application of function f through inits. The number of applications of function f on the left hand side is $n(n+1)/2$, while that on the right hand side is n. So, transformation from the left hand side to the right hand side improves the efficiency. Also, the lemma enables us to ignore a function g in the generate-and-test specification $(\!|\oplus, f|\!) \circ (\!|\otimes, g|\!) * \circ p \triangleleft \circ \mathsf{inits}$, since we can replace g with the identity function id and apply g to the input of inits beforehand (of course, we need some tricks to through $p\triangleleft$).

A related example problem is found in Section 3.3.1. Related implementation on GG library is found in Section 5.3.1.

4.3.2 Theorem for Reduction with Distributive Operators

The following theorem gives efficient parallel implementation of nested reductions for inits when two reductions have distributivity. One of the most famous problems for which this theorem is applicable is maximum initial-segment sum problem (also known as maximum prefix sum problem), which is an instance of maximum marking problems [SHTO00, Bir01]. A related example problem is found in Section 3.3.2. Related implementation on GG library is found in Section 5.3.2.

Theorem 15 (Maximum Initial-segment Sum)**.** *Provided that $\oplus$ is associative, and $\otimes$ is associative and left-distributive over $\oplus$, the following equation holds.*

$$\oplus/ \circ \otimes/ * \circ \mathsf{inits} \quad = \quad \pi_1 \circ (\!|\odot, \mathit{pair}|\!) \quad \textbf{where} \quad \begin{array}{l}(i_1, s_1) \odot (i_2, s_2) = (i_1 \oplus (s_1 \otimes i_2), s_1 \otimes s_2) \\ \mathit{pair}\ a = (a, a)\end{array}$$

Proof. We show the theorem by induction.

For base case, we have $\mathit{LHS}\ [a] = \mathit{RHS}\ [a]$ by the following calculation.

$$\begin{array}{ll} & \mathit{LHS}\ [a] \\ = & \quad \{\ \text{LHS}\ \} \\ & (\oplus/ \circ \otimes/ * \circ \mathsf{inits})\ [a] \\ = & \quad \{\ \text{definition of inits, } *, /\ \} \\ & a \\ = & \quad \{\ \text{definition of } \pi_1, (\!|,|\!), \mathit{pair}\ \} \\ & (\pi_1 \circ (\!|\odot, \mathit{pair}|\!))\ [a] \\ = & \quad \{\ \text{RHS}\ \} \\ & \mathit{RHS}\ [a] \end{array}$$

For induction case, we have $LHS\ (x +\!\!+ y) = RHS\ (x +\!\!+ y)$ by the following calculation.

$$
\begin{array}{ll}
 & LHS\ (x +\!\!+ y) \\
= & \quad \{\ \text{LHS}\ \} \\
 & (\oplus/ \circ \otimes/* \circ \mathsf{inits})\ (x +\!\!+ y) \\
= & \quad \{\ \text{definition of function composition, unfolding } \mathsf{inits}, \mathsf{last}\ (\mathsf{inits}\ x) = x\ \} \\
 & (\oplus/ \circ \otimes/*)\ (\mathsf{inits}\ x +\!\!+ (x +\!\!+)*(\mathsf{inits}\ y)) \\
= & \quad \{\ \text{definition of function composition, } * \text{ and } /\ \} \\
 & \oplus/\ (\otimes/*\ (\mathsf{inits}\ x)) \oplus \oplus/\ ((\otimes/\ x \otimes)*(\otimes/*\ (\mathsf{inits}\ y))) \\
= & \quad \{\ \text{distributivity of } \otimes\ \} \\
 & \oplus/\ (\otimes/*\ (\mathsf{inits}\ x)) \oplus (\otimes/\ x \otimes \oplus/\ (\otimes/*\ (\mathsf{inits}\ y))) \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
 & (\pi_1 \circ (\!|\odot, pair|\!))\ x \oplus (\otimes/\ x \otimes (\pi_1 \circ (\!|\odot, pair|\!))\ y) \\
= & \quad \{\ \otimes/\ x = \pi_2\ ((\!|\odot, pair|\!)\ x)\ \text{(shown below)}\ \} \\
 & (\pi_1 \circ (\!|\odot, pair|\!))\ x \oplus ((\pi_2\ ((\!|\odot, pair|\!)\ x)) \otimes (\pi_1 \circ (\!|\odot, pair|\!))\ y) \\
= & \quad \{\ \text{definition of } \odot\ \} \\
 & \pi_1\ ((\!|\odot, pair|\!)\ x \odot (\!|\odot, pair|\!)\ y) \\
= & \quad \{\ \text{definition of } (\!|,|\!), \text{function composition}\ \} \\
 & (\pi_1 \circ (\!|\odot, pair|\!))\ (x +\!\!+ y) \\
= & \quad \{\ \text{RHS}\ \} \\
 & RHS\ (x +\!\!+ y)
\end{array}
$$

Finally, we show that $\otimes/\ x = \pi_2\ ((\!|\odot, pair|\!)\ x)$. For base case, we have

$$
\begin{array}{ll}
 & LHS\ [a] \\
= & \quad \{\ \text{LHS}\ \} \\
 & \otimes/[a] \\
= & \quad \{\ \text{definition of } /\ \} \\
 & a \\
= & \quad \{\ \text{definition of } \pi_2,\ (\!|,|\!),\ pair\ \} \\
 & \pi_2\ ((\!|\odot, pair|\!)\ [a]) \\
= & \quad \{\ \text{RHS}\ \} \\
 & RHS\ [a]
\end{array}
$$

For induction case, we have

$$
\begin{array}{ll}
 & LHS\ (x +\!\!+ y) \\
= & \quad \{\ \text{LHS}\ \} \\
 & \otimes/\ (x +\!\!+ y) \\
= & \quad \{\ \text{definition of } /\ \} \\
 & \otimes/\ x \otimes \otimes/\ y \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
 & \pi_2\ ((\!|\odot, pair|\!)\ x) \otimes \pi_2\ ((\!|\odot, pair|\!)\ y) \\
= & \quad \{\ \text{definition of } \odot\ \} \\
 & \pi_2\ ((\!|\odot, pair|\!)\ x \odot (\!|\odot, pair|\!)\ y) \\
= & \quad \{\ \text{definition of } (\!|,|\!)\ \} \\
 & \pi_2\ ((\!|\odot, pair|\!)\ (x +\!\!+ y)) \\
= & \quad \{\ \text{RHS}\ \} \\
 & RHS\ (x +\!\!+ y)
\end{array}
$$

□

Similar proof is also shown in [Gor97].

The resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\odot$, while the original program (the left hand side) uses two nested reductions with inits. The cost of the new operator is proportional to the cost of operators in the original reductions. So, the resulting program is more efficient than the original program.

The new operator $\odot$ is applied to tuples. The first element of a tuple is equal to the result of the original program. The second element of the tuple is equal to the reduction of the same input with operator $\otimes$, which is used to improve efficiency of the program by reusing the partial results effectively.

4.3.3 Theorem for Filtering with Relational Predicates

The following theorem extends Theorem 15 by allowing filtering with the given predicate. It gives efficient parallel implementation of nested reductions for inits when two reductions have distributivity and the predicate is relational. Derivation of efficient sequential implementations for those programs with filtering is shown in [Zan92, Jeu93]. A related example problem is found in Section 3.3.3. Related implementation on GG library is found in Section 5.3.3.

Theorem 16 (Maximum p-Initial-segment Sum)**.** *Provided that $\oplus$ is associative, $\otimes$ is associative and left-distributive over $\oplus$, and predicate p is relational, the following equation holds.*

$$
\begin{array}{l}
\oplus/ \circ \otimes/* \circ p\triangleleft \circ \mathsf{inits} = \pi_1 \circ (\!|\boxdot, \mathit{pentuple}|\!) \\
\quad \textbf{where} \\
\quad (i_1, s_1, h_1, l_1, p_1) \boxdot (i_2, s_2, h_2, l_2, p_2) = (i, s_1 \otimes s_2, h_1 \ll h_2, l_1 \gg l_2, p_1 \wedge p_2 \wedge p\ ([l_1, h_2])) \\
\quad\quad \textbf{where } i = i_1 \oplus \textbf{if } p_1 \wedge p\ ([l_1, h_2]) \textbf{ then } s_1 \otimes i_2 \textbf{ else } \imath_\oplus \\
\quad \mathit{pentuple}\ a = (a, a, a, a, T)
\end{array}
$$

Proof. We show the theorem by induction.

For base case, we have

$$
\begin{array}{ll}
 & \mathit{LHS}\ [a] \\
= & \quad \{\ \text{LHS}\ \} \\
 & (\oplus/ \circ \otimes/* \circ p\triangleleft \circ \mathsf{inits})\ [a] \\
= & \quad \{\ \text{definition of inits, } *, /\text{ and } \triangleleft, p \text{ is true for singleton}\ \} \\
 & a \\
= & \quad \{\ \text{definition of } \pi_1, (\!|,|\!), \text{ and } \mathit{pentuple}\ \} \\
 & (\pi_1 \circ (\!|\boxdot, \mathit{pentuple}|\!))\ [a] \\
= & \quad \{\ \text{RHS}\ \} \\
 & \mathit{RHS}\ [a]
\end{array}
$$

For induction case, we have the following calculation.

$$
\begin{array}{ll}
 & \mathit{LHS}\ (x +\!\!+ y) \\
= & \quad \{\ \text{LHS}\ \} \\
 & (\oplus/ \circ \otimes/* \circ p\triangleleft \circ \mathsf{inits})\ (x +\!\!+ y) \\
= & \quad \{\ \text{definition of inits}\ \} \\
 & (\oplus/ \circ \otimes/* \circ p\triangleleft)\ (\mathsf{inits}\ x +\!\!+ ((x+\!\!+)*(\mathsf{inits}\ y))) \\
= & \quad \{\ \text{definition of } \triangleleft, /, \text{ and } *\ \} \\
 & (\oplus/ \circ \otimes/* \circ p\triangleleft \circ \mathsf{inits})\ x \oplus (\oplus/ \circ \otimes/* \circ p\triangleleft)\ ((x+\!\!+)*(\mathsf{inits}\ y))
\end{array}
$$

To proceed more, we calculate a part of the above equation $p\triangleleft((x+\!\!+)*(\mathsf{inits}\ y))$ as follows.

$$
\begin{array}{ll}
 & p\triangleleft((x+\!\!+)*(\mathsf{inits}\ y)) \\
= & \quad \{\ p \text{ is relational: } p\ (x +\!\!+ y) \Rightarrow p\ (y)\ \} \\
 & p\triangleleft((x+\!\!+)*(p\triangleleft(\mathsf{inits}\ y))) \\
= & \quad \{\ p \text{ is relational: } p\ (x +\!\!+ y) = p\ (y) \wedge p\ (x) \wedge p\ ([\mathsf{last}\ x, \mathsf{head}\ y])\ \} \\
 & \textbf{if } p\ (x) \wedge p\ ([\mathsf{last}\ x, \mathsf{head}\ y]) \textbf{ then } (x+\!\!+)*(p\triangleleft(\mathsf{inits}\ y)) \textbf{ else } [\,]
\end{array}
$$

Using this result, we proceed a wider part of the above equation $(\oplus/ \circ \otimes/* \circ p\triangleleft)\ ((x +\!\!+)* (\mathsf{inits}\ y))$ as follows.

$$
\begin{array}{ll}
 & (\oplus/ \circ \otimes/* \circ p\triangleleft)\ ((x +\!\!+)* (\mathsf{inits}\ y)) \\
= & \quad \{\ \text{above calculation}\ \} \\
 & (\oplus/ \circ \otimes/*)\ (\mathbf{if}\ p\ (x) \wedge p\ ([\mathsf{last}\ x, \mathsf{head}\ y])\ \mathbf{then}\ (x +\!\!+)* (p\triangleleft (\mathsf{inits}\ y))\ \mathbf{else}\ [\,]) \\
= & \quad \{\ \text{distributivity of if-then-else, definition of } * \text{ and } /\ \} \\
 & \mathbf{if}\ p\ (x) \wedge p\ ([\mathsf{last}\ x, \mathsf{head}\ y])\ \mathbf{then}\ \oplus/\ ((\otimes/ x \otimes)* (\otimes/* (p\triangleleft (\mathsf{inits}\ y))))\ \mathbf{else}\ \imath_\oplus \\
= & \quad \{\ \text{distributivity of } \otimes\ \} \\
 & \mathbf{if}\ p\ (x) \wedge p\ ([\mathsf{last}\ x, \mathsf{head}\ y])\ \mathbf{then}\ \otimes/ x \otimes \oplus/\ (\otimes/* (p\triangleleft (\mathsf{inits}\ y)))\ \mathbf{else}\ \imath_\oplus
\end{array}
$$

Now, we resume the suspended calculation for induction case.

$$
\begin{array}{ll}
 & \mathit{LHS}\ (x +\!\!+ y) \\
= & \quad \{\ \text{resume}\ \} \\
 & (\oplus/ \circ \otimes/* \circ p\triangleleft \circ \mathsf{inits})\ x \oplus (\oplus/ \circ \otimes/* \circ p\triangleleft)\ ((x +\!\!+)* (\mathsf{inits}\ y)) \\
= & \quad \{\ \text{above calculation}\ \} \\
 & (\oplus/ \circ \otimes/* \circ p\triangleleft \circ \mathsf{inits})\ x \\
 & \quad \otimes\ \mathbf{if}\ p\ (x) \wedge p\ ([\mathsf{last}\ x, \mathsf{head}\ y])\ \mathbf{then}\ \otimes/ x \otimes \oplus/\ (\otimes/* (p\triangleleft (\mathsf{inits}\ y)))\ \mathbf{else}\ \imath_\oplus \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
 & (\pi_1 \circ (\!|[\boxdot, \mathit{pentuple}]|\!))\ x \\
 & \quad \otimes\ \mathbf{if}\ p\ (x) \wedge p\ ([\mathsf{last}\ x, \mathsf{head}\ y])\ \mathbf{then}\ \otimes/ x \otimes (\pi_1 \circ (\!|[\boxdot, \mathit{pentuple}]|\!))\ y\ \mathbf{else}\ \imath_\oplus \\
= & \quad \{\ (\_, \otimes/ x, \mathsf{head}\ x, \mathsf{last}\ x, p\ (x)) = (\!|[\boxdot, \mathit{pentuple}]|\!)\ x\ \text{(shown below)}\ \} \\
 & (\pi_1 \circ (\!|[\boxdot, \mathit{pentuple}]|\!))\ x \\
 & \quad \otimes\ \mathbf{if}\ (\pi_5 \circ (\!|[\boxdot, \mathit{pentuple}]|\!))\ x \wedge p\ ([(\pi_4 \circ (\!|[\boxdot, \mathit{pentuple}]|\!))\ x, (\pi_3 \circ (\!|[\boxdot, \mathit{pentuple}]|\!))\ y]) \\
 & \quad\ \mathbf{then}\ (\pi_2 \circ (\!|[\boxdot, \mathit{pentuple}]|\!))\ x \otimes (\pi_1 \circ (\!|[\boxdot, \mathit{pentuple}]|\!))\ y\ \mathbf{else}\ \imath_\oplus \\
= & \quad \{\ \text{definition of } \boxdot\ \} \\
 & (\pi_1 \circ (\!|[\boxdot, \mathit{pentuple}]|\!))\ (x +\!\!+ y) \\
= & \quad \{\ \text{RHS}\ \} \\
 & \mathit{RHS}\ (x +\!\!+ y)
\end{array}
$$

Finally, we show $(\_, \otimes/ x, \mathsf{head}\ x, \mathsf{last}\ x, p\ (x)) = (\!|[\boxdot, \mathit{pentuple}]|\!)\ x$. For base case, we have

$$
\begin{array}{ll}
 & (\_, \otimes/ [a], \mathsf{head}\ [a], \mathsf{last}\ [a], p\ ([a])) \\
= & \quad \{\ \text{definition of each function, } p \text{ is relational}\ \} \\
 & (\_, a, a, a, T) \\
= & \quad \{\ \text{definition of } (\!|,|\!) \text{ and } \mathit{pentuple}\ \} \\
 & (\!|[\boxdot, \mathit{pentuple}]|\!)\ [a]
\end{array}
$$

For induction case, we have

$$
\begin{array}{ll}
 & (\_, \otimes/ (x +\!\!+ y), \mathsf{head}\ (x +\!\!+ y), \mathsf{last}\ (x +\!\!+ y), p\ (x +\!\!+ y)) \\
= & \quad \{\ \text{definition of each function, } p \text{ is relational}\ \} \\
 & (\_, \otimes/ x \otimes \otimes/ y, \mathsf{head}\ x \ll \mathsf{head}\ y, \mathsf{last}\ x \gg \mathsf{head}\ y, p\ x \wedge p\ y \wedge p\ ([\mathsf{head}\ x, \mathsf{last}\ y])) \\
= & \quad \{\ \text{definition of } \boxdot\ \} \\
 & (\_, \otimes/ x, \mathsf{head}\ x, \mathsf{last}\ x, p\ x) \boxdot (\_, \otimes/ y, \mathsf{head}\ y, \mathsf{last}\ y, p\ y) \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
 & (\!|[\boxdot, \mathit{pentuple}]|\!)\ x \boxdot (\!|[\boxdot, \mathit{pentuple}]|\!)\ y \\
= & \quad \{\ \text{definition of } (\!|,|\!)\ \} \\
 & (\!|[\boxdot, \mathit{pentuple}]|\!)\ (x +\!\!+ y)
\end{array}
$$

Thus, we have $(\_, \otimes/ x, \mathsf{head}\ x, \mathsf{last}\ x, p\ (x)) = (\!|[\boxdot, \mathit{pentuple}]|\!)\ x$. □

The resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\boxdot$, while the original program (the left hand side) uses two nested reductions with inits. The cost of the new operator is proportional to the cost of operators in the original reductions and the application of p. So, the resulting program is more efficient than the original program.

The new operator $\boxdot$ is applied to pentuples. The first element of a pentuple is equal to the result of the original program. The second element of the pentuple is equal to the reduction of the same input with operator $\otimes$, which is used to improve efficiency of the program by reusing the partial results effectively. The third and forth elements are the edge elements of the input. Those edge elements are used to check whether results from two recursions in divide-and-conquer computation can be connected to make a better solution. Since the predicate p is relational, we can check the connectability by using only elements on the edge. The fifth element is a Boolean value that is the result of p applied to the input.

Here, we want to reduce the size of pentuples by eliminating the fifth element (it corresponds to $p\ (x)$) for simplicity. To do so, we use an assumption that $\imath_\otimes$ is the zero of $\otimes$, i.e. $\imath_\oplus \otimes a = \imath_\oplus$.

Corollary 17 (Maximum p-Initial-segment Sum (Simplified))**.** *Provided that $\oplus$ is associative, $\otimes$ is associative and left-distributive over $\oplus$, the identity $\imath_\oplus$ is the zero of $\otimes$, and predicate p is relational, the following equation holds.*

$$
\begin{aligned}
&\oplus/ \circ \otimes/* \circ p\triangleleft \circ \mathsf{inits} = \pi_1 \circ (\!|\boxtimes, \mathit{quadruple}|\!)\\
&\quad \textbf{where}\\
&\qquad (i_1, s_1, h_1, l_1) \boxtimes (i_2, s_2, h_2, l_2) = (i_1 \oplus (s_1 \otimes i_2)_{l_1,h_2}, (s_1 \otimes s_2)_{l_1,h_2}, h_1 \ll h_2, l_1 \gg l_2)\\
&\qquad \mathit{quadruple}\ a = (a, a, a, a)\\
&\qquad (a)_{l,h} = \textbf{if}\ p\ ([l, h])\ \textbf{then}\ a\ \textbf{else}\ \imath_\oplus
\end{aligned}
$$

Proof. To simplify the result of the theorem, we add an invariant to the result of Theorem 16. The invariant added to the pentuple (i, s, h, t, p) is $\neg p \Rightarrow s = \imath_\oplus$. In the following calculation, we derive a operator slightly changed from that of Theorem 16.

$$
\begin{aligned}
& i\\
=\quad & \{\ \text{definition}\ \}\\
& i_1 \oplus \textbf{if}\ p_1 \wedge p\ ([l_1, h_2])\ \textbf{then}\ s_1 \otimes i_2\ \textbf{else}\ \imath_\oplus\\
=\quad & \{\ \text{splitting condition}\ \}\\
& i_1 \oplus \textbf{if}\ p\ ([l_1, h_2])\ \textbf{then}\ (\textbf{if}\ p_1\ \textbf{then}\ (s_1 \otimes i_2)\ \textbf{else}\ \imath_\oplus)\ \textbf{else}\ \imath_\oplus\\
=\quad & \{\ \text{assumption: } \imath_\oplus \text{ is the zero}\ \}\\
& i_1 \oplus \textbf{if}\ p\ ([l_1, h_2])\ \textbf{then}\ ((\textbf{if}\ p_1\ \textbf{then}\ s_1\ \textbf{else}\ \imath_\oplus) \otimes i_2)\ \textbf{else}\ \imath_\oplus\\
=\quad & \{\ s_1' = \textbf{if}\ p_1\ \textbf{then}\ s_1\ \textbf{else}\ \imath_\oplus\ \}\\
& i_1 \oplus \textbf{if}\ p\ ([l_1, h_2])\ \textbf{then}\ (s_1' \otimes i_2)\ \textbf{else}\ \imath_\oplus\\
=\quad & \{\ \text{definition of } (a)_{t,h}\ \}\\
& i_1 \oplus (s_1' \otimes i_2)_{l_1,h_2}
\end{aligned}
$$

Computation of $s' = \textbf{if}\ p\ \textbf{then}\ s\ \textbf{else}\ \imath_\oplus$ is as follows.

$$
\begin{aligned}
& s'\\
=\quad & \{\ \text{definition}\ \}\\
& \textbf{if}\ p\ \textbf{then}\ s\ \textbf{else}\ \imath_\oplus\\
=\quad & \{\ \text{computation of } s \text{ and } p\ \}\\
& \textbf{if}\ p_1 \wedge p_2 \wedge p\ ([l_1, h_2])\ \textbf{then}\ s_1 \otimes s_2\ \textbf{else}\ \imath_\oplus\\
=\quad & \{\ \text{splitting condition}\ \}\\
& \textbf{if}\ p\ ([l_1, h_2])\ \textbf{then}\ (\textbf{if}\ p_1\ \textbf{then}\ s_1\ \textbf{else}\ \imath_\oplus) \otimes (\textbf{if}\ p_1\ \textbf{then}\ s_2\ \textbf{else}\ \imath_\oplus)\ \textbf{else}\ \imath_\oplus\\
=\quad & \{\ \text{definition of s'}\ \}\\
& \textbf{if}\ p\ ([l_1, h_2])\ \textbf{then}\ s_1' \otimes s_2'\ \textbf{else}\ \imath_\oplus\\
=\quad & \{\ \text{definition of } (a)_{t,h}\ \}\\
& (s_1' \otimes s_2')_{l_1,h_2}
\end{aligned}
$$

Now, we can use s' instead of s and p in the pentuple (i, s, h, t, p), since p and s are not used by computation of other parts. Thus, replacing s and p in the pentuple with s' and rename s' as s, we get the reduced operator $\boxtimes$. □

The resulting program (the right hand side) uses the new reduction operator $\boxtimes$ that is applied on quadruples. The difference from the result of Theorem 16 is that the second element becomes the identity $\imath_\oplus$ of $\oplus$ when the input does not satisfy the predicate p.

A sufficient condition of the assumption that $\imath_\oplus$ is the zero of $\otimes$ is that $\otimes$ distributes over $\oplus$. So, we will use this condition in implementation of the library.

4.4 Theory of tails

This section gives formal definition and theory of GGenerator tails.

4.4.1 Formal Definition and Basic Lemmas

First of all, we give a formal definition of tails.

Definition 18 (GGenerator Tails)**.** *GGenerator* tails *is defined with homomorphism as follows.*

$$\mathsf{tails} = (\!|\oplus, [\cdot] \circ [\cdot]|\!) \quad \textbf{where } x \oplus y = (+\!\!+(\mathsf{head}\ y))* x +\!\!+ y$$

The operator $\oplus$ in the above definition makes a list of tail segments of a list $u +\!\!+ v$ from lists (x and y in the above equation) of tail segments of u and v. Since each tail segment of v is also a tail segment of $u +\!\!+ v$, y remains in the result. Since each tail segment of u need to be concatenated with v to become a tail segment of $u +\!\!+ v$, the operator maps $(+\!\!+(\mathsf{head}\ y))$ (v is the head element of y) to x.

The following two lemmas are well-known lemmas of tails [Bir87].

Lemma 19 (Scanr)**.** *For any associative binary operator $\oplus$, the following equation holds.*

$$\mathsf{scanr}\ (\oplus) = \oplus/* \circ \mathsf{tails}$$

Lemma 20 (Tails-Map Promotion)**.** *For function f, the following equation holds.*

$$f** \circ \mathsf{tails} = \mathsf{tails} \circ f*$$

First lemma gives us a way to compute suffix sums by tails. The second lemma gives us a way to promote the application of function f through tails. The number of applications of function f on the left hand side is $n(n+1)/2$, while that on the right hand side is n. So, transformation from the left hand side to the right hand side improves the efficiency. Also, the lemma enables us to ignore a function g in the generate-and-test specification $(\!|\oplus, f|\!) \circ (\!|\otimes, g|\!)* \circ p\triangleleft \circ \mathsf{tails}$, since we can replace g with the identity function id and apply g to the input of tails beforehand (of course, we need some tricks to through $p\triangleleft$).

A related example problem is found in Section 3.4.1. Related implementation on GG library is found in Section 5.4.1.

4.4.2 Theorem for Reduction with Distributive Operators

The following theorem gives efficient parallel implementation of nested reductions for tails when two reductions have distributivity. One of the most famous problems for which this theorem is applicable is maximum tail-segment sum problem (also known as maximum suffix sum problem), which is an instance of maximum marking problems [SHTO00, Bir01]. A related example problem is found in Section 3.4.2. Related implementation on GG library is found in Section 5.4.2.

Theorem 21 (Maximum Tail-segment Sum)**.** *Provided that* $\oplus$ *is associative, and* $\otimes$ *is associative and right-distributive over* $\oplus$*, the following equation holds.*

$$\oplus/ \circ \otimes/* \circ \mathsf{tails} \;=\; \pi_1 \circ (\!|\circledast, \mathit{pair}|\!) \quad \textbf{where} \quad \begin{array}{l}(t_1, s_1) \circledast (t_2, s_2) = ((t_1 \otimes s_2) \oplus t_2, s_1 \otimes s_2) \\ \mathit{pair}\ a = (a, a)\end{array}$$

Proof. Similar to the proof of Theorem 15 in Section 4.3.2. □

The resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\circledast$, while the original program (the left hand side) uses two nested reductions with tails. The cost of the new operator is proportional to the cost of operators in the original reductions. So, the resulting program is more efficient than the original program.

The new operator $\circledast$ is applied to tuples. The first element of a tuple is equal to the result of the original program. The second element of the tuple is equal to the reduction of the same input with operator $\otimes$, which is used to improve efficiency of the program by reusing the partial results effectively.

4.4.3 Theorem for Filtering with Relational Predicates

The following theorem extends Theorem 21 by allowing filtering with the given predicate. It gives efficient parallel implementation of nested reductions for tails when two reductions have distributivity and the predicate is relational. Derivation of efficient sequential implementations for those programs with filtering is shown in [Zan92, Jeu93]. A related example problem is found in Section 3.4.3. Related implementation on GG library is found in Section 5.4.3.

Theorem 22 (Maximum p-Tail-segment Sum)**.** *Provided that* $\oplus$ *is associative,* $\otimes$ *is associative and right-distributive over* $\oplus$*, and predicate* p *is relational, the following equation holds.*

$$\oplus/ \circ \otimes/* \circ p \triangleleft \circ \mathsf{tails} = \pi_1 \circ (\!|\circledcirc, \mathit{pentuple}|\!)$$

where

$$(t_1, s_1, h_1, l_1, p_1) \circledcirc (t_2, s_2, h_2, l_2, p_2) = (t, s_1 \otimes s_2, h_1 \ll h_2, l_1 \gg l_2, p_1 \wedge p_2 \wedge p\ ([l_1, h_2]))$$

where $i = (\textbf{if}\ p_1 \wedge p\ ([l_1, h_2])\ \textbf{then}\ t_1 \otimes s_2\ \textbf{else}\ \imath_\oplus) \oplus t_2$

$$\mathit{pentuple}\ a = (a, a, a, a, T)$$

Proof. Similar to the proof of Theorem 16 in Section 4.3.3. □

The resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\circledcirc$, while the original program (the left hand side) uses two nested reductions with tails. The cost of the new operator is proportional to the cost of operators in the original reductions and the application of p. So, the resulting program is more efficient than the original program.

The new operator $\circledcirc$ is applied to pentuples. The first element of a pentuple is equal to the result of the original program. The second element of the pentuple is equal to the reduction of the same input with operator $\otimes$, which is used to improve efficiency of the program by reusing the partial results effectively. The third and forth elements are the edge elements of the input. Those edge elements are used to check whether results from two recursions in divide-and-conquer computation can be connected to make a better solution. Since the predicate p is relational, we can check the connectability by using only elements on the edge. The fifth element is a Boolean value that is the result of p applied to the input.

Here, we want to reduce the size of pentuples by eliminating the fifth element (it corresponds to $p\ (x)$) for simplicity. To do so, we use an assumption that $\imath_\otimes$ is the zero of $\otimes$, i.e. $a \otimes \imath_\oplus = \imath_\oplus$.

Corollary 23 (Maximum p-Tail-segment Sum (Simplified))**.** *Provided that* $\oplus$ *is associative,* $\otimes$ *is associative and right-distributive over* $\oplus$*, the identity* $\imath_\oplus$ *is the zero of* $\otimes$*, and predicate* p *is relational, the following equation holds.*

$$\oplus/ \circ \otimes/* \circ p\triangleleft \circ \mathsf{tails} = \pi_1 \circ ([\boxtimes, \mathit{quadruple}])$$
where
$$(t_1, s_1, h_1, l_1) \boxtimes (t_2, s_2, h_2, l_2) = ((t_1 \otimes s_2)_{l_1,h_2} \oplus t_2, (s_1 \otimes s_2)_{l_1,h_2}, h_1 \ll h_2, l_1 \gg l_2)$$
$$\mathit{quadruple}\ a = (a, a, a, a)$$
$$(a)_{l,h} = \textbf{if}\ p\ ([l, h])\ \textbf{then}\ a\ \textbf{else}\ \imath_\oplus$$

Proof. Similar to the proof of Corollary 17 in Section 4.3.3. □

The resulting program (the right hand side) uses the new reduction operator $\boxtimes$ that is applied on quadruples. The difference from the result of Theorem 22 is that the second element becomes the identity $\imath_\oplus$ of $\oplus$ when the input does not satisfy the predicate p.

A sufficient condition of the assumption that $\imath_\oplus$ is the zero of $\otimes$ is that $\otimes$ distributes over $\oplus$. So, we will use this condition in implementation of the library.

4.5 Theory of segs

This section gives formal definition and theory of GGenerator segs.

4.5.1 Formal Definition and Basic Lemmas

First of all, we give a formal definition of segs using inits and tails.

Definition 24 (GGenerator Segs)**.** *GGenerator* segs *is defined with homomorphism as follows.*

$$\mathsf{segs} = +\!\!+/ \circ \mathsf{inits}* \circ \mathsf{tails}$$

This definition is based on the fact that a segment of a list is a prefix of a suffix of the list.

The following lemma is well-known lemma of segs [Bir87].

Lemma 25 (Segs-Map Promotion)**.** *For function* f*, the following equation holds.*

$$f** \circ \mathsf{segs} = \mathsf{segs} \circ f*$$

The lemma gives us a way to promote the application of function f through segs. The number of applications of function f on the left hand side is $O(n^3)$, while that on the right hand side is n. So, transformation from the left hand side to the right hand side improves the efficiency. Also, the lemma enables us to ignore a function g in the generate-and-test specification $([\oplus, f]) \circ ([\otimes, g])* \circ p\triangleleft \circ \mathsf{segs}$, since we can replace g with the identity function id and apply g to the input of segs beforehand (of course, we need some tricks to through $p\triangleleft$).

A related example problem is found in Section 3.5. Related implementation on GG library is found in Section 5.5.1.

4.5.2 Theorem for Reduction with Distributive Operators

The following theorem gives efficient parallel implementation of nested reductions for segs when two reductions have distributivity. One of the most famous problems for which this theorem is applicable is maximum segment sum problem [Bir87], which is an instance of maximum marking problems [SHTO00, Bir01]. A related example problem is found in Section 3.5.2. Related implementation on GG library is found in Section 5.5.2.

Theorem 26 (Maximum Segment Sum)**.** *Provided that $\oplus$ is associative and commutative, and $\otimes$ is associative and distributive over $\oplus$, the following equation holds.*

$$\oplus/ \circ \otimes/* \circ \mathsf{segs} = \pi_1 \circ ([\circledcirc, \mathit{quadruple}])$$

where

$$(m_1, t_1, i_1, s_1) \circledcirc (m_2, t_2, i_2, s_2) = (m_1 \oplus m_2 \oplus (t_1 \otimes i_2), (t_1 \otimes s_2) \oplus t_2, i_1 \oplus (s_1 \otimes i_2), s_1 \otimes s_2)$$

$$\mathit{quadruple}\ a = (a, a, a, a)$$

Proof. We can prove the theorem by the following calculation.

$$\begin{array}{ll}
 & \mathit{LHS} \\
= & \quad \{\ \text{LHS}\ \} \\
 & \oplus/ \circ \otimes/* \circ \mathsf{segs} \\
= & \quad \{\ \text{definition of } \mathsf{segs}\ \} \\
 & \oplus/ \circ \otimes/* \circ +\!\!+/ \circ \mathsf{inits}* \circ \mathsf{tails} \\
= & \quad \{\ \text{promotion of } *\ \} \\
 & \oplus/ \circ +\!\!+/ \circ \otimes/** \circ \mathsf{inits}* \circ \mathsf{tails} \\
= & \quad \{\ \text{promotion of } /\ \} \\
 & \oplus/ \circ \oplus/* \circ \otimes/** \circ \mathsf{inits}* \circ \mathsf{tails} \\
= & \quad \{\ \text{distributivity of } *\ \} \\
 & \oplus/ \circ (\oplus/ \circ \otimes/* \circ \mathsf{inits})* \circ \mathsf{tails} \\
= & \quad \{\ \text{Theorem 15}\ \} \\
 & \oplus/ \circ (\pi_1 \circ ([\odot, \mathit{pair}]))* \circ \mathsf{tails} \\
= & \quad \{\ \text{making } \oplus' \text{ so that } (a_1, b_1) \oplus' (a_2, b_2) = (a_1 \oplus a_2, b_1 \oplus b_2), \text{ Lemma 20}\ \} \\
 & \pi_1 \circ \oplus'/ \circ \odot/* \circ \mathsf{tails} \circ \mathit{pair}* \\
= & \quad \{\ \text{Theorem 21 (commutativity of } \oplus \text{ guarantees distributivity of } \odot \text{ over } \oplus')\ \} \\
 & \pi_1 \circ \pi_1 \circ ([\circledast, \mathit{pair}]) \circ \mathit{pair}* \\
= & \quad \{\ \text{fusing two } \pi_1\text{s, and two } \mathit{pair}\text{s}\ \} \\
 & \pi_1 \circ ([\circledcirc, \mathit{quadruple}]) \\
= & \quad \{\ \text{RHS}\ \} \\
 & \mathit{RHS}
\end{array}$$

Definition of $\circledast$ in the above calculation is given as follows.

$$\begin{array}{ll}
 & ((m_1, t_1), (i_1, s_1)) \circledast ((m_2, t_2), (i_2, s_2)) \\
= & \quad \{\ \text{definition of } \circledast \text{ in Theorem 21}\ \} \\
 & (((m_1, t_1) \odot (i_2, s_2)) \oplus' (m_2, t_2), (i_1, s_1) \odot (i_2, s_2)) \\
= & \quad \{\ \text{definition of } \odot \text{ in Theorem 15}\ \} \\
 & ((m_1 \oplus (t_1 \otimes i_2), t_1 \otimes s_2) \oplus' (m_2, t_2), (i_1 \oplus (s_1 \otimes i - 2), s_1 \otimes s_2)) \\
= & \quad \{\ \text{definition of } \oplus' \text{ shown above}\ \} \\
 & ((m_1 \oplus (t_1 \otimes i_2) \oplus m_2, (t_1 \otimes s_2) \oplus t_2), (i_1 \oplus (s_1 \otimes i - 2), s_1 \otimes s_2))
\end{array}$$

Flattening the nested pair into quadruple, we get the definition of $\circledcirc$. □

The resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\circledcirc$, while the original program (the left hand side) uses two nested reductions with segs. The cost of the new operator is proportional to the cost of operators in the original reductions. So, the resulting program is more efficient than the original program.

The new operator $\circledcirc$ is applied to quadruples. The first element of a tuple is equal to the result of the original program. The second and the third elements are results of the original nested reductions with tails and inits. The last element is equal to the reduction of the same input with operator $\otimes$. These extra elements are used to improve efficiency of the program by reusing the partial results effectively.

4.5.3 Theorem for Filtering with Relational Predicates

The following theorem extends Theorem 26 by allowing filtering with the given predicate. It gives efficient parallel implementation of nested reductions for segs when two reductions have distributivity and the predicate is relational. Derivation of efficient sequential implementations for those programs with filtering is shown in [Zan92, Jeu93]. A related example problem is found in Section 3.5.3. Related implementation on GG library is found in Section 5.5.3.

Theorem 27 (Maximum p-Segment Sum)**.** *Provided that $\oplus$ is associative and commutative, $\otimes$ is associative and distributive over $\oplus$, and predicate p is relational, the following equation holds.*

$$
\begin{aligned}
&\oplus/ \circ \otimes/* \circ p\triangleleft \circ \mathsf{segs} = \pi_1 \circ (\!|\boxdot, \mathit{hextuple}|\!) \\
&\quad \textbf{where} \quad (m_1, t_1, i_1, s_1, h_1, l_1) \boxdot (m_2, t_2, i_2, s_2, h_2, l_2) \\
&\qquad\qquad = (\ m_1 \oplus m_2 \oplus (t_1 \otimes i_2)_{l_1,h_2}, \\
&\qquad\qquad\quad\ (t_1 \otimes s_2)_{l_1,h_2} \oplus t_2, \\
&\qquad\qquad\quad\ i_1 \oplus (s_1 \otimes i_2)_{l_1,h_2}, \\
&\qquad\qquad\quad\ (s_1 \otimes s_2)_{l_1,h_2}, \\
&\qquad\qquad\quad\ h_1 \ll h_2, l_1 \gg l_2) \\
&\qquad \mathit{hextuple}\ a = (a, a, a, a, a, a) \\
&\qquad (a)_{l,h} = \textbf{if}\ p\ ([l, h])\ \textbf{then}\ a\ \textbf{else}\ \imath_\oplus
\end{aligned}
$$

Proof. We can prove the theorem by the following calculation.

$$
\begin{aligned}
& \mathit{LHS} \\
= \quad & \{\ \text{LHS}\ \} \\
& \oplus/ \circ \otimes/* \circ p\triangleleft \circ \mathsf{segs} \\
= \quad & \{\ \text{definition of segs}\ \} \\
& \oplus/ \circ \otimes/* \circ p\triangleleft \circ +\!\!+/ \circ \mathsf{inits}* \circ \mathsf{tails} \\
= \quad & \{\ \text{promotion of } \triangleleft, *\text{ and } /\ \} \\
& \oplus/ \circ \oplus/* \circ \otimes/* \circ p\triangleleft* \circ \mathsf{inits}* \circ \mathsf{tails} \\
= \quad & \{\ *\text{ distributivity}\ \} \\
& \oplus/ \circ (\oplus/ \circ \otimes/ \circ p\triangleleft \circ \mathsf{inits})* \circ \mathsf{tails} \\
= \quad & \{\ \text{Theorem 17}\ \} \\
& \oplus/ \circ (\pi_1 \circ (\!|\boxtimes, \mathit{quadruple}|\!))* \circ \mathsf{tails} \\
= \quad & \left\{ \begin{array}{l} \text{making } \oplus' \text{ such that} \\ (i_1, s_1, h_1, l_1) \oplus' (i_2, s_2, h_2, l_2) = (i_1 \oplus i_2, s_1 \oplus s_2, h_1 \ll h_2, l_1 \gg l_2) \end{array} \right\} \\
& \pi_1 \circ \oplus'/ \circ \boxtimes/* \circ \mathsf{tails} \circ \mathit{quadruple}* \\
= \quad & \{\ \text{Theorem 21 } (\boxtimes \text{ distributes over } \oplus')\ \} \\
& \pi_1 \circ \pi_1 \circ (\!|\circledast, \mathit{pair}|\!) \circ \mathit{quadruple}* \\
= \quad & \{\ \text{fusing two } \pi_1\text{s, fusing } \mathit{pair} \text{ and } \mathit{quadruple}\text{, removing duplicated parts}\ \} \\
& \pi_1 \circ (\!|\boxdot, \mathit{hextuple}|\!) \\
= \quad & \{\ \text{RHS}\ \} \\
& \mathit{RHS}
\end{aligned}
$$

Note that distributivity of $\boxtimes$ over $\oplus'$ is guaranteed for quadruples (i_1, s_1, h_1, l_1) and (i_2, s_2, h_2, l_2) of operands of $\oplus'$ when $l_1 = l_2$. Condition $l_1 = l_2$ holds in the above calculation since l_1 and l_2 are the last element of tail-segments of the same list.

Definition of $\circledast$ in the above calculation is given as follows.

$$
\begin{aligned}
& ((m_1,t_1,h_1,l_1),(i_1,s_1,k_1,n_1)) \circledast ((m_2,t_2,h_2,l_2),(i_2,s_2,k_2,n_2)) \\
= \quad & \{ \text{ definition of } \circledast \text{ in Theorem 21 } \} \\
& (((m_1,t_1,h_1,l_1) \boxtimes (i_2,s_2,k_2,n_2)) \oplus' (m_2,t_2,h_2,l_2),(i_1,s_1,k_1,n_1) \boxtimes (i_2,s_2,k_2,n_2)) \\
= \quad & \{ \text{ definition of } \boxtimes \text{ in Theorem 17 } \} \\
& ((m_1 \oplus (t_1 \otimes i_2)_{l_1,k_2}, (t_1 \otimes s_2)_{l_1,k_2}, h_1 \ll k_2, l_1 \gg n_2) \oplus' (m_2,t_2,h_2,l_2), \\
& \quad (i_1 \oplus (s_1 \otimes i_2)_{n_1,k_2}, (s_1 \otimes s_2)_{n_1,k_2}, k_1 \ll k_2, n_1 \gg n_2) \\
= \quad & \{ \text{ definition of } \oplus' \text{ shown above } \} \\
& ((m_1 \oplus m_2 \oplus (t_1 \otimes i_2)_{l_1,k_2}, (t_1 \otimes s_2)_{l_1,k_2} \oplus t_2, h_1 \ll k_2 \ll h_2, l_1 \gg n_2 \gg l_2), \\
& \quad (i_1 \oplus (s_1 \otimes i_2)_{n_1,k_2}, (s_1 \otimes s_2)_{n_1,k_2}, k_1 \ll k_2, n_1 \gg n_2)
\end{aligned}
$$

If $h_1 = k_1$, $l_1 = n_1$, $h_2 = k_2$ and $l_2 = n_2$, then we have $h_1 \ll k_2 \ll h_2 = k_1 \ll k_2$ and $l_1 \gg n_2 \gg l_2 = n_1 \gg n_2$. So, for octuple $((m,t,h,l),(i,s,k,n))$, we have invariant $h = k$ and $l = n$. Using this invariant, we can eliminate k and n from octuples and we get computation using hextuples. Finally, flattening the hextuples, we get the definition of $\circledcirc$. □

The resulting program (the right hand side) of the theorem uses only one reduction with the new operator $\boxdot$, while the original program (the left hand side) uses two nested reductions with segs. The cost of the new operator is proportional to the cost of operators in the original reductions. So, the resulting program is more efficient than the original program.

The new operator $\boxdot$ is applied to hextuples. The first element of a tuple is equal to the result of the original program. The second and the third elements are results of the original program in which segs is replaced with tails and inits. The fourth element is equal to the reduction of the same input with operator $\otimes$. The fifth and the sixth elements are the edge elements of the input. Those edge elements are used to check whether results from two recursions in divide-and-conquer computation can be connected to make a better solution. Since the predicate p is relational, we can check the connectability by using only elements on the edge. These extra elements are used to improve efficiency of the program by reusing the partial results effectively.

4.6 Related Work

4.6.1 Maximum Marking Problem

Maximum marking problem is one of optimization problems on sequences. The objective of maximum marking problem is to find a marking of the input sequence that gives the maximum weight sum. We can produce many instances of maximum marking problem by changing the rule of marking on the input sequence. For example, maximum prefix sum problem (Section 3.3.2), maximum suffix sum problem (Section 3.4.2), and maximum segment sum problem (Section 3.5.2) are instances of maximum marking problem.

Derivation of efficient sequential programs for maximum marking problem was studied by Sasano et al. [SHTO00, SHT01] and Bird [Bir01]. They showed systematic construction of efficient sequential programs when rules of marking are given as recursive functions of a certain class. Since marking of a prefix/suffix/segment is given as a recursive function of the class, an efficient sequential program for maximum prefix/suffix/segment problem is obtained systematically by their results.

4.6.2 Longest Segment Problems

The objective of longest segment problems is to find the longest segment that satisfies a given predicate. Derivation of efficient sequential programs for longest segment problems were stud-

ied by Zatema [Zan92], Jeuring [Jeu93] and Zhao [Zha02]. Zatema [Zan92] and Jeuring [Jeu93] showed derivation of efficient sequential programs for various predicates. Zhao [Zha02] studied derivation of efficient sequential programs for predicates constructed as a combination of primitive predicates. She applied it for data mining to make a querying system that supports efficient querying of combined predicates.

```
trait GGenerator⟦E⟧ extends List⟦List⟦E⟧⟧
  getter list() : List⟦E⟧
  getter defaultImplementation() : List⟦E⟧ → List⟦List⟦E⟧⟧
  generate2⟦R,F,M,N,P,Z,Y⟧(r : R, f : F, mr : M, mf : N, p : P) : Z
= defaultgeneration⟦R,F,M,N,P,Z,Y⟧(r, f, mr, mf, p)
  defaultgeneration⟦R,F,M,N,P,Z,Y⟧(r : R, f : F, mr : M, mf : N, p : P) : Z =
    actualList().filter(fn (e:List⟦E⟧):Boolean ⇒ p.judge(e))
                .generate⟦Z⟧(r, (fn a ⇒ f.apply(a))∘
                  (fn (x) ⇒ x.generate⟦Y⟧(mr, (fn a ⇒ mf.apply(a)))))
  actualList() : List⟦List⟦E⟧⟧ = defaultImplementation()(list())
end
```

Figure 3: Base trait of GGenerators

5 Implementation of GG Library

This section gives implementation of GG library. First, implementation of the core of GG library is given in Section 5.1. Then, a collection of traits to describe properties is given in Section 5.2. After that, a collection of implementations for various GGenerators follows.

5.1 Core Implementation of GG Library

The core of GG Library is trait GGenerator (Section 5.1.1), which is the base of all GGenerators, and function *dispatching* (Section 5.1.2), which acts as a dispatching table of efficient implementations. Intuitive interpretation of GGenerators is found in Section 3.1.2. Formal discussion of GGenerators is found in Section 4.1.2

Besides the above two constructs, auxiliary functions and object to define GGenerators are shown in Section 5.1.3, and desugaring of user programs for GG library is discussed in Section 5.1.4.

5.1.1 **Trait** GGenerator

Trait GGenerator is the base trait of all GGenerators. Figure 3 shows the definition of trait GGenerator (the figure contains only essential fields).

Currently, GGenerator⟦E⟧ is defined as a subtrait of trait List⟦List⟦E⟧⟧, which is a subtrait of Fortress's Generator. The core of trait Generator is method *generate* that generates elements of type E, passes each of them to the function *body*, and combines the results using the reduction r.

Trait GGenerator extends the existing Generator as follows. First, GGenerator itself can work as Generator to generate elements of type List⟦E⟧, since GGenerator is a subtrait of Generator. Next, method *generate2* of GGenerator is an extension of method *generate* of Generator. Method *generate2* takes two pairs of operators and body functions, namely the pair (r, f) and the pair (mr, mf), for nested reductions so that it can use relationship between the given pairs to perform the reductions efficiently, while *generate* takes a pair of an operator and a function to perform a single reduction, performing nested reductions individually. Of course, if the nested reductions use the same operator for each reduction, both *generate2* and nested use of *generate* result in the same computation.

The rest of this section explains methods of trait GGenerator.

Getter *list*() returns the original list given to the GGenerator. Basically, values in the generated nested lists are taken from this original list.

Getter *defaultImplementation*() returns implementation to generate the actual nested data structures. Basically, an efficient implementation does not generate such actual nested data structures. Implementation given by *defaultImplementation* is used to perform the nested reductions naively as the default of dispatching.

Method *generate2* is the most important method of trait GGenerator that is an interface of dispatching implementation to the nested reductions. Meaning of an invocation of $generate2(r, f, mr, mf, p)$ of GGenerator gg is the same as the following nested comprehension with nested reductions.

$$r\ [\ f(\ mr\ [\ mf\ y\ |\ y \leftarrow ys\ ]\ )\ |\ ys \leftarrow gg\ xs, p\ ys\ ]$$

GGenerator gg generates a nested data structure, and its element (basically a subsequence of the input xs) is bound to variable ys. Predicate p is used to filter the generated element ys. Function mf is applied to each element of ys passed the filtering, and a reduction with mr is taken on the result. And then, function f is applied for each result of the inner reduction with mr, and finally reduction with r is taken on those results. Straightforward implementation of the computation explained above is seen in method *defaultgeneration* explained below.

Method *defaultgeneration* performs the default naive nested reductions on the actual nested data structure. Arguments are the same as method *generate2* explained above. The actual nested data structure is generated by *actualList*() explained later. Against the generated nested data structure, it performs filtering with the given predicate p by method *filter* of List. Judgment by the predicate p is denoted by $p.judge(e)$. After that, it performs the nested reductions with two invocations of method *generate* of Generators. The inner reduction is performed with the given reduction mr and function mf. Application of the function mf is denoted by $mf.apply(a)$. The outer reduction is performed with the given reduction r and function f composed with the inner reduction. The computation by method *defaultgeneration* is the same as a program of nested comprehensions desugared by the usual desugaring process of Fortress.

Method *acutalList*() generates the actual nested data structure by the implementation given by *defaultImplementation*(). This actual list is used in some methods, such as taking the head of the generated nested data structure, as well as the default naive nested reductions.

It is worth mentioning about type variables used in method *generate2*. There are many type variables in the method and they have no restrictions. The reason of a number of type variables is as follows. Method *generate2* wants to know complete types of the arguments and bind those types to variables, since the dispatching process needs to check properties of the arguments by their types to dispatch efficient implementation. The reason of no restriction on types is basically the limitation of the current interpreter, which does not completely support where-clause. When where-clause is supported in the future, we can add restriction on type variables like shown below.

$$\begin{array}{l} generate_2[\![\, R, F, M, N, P, Z, Y\,]\!](r:\ R, f:\ F, mr:\ M, mf:\ N, p:\ P):\ Z \\ \quad \texttt{where}\ \{\, R\ \texttt{extends}\ \text{Reduction}[\![Z]\!], F\ \texttt{extends}\ \text{Function}[\![Y, Z]\!], \\ \qquad\qquad M\ \texttt{extends}\ \text{Reduction}[\![Y]\!], N\ \texttt{extends}\ \text{Function}[\![E, Y]\!], \\ \qquad\qquad P\ \texttt{extends}\ \text{ListPredicate}[\![E]\!]\} \\ \quad = defaultgeneration[\![\, R, F, M, N, P, Z, Y\,]\!](r, f, mr, mf, p) \end{array}$$

5.1.2 Dispatching Table

The dispatching process dispatches efficient implementation given by a theorem to an invocation of *generate2* of a GGenerator (a desugared user program) when the arguments satisfy the applicable condition of the theorem. Basically, checking of applicable conditions of theorems against the arguments is performed by checking types of the arguments, since properties of the arguments should be specified by traits shown in Section 5.2.

The dispatching table shown in Figure 4 is the core of dispatching process. The dispatching table is defined as function *dispatching* that checks types of the given arguments and selects

```
dispatching⟦G, R, F, M, N, P, Z, Y, E⟧(g : G, r : R, f : F, mr : M, mf : N, p : P) = do
 typecase (g, r, f, mr, mf, p) of
  (InitsGenerator⟦E⟧, R, IdFunction⟦Y⟧, LeftDistributiveOver⟦R⟧, N, TrueListPredicate⟦E⟧)
    ⇒ g.efficientImplTrueDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
  (InitsGenerator⟦E⟧, R, IdFunction⟦Y⟧, LeftDistributiveOver⟦R⟧, N, RelationalPredicate⟦E⟧)
    ⇒ g.efficientImplRelationalDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
  (TailsGenerator⟦E⟧, R, IdFunction⟦Y⟧, RightDistributiveOver⟦R⟧, N, TrueListPredicate⟦E⟧)
    ⇒ g.efficientImplTrueDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
  (TailsGenerator⟦E⟧, R, IdFunction⟦Y⟧, RightDistributiveOver⟦R⟧, N, RelationalPredicate⟦E⟧)
    ⇒ g.efficientImplRelationalDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
  (SegsGenerator⟦E⟧, Commutative, IdFunction⟦Y⟧, DistributiveOver⟦R⟧, N, TrueListPredicate⟦E⟧)
    ⇒ g.efficientImplTrueDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
  (SegsGenerator⟦E⟧, Commutative, IdFunction⟦Y⟧, DistributiveOver⟦R⟧, N, RelationalPredicate⟦E⟧)
    ⇒ g.efficientImplRelationalCommutativeDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
  else
    ⇒ g.defaultgeneration⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
 end
end
```

Figure 4: Dispatching table

suitable implementation according to the arguments. Function *dispatching* receives a GGenerator and arguments given to *generate2* of the GGenerator. Each GGenerator should invoke function *dispatching* to perform dispatching in *generate2* (see implementation of GGenerators shown in the following sections).

Function *dispatching* checks types of arguments by `typecase` of Fortress to dispatch suitable implementation. Basically, one case of `typecase` corresponds to one theorem, and the default (`else`) case corresponds to naive implementation of nested reductions. Figure 4 contains cases for theorems given in Section 4 and the default case. For the default case (`else` case), the dispatching table invokes method *defaultgeneration* of GGenerator g. In the other cases, it checks whether properties of the arguments satisfy applicable conditions by checking types of the arguments, to dispatch the corresponding efficient implementation.

For example, the first case of the table shown in Figure 4 corresponds to Theorem 15 in Section 4.3.2. The theorem requires that the inner reduction (mr) has distributivity over the outer reduction (r). So, function *dispatching* checks whether mr extends LeftDistributiveOver⟦R⟧ (see Section 5.2.1). Also, function *dispatching* checks that f is the identity function (see Section 5.2.2) and p is the true-predicate (see Section 5.2.3), since the theorem cannot deal with other functions and predicates. If those conditions on types are satisfied, function *dispatching* confirms that the applicable condition of the theorem is satisfied, and it invokes the efficient implementation *efficientImplTrueDistributive* of InitsGenerator (see Section 5.3.2) to perform the nested reductions of a user program efficiently by the implementation.

Summary of dispatching is as follows. Method *generate2* of each GGenerator invokes function *dispatching* with the GGenerator and its arguments. The `typecase` in function *dispatching* checks whether the given arguments satisfy applicable condition of each theorem by checking their types. If it confirms that the arguments satisfy the condition, it invokes the corresponding efficient implementation of the theorem. If no condition is satisfied, the default implementation is used in the default case of the `typecase`.

To exploit knowledge of a theorem, an implementer has to do two things. One is to implement the optimization given by the theorem in its corresponding GGenerator. The other is to modify the `typecase` in function *dispatching* by adding a new case of type conditions corresponding to

```
takeleft⟦T⟧(a : Nothing⟦T⟧, b : Any) = b
takeleft(a : Any, b : Any) = a
takeright⟦T⟧(a : Any, b : Nothing⟦T⟧) = a
takeright(a : Any, b : Any) = b

object MapReduceReduction⟦R⟧(j : (R, R) → R, z : R) extends Reduction⟦R⟧
  empty() = z
  join(a : R, b : R): R = (j)(a, b)
end
```

Figure 5: Auxiliary functions and object

the applicable condition of the theorem, and a code to invoke the efficient implementation of the GGenerator in the case. Then, nested reductions of a user program can be executed with the efficient implementation given by the theorem, if the nested reduction satisfies the applicable condition of the theorem.

5.1.3 Auxiliary Functions and Object

Figure 5 gives implemetation of auxiliary functions and object for defining GGenerators.

Functions *takeleft* and *takeright* provides function of operators $\ll$ and $\gg$ (used in Section 4) with identities. Their identities are represented by a special value Nothing.

Object MapReduceReduction makes a reduction object from an associative binary operator and its identity, and is used in invocations of method *generate* of Generator. This object is borrowed from the Fortress Standard Library.

5.1.4 Desugaring of User Programs

Currently, desugaring of a user program into invocations of method *generate2* of GGenerators does not completely work. It is very difficult to desugar any expression into *generate2*, since it needs to split an expression into a function and a reduction. Thus, we are planning to desugar restricted nested comprehensions that can be transformed into a form below.

$$\bigoplus [\ \bigotimes [\ f\ y \mid y \leftarrow ys\ ] \mid ys \leftarrow gg\ xs, p\ ys\ ] \tag{1}$$

Here, gg is one of GGenerators, p is a predicate, f is a function, and $\oplus$ and $\otimes$ are associative operators. This form is the same computation as the following invocation of *generate2*.

$$gg(x).generate_2(\text{BinReduction}[\![\oplus]\!], \text{IdFunction}, \text{BinReduction}[\![\otimes]\!], f, p)$$

Here, BinReduction makes a reduction object from an associative binary operator.

We show, with an example, that quite a lot of nested comprehensions can be systematically desugared into the above form. Consider the next nested comprehension as our example.

$$\uparrow[+[\ f(y, b, w) \mid y \leftarrow ys, even\ y\ ] \mid ys \leftarrow inits\ xs, ascending\ ys, b \leftarrow bs\ ]$$

This example computes a variant of the maximum prefix sum, in which the maximum is considered only on ascending prefixes, the summation is taken only on even numbers, and the value is replaced with $f(y, b, w)$ instead of the number itself (y) at the summation.

Transformation steps with the example program are shown below.

Step. 1 Remove guards $p\ x$ by fusing it with generators $x \leftarrow g\ xs$
There are two occurrences of guards in the example: *even* y and *ascending* ys. Fusing these guards, we can get the following program.

$$\uparrow[+[\ f(y, b, w) \mid y \leftarrow filter\ even\ ys\ ] \mid ys \leftarrow filter ascending\ (inits\ xs), b \leftarrow bs\ ]$$

This transformation is applicable, when each predicate depends only on a variable in the left hand side of generators in the same comprehension.

Step. 2 Move depending generations to the edges
If there is depending generations in generators of two comprehensions, move those depending generation to the edges of comprehensions as follows.

$$\oplus[\,\otimes[\,e \mid gs_1\,] \mid gs_2\,] \;\Rightarrow\; \oplus[\,\otimes[\,e \mid y \leftarrow fg\ ys, gs_1'\,] \mid gs_2', ys \leftarrow fgg\ xs\,]$$

Here, *fgg* is one of GGenerators with filter, and *fg* is the identity function or filter. This transformation is valid if each operator of reductions is commutative and there is no dependency of gs_2' to ys.

The example program has a pair of depending generations $y \leftarrow \mathit{filter\ even\ ys}$ and $ys \leftarrow \mathit{filter ascending\ (inits\ xs)}$. Since operators used in our example are both commutative, we can perform this transformation to get the following program.

$$\uparrow[\,+[\,f(y,b,w) \mid y \leftarrow \mathit{filter\ even\ ys}\,] \mid b \leftarrow bs, ys \leftarrow \mathit{filter ascending\ (inits\ xs)}\,]$$

Here, $ys \leftarrow \mathit{filter ascending\ (inits\ xs)}$ is moved to the edge using commutativity of $\uparrow$.

Step. 3 Restructure comprehensions to extract the form
The following is a rule used in this step.

$$\begin{array}{l}\oplus[\,\otimes[\,e \mid y \leftarrow fg\ ys, gs_1'\,] \mid gs_2', ys \leftarrow fgg\ xs\,] \\ \quad \Rightarrow \oplus[\,\oplus[\,\otimes[\,\otimes[\,e \mid gs_1'\,] \mid y \leftarrow fg\ ys\,] \mid ys \leftarrow fgg\ xs\,] \mid gs_2'\,]\end{array}$$

This transformation is always valid, since it is a combination of steps used in the usual desugaring process in Fortress. For readability, the result of this transformation can be written as the following form.

$$\begin{array}{l} h(\oplus[\,\otimes[\,f'(y) \mid y \leftarrow fg\ ys\,] \mid ys \leftarrow fgg\ xs\,]) \\ \quad \textbf{where}\ \ h(z) = \oplus[\,z \mid gs_2'\,] \\ \qquad\qquad f'(y) = \otimes[\,e \mid gs_1'\,]\end{array}$$

Here, the argument of h is almost the same as the form (1). The difference can be eliminated in the following way. If *fg* is *filter q*, we introduce another function $f''(x) =$ **if** $q(x)$ **then** x **else** $\imath_\otimes$, in which $\imath_\otimes$ is the identity of $\otimes$. Otherwise, let $f'' = f'$. If *fgg* does not includes filter, we introduce $p = \mathit{true}$ that always returns true. Otherwise, let p be the predicate of the filter, i.e., $\mathit{fgg\ xs} = \mathit{filter\ p\ (gg\ xs)}$. Using these f'' and p, the argument of h is now the same as the form (1).

$$\oplus[\,\otimes[\,f''(y) \mid y \leftarrow ys\,] \mid ys \leftarrow gg\ xs, p\ ys\,])$$

If there is no direct dependency of f'' to ys, we can replace this part by an invocation of method *generate2* of *gg*.

Applying the above transformation, we get the following result for the example.

$$\begin{array}{ll} & \uparrow[\,+[\,f(y,b,w) \mid y \leftarrow \mathit{filter\ even\ ys}\,] \mid b \leftarrow bs, ys \leftarrow \mathit{filter ascending\ (inits\ xs)}\,] \\ \Rightarrow & h(\uparrow[\,+[\,f(y,b,w) \mid y \leftarrow \mathit{filter\ even\ ys}\,] \mid ys \leftarrow \mathit{filter ascending\ (inits\ xs)}\,]) \\ & \quad \textbf{where}\ h(z) = \oplus[\,z \mid b \leftarrow bs\,] \\ \Rightarrow & h(\uparrow[\,+[\,f''\ y \mid y \leftarrow ys\,] \mid ys \leftarrow \mathit{inits\ xs}, \mathit{ascending\ ys}\,]) \\ & \quad \textbf{where}\ h(z) = \oplus[\,z \mid b \leftarrow bs\,] \\ & \qquad\qquad f''(z) = \textbf{if}\ \mathit{even}\ x\ \textbf{then}\ x\ \textbf{else}\ 0 \end{array}$$

Here, the argument of h is the same as the form (1). Since there is no direct dependency of f'' to ys, the part can be replaced with invocation of *generate2* of GGenerator *inits*.

$$mis = \texttt{BIG MAX}\,\langle \sum y \mid y \leftarrow inits\ x \rangle$$
$$mais = \texttt{BIG MAX}\,\langle \sum y \mid y \leftarrow inits\ x, ascending(y) \rangle$$

(a) User program

$$\begin{aligned} mis = inits(x).generate_2[\!| & \text{MaxReductionZZ32}, \text{IdFunction}[\![\mathbb{Z}32]\!], \\ & \text{SumReductionZZ32}, \text{IdFunction}[\![\mathbb{Z}32]\!], \\ & \text{TrueListPredicate}[\![\mathbb{Z}32]\!], \mathbb{Z}32, \mathbb{Z}32] \\ & (\text{MaxReductionZZ32}, \text{IdFunction}[\![\mathbb{Z}32]\!], \\ & \text{SumReductionZZ32}, \text{IdFunction}[\![\mathbb{Z}32]\!], \\ & \text{TrueListPredicate}[\![\mathbb{Z}32]\!]) \end{aligned}$$
$$\begin{aligned} mais = inits(x).generate_2[\!| & \text{MaxReductionZZ32}, \text{IdFunction}[\![\mathbb{Z}32]\!], \\ & \text{SumReductionZZ32}, \text{IdFunction}[\![\mathbb{Z}32]\!], \\ & \text{Ascending}[\![\mathbb{Z}32]\!], \mathbb{Z}32, \mathbb{Z}32] \\ & (\text{MaxReductionZZ32}, \text{IdFunction}[\![\mathbb{Z}32]\!], \\ & \text{SumReductionZZ32}, \text{IdFunction}[\![\mathbb{Z}32]\!], \\ & \text{Ascending}[\![\mathbb{Z}32]\!]) \end{aligned}$$

(b) Desugared program

Figure 6: Examples of desugaring user programs

The desugaring transformation shown above has some restrictions on target comprehensions. For example, Step. 1 requires that each predicate should depend only on a variable in the left hand side of generators in the same comprehension, Step. 2 requires that the operators should be commutative and there is no dependency of outer generators to generation of the GGenerator, and replacement of comprehension with *generate2* in Step. 3 requires that there is no direct dependency of the inner function to generation of the GGenerator. One easy sufficient restriction for dependencies of generators is that a variable on the left hand side of arrows in generators is used at most once in the right hand side of arrows in the generators and the body function. This restriction is often satisfied.

Some examples of desugaring is shown in Figure 6. Please refer to Section 5.2 for objects used in the desugared program.

5.2 Traits for Describing Properties

5.2.1 Properties on Reduction Operators

Figure 7 gives definitions of traits for describing properties on reduction operators, which are defined in Section 4.2.1.

Trait LeftDistributiveOver$[\![X]\!]$ is used to indicate that a reduction object extending the trait is left distributive over the reduction object X. Similarly, trait RightDistributiveOver$[\![X]\!]$ is used to indicate right-distributivity. Trait DistributiveOver$[\![X]\!]$ indicates the reduction object extending the trait is distributive, i.e., left-distributive and right-distributive, over the reduction object X. Thus, trait DistributiveOver extends both trait LeftDistributiveOver and trait RightDistributiveOver. Trait Commutative indicates commutativity.

Example use of those traits is also shown in Figure 7. Object MaxReductionZZ32 is a reduction object of the maximum operator $\uparrow$ defined in Section 3.1. Since the maximum operator is commutative, MaxReductionZZ32 extends the trait Commutative. Object SumReductionZZ32 is a reduction object of the usual plus operator $+$. Since the plus distributes over the maximum operator, SumReductionZZ32 extends trait DistributiveOver$[\![$MaxReductionZZ32$]\!]$, as well as trait Commutative. Similarly, object ProdReductionZZ32 for the usual product operator $\times$ extends DistributiveOver$[\![$SumReductionZZ32$]\!]$ and Commutative.

```
trait LeftDistributiveOver⟦X⟧ end
trait RightDistributiveOver⟦X⟧ end
trait DistributiveOver⟦X⟧ extends {LeftDistributiveOver⟦X⟧, RightDistributiveOver⟦X⟧} end

trait Commutative end

object MaxReductionZZ32
  extends { Reduction⟦ℤ32⟧, Commutative }
  empty(): ℤ32 = −infinity
  join(a: ℤ32, b: ℤ32): ℤ32 = a MAX b
end

object SumReductionZZ32
  extends { Reduction⟦ℤ32⟧, DistributiveOver⟦MaxReductionZZ32⟧, Commutative }
  empty(): ℤ32 = 0
  join(a: ℤ32, b: ℤ32): ℤ32 = a + b
end

object ProdReductionZZ32
  extends { Reduction⟦ℤ32⟧, DistributiveOver⟦SumReductionZZ32⟧, Commutative }
  empty(): ℤ32 = 1
  join(a: ℤ32, b: ℤ32): ℤ32 = a ∗ b
end
```

Figure 7: Traits for describing properties on reductions and example reductions

5.2.2 Properties on Functions

Figure 8 gives definitions of traits for describing properties on functions.

Trait Function$[\![X, Y]\!]$ is the base trait of functions that take a value of type X and return a value of type Y. Every function given to *dispatching* and *generate2* should extend trait Function to make the library check properties of the function. Basically, a function does not necessarily need to be a trait or an object in Fortress, since Fortress can handle functions directly. However, to describe properties of functions by themselves, we require a function to be a trait or an object. Otherwise, we need an extra argument in *generate2* to tell properties of functions.

Object IdFunction is the identity function that returns the given argument as its return value. Object IdFunction is used to cancel functions in the generate-and-test specification. If the user program does not need mapping of a function to elements, the object IdFunction must be given to method *generate2* of GGenerator (and function *dispatching*) to let the library know there is no function to map. Note that when a function object other than IdFunction is given to method *generate2*, the library determines that there is a function to map that can return a value different form the input value, even if the function actually returns always the input value itself.

5.2.3 Properties on Predicates

Figure 9 gives definitions of traits for describing properties on predicates, which are defined in Section 4.2.2.

Trait Predicate is the base of all predicates. It has method *judge* that returns a Boolean value (true or false) of the given argument x.

Trait ListPredicate is a predicate on lists. Every predicate given to method *generate2* of GGenerators or the dispatching table *dispatching* should extend trait ListPredicate.

Object TrueListPredicate is a predicate returning true for any input. This predicate is used to cancel the filtering in the generate-and-test specification, since filtering with the pred-

```
trait Function⟦X, Y⟧
  apply(x : X) : Y
end

object IdFunction⟦X⟧ extends Function⟦X, X⟧
  apply(x : X) : X = x
end

object SingletonFunction⟦X⟧ extends Function⟦X, List⟦X⟧⟧
  apply(x : X) : List⟦X⟧ = singleton⟦X⟧(x)
end

object FunctionWrapper⟦X, Y⟧(f : X → Y) extends Function⟦X, Y⟧
  apply(x : X) : Y = f x
end
```

Figure 8: Traits for function objects

icate is equals to the identity function. If the user program does not need filtering, the object TrueListPredicate must be given to method *generate2* of GGenerator (and function *dispatching*) to let the library know there is no filtering. Note that when a predicate object other than TrueListPredicate is given to method *generate2*, the library determines that there is a filtering with a predicate that can return false, even if the predicate actually returns always true.

Trait SuffixClosedPredicate indicates that a predicate extending SuffixClosedPredicate is suffix-closed. Similarly, trait PrefixClosedPredicate indicates prefix-closedness of a predicate. Since a segment-closed predicate is both suffix-closed and prefix-closed, SegmentClosedPredicate extends both SuffixClosedPredicate and PrefixClosedPredicate. Trait OverlapClosedPredicate indicates a predicate is overlap-closed.

Trait RelationalPredicate is the base trait of all relational predicates. Since a relational predicate is segment-closed and overlap-closed (Lemma 11 in Section 4.2.2), RelationalPredicate extends SuffixClosedPredicate and OverlapClosedPredicate. Method *related* corresponds to the relation that determines the predicate, and it returns true when the given arguments are related by the relation. Method *judge* uses *related* to check all pairs of consecutive elements are related by the relation.

Operator `AND` and object AndRelationalPredicate composes two relational predicates to make another relational predicate that returns true when both of the two predicates return true.

Example instances of RelationalPredicate are specified by comparison operators. Object Ascending is specified by $<$, and returns true when the given list is sorted in ascending order. Similarly, we can define objects Descending, WeaklyAscending, WeaklyDescending and Flat by $>$, $\leq$, $\geq$ and $=$. Object Smooth makes a relational predicate to check that the difference between any pair of consecutive elements is less than or equal to the given threshold c.

```
trait Predicate⟦E⟧
  judge(x : E) : Boolean
end
trait ListPredicate⟦E⟧ extends Predicate⟦List⟦E⟧⟧ end
object TrueListPredicate⟦E⟧ extends ListPredicate⟦E⟧
  judge(x : List⟦E⟧) : Boolean = true
end
trait SuffixClosedPredicate⟦E⟧ extends ListPredicate⟦E⟧ end
trait PrefixClosedPredicate⟦E⟧ extends ListPredicate⟦E⟧ end
trait SegmentClosedPredicate⟦E⟧
  extends {PrefixClosedPredicate⟦E⟧, SuffixClosedPredicate⟦E⟧}
end
trait OverlapClosedPredicate⟦E⟧ extends ListPredicate⟦E⟧ end
trait RelationalPredicate⟦E⟧
  extends {SegmentClosedPredicate⟦E⟧, OverlapClosedPredicate⟦E⟧}
  related(a:E, b:E) : Boolean
  judge(x : List⟦E⟧) : Boolean = do
    if x.size() ≤ 1 then
      true
    else
      sz = x.size() − 1
      (0 # sz).generate⟦Boolean⟧(AndReduction, fn i ⇒ related(x_i, x_{i+1}))
    end
  end
end
opr ∧⟦E⟧(p1 : RelationalPredicate⟦E⟧, p2 : RelationalPredicate⟦E⟧): RelationalPredicate⟦E⟧
  = AndRelationalPredicate⟦E⟧(p1, p2)
object AndRelationalPredicate⟦E⟧(p1 : RelationalPredicate⟦E⟧, p2 : RelationalPredicate⟦E⟧)
  extends RelationalPredicate⟦E⟧
  related(a:E, b:E) : Boolean = p1.related(a, b) ∧ p2.related(a, b)
end
object Ascending⟦E⟧ extends RelationalPredicate⟦E⟧
  related(a:E, b:E) : Boolean = a < b
end
object Descending⟦E⟧ extends RelationalPredicate⟦E⟧
  related(a:E, b:E) : Boolean = a > b
end
object WeaklyAscending⟦E⟧ extends RelationalPredicate⟦E⟧
  related(a:E, b:E) : Boolean = a ≤ b
end
object WeaklyDescending⟦E⟧ extends RelationalPredicate⟦E⟧
  related(a:E, b:E) : Boolean = a ≥ b
end
object Flat⟦E⟧ extends RelationalPredicate⟦E⟧
  related(a:E, b:E) : Boolean = a = b
end
object Smooth⟦E⟧(c:E) extends RelationalPredicate⟦E⟧
  related(a:E, b:E) : Boolean = |a − b| ≤ c
end
```

Figure 9: Traits for describing properties on predicates and example predicates

```
object InitsGenerator⟦E⟧(arglist : List⟦E⟧) extends GGenerator⟦E⟧
  getter list() : List⟦E⟧ = arglist
  getter defaultImplementation() : List⟦E⟧ → List⟦List⟦E⟧⟧ = initsImpl⟦E⟧
  generate2⟦R, F, M, N, P, Z, Y⟧(r : R, f : F, mr : M, mf : N, p : P) : Z
  = do
    dispatching⟦InitsGenerator⟦E⟧, R, F, M, N, P, Z, Y, E⟧(self, r, f, mr, mf, p)
  end
end

initsImpl⟦E⟧(x : List⟦E⟧) : List⟦List⟦E⟧⟧ = do
  x.generate⟦List⟦List⟦E⟧⟧⟧(InitsReduction⟦E⟧, fn (a) ⇒ singleton⟦List⟦E⟧⟧(singleton⟦E⟧(a)))
end

object InitsReduction⟦E⟧ extends Reduction⟦List⟦List⟦E⟧⟧⟧
  empty(): List⟦List⟦E⟧⟧ = emptyList⟦List⟦E⟧⟧()
  join(a : List⟦List⟦E⟧⟧, b : List⟦List⟦E⟧⟧): List⟦List⟦E⟧⟧ = do
    l = a.right().generate⟦List⟦E⟧⟧(Concat⟦E⟧, fn (x) ⇒ x);
    a.append(b.map⟦List⟦E⟧⟧(fn (x) ⇒ l.append(x)));
  end
end

inits⟦E⟧(x : List⟦E⟧) : GGenerator⟦E⟧ = InitsGenerator⟦E⟧(x)
```

Figure 10: Base definition of GGenerator inits

5.3 Implementation of GGenerator inits

This section gives implementations for GGenerator inits: an object for base implementation of inits and methods for efficient implementations of nested reductions given by theorems in Section 4.3.

5.3.1 Base Implementation

The base implementation of GGenerator inits is object InitsGenerator given in Figure 10. A related example problem is found in Section 3.3.1. A related theory is found in Section 4.3.1.

Object InitsGenerator takes the original list as its argument, and getter *list*() returns the given original list. Default implementation of GGenerator inits is defined outside the object InitsGenerator as function *initsImpl* shown below. Method *generate2* invokes function *dispatching* (Section 5.1.2) to dispatch suitable implementation to a user program specified by the given arguments.

Function *initsImpl* generates a list of prefix (initial) segments using method *generate* of the given list with reduction object InitsReduction shown below.

InitsReduction takes two lists of initial segments (a and b in the code), and returns a list of initial segments of the concatenated list. Suppose a is a list of initial segments of a list x, and b is that of a list y. InitsReduction makes a list of initial segments of the concatenated list $x.append(y)$ from a and b as follows. Initial segments of x are also initial segments of the concatenated list. So, all elements in a remain in the result. Each initial segment of y needs to be concatenated with x to become an initial segment of the concatenated list. So, InitsReduction maps a function to concatenate the last (rightmost) element of a (x is the last element of a) to each element of b so that it can become an initial segment of the concatenated list.

```
efficientImplTrueDistributive⟦R, F, M, N, P, Z, Y⟧
  (r : R, f : IdFunction⟦Y⟧, mr : LeftDistributiveOver⟦R⟧, mf : N, p : TrueListPredicate⟦E⟧)
 = do
  join(x, y) = do
    (i1, s1) = x
    (i2, s2) = y
    (r.join(i1, mr.join(s1, i2)), mr.join(s1, s2))
  end
  zero = (r.empty(), mr.empty())
  (r1, r2) = list().generate⟦(Z, Z)⟧(MapReduceReduction⟦(Z, Z)⟧(join, zero),
                                     (fn a ⇒ do b = mf.apply(a); (b, b) end))
  r1
end
```

Figure 11: Efficient implementation of nested reductions with distributive operators for GGenerator inits

5.3.2 Implementation for Nested Reductions with Distributive Operators

Figure 11 shows the efficient implementation *efficientImplTrueDistributive* for Theorem 15 in Section 4.3.2, which should be implemented in object InitsGenerator to be used in dispatching. A related example problem is found in Section 3.3.2.

The signature of *efficientImplTrueDistributive* is almost the same as method *generate2* of trait GGenerator. The types of some arguments are restricted to guarantee that the applicable condition of the theorem is satisfied by the arguments. Basically, restriction on types here is not necessary because the checking of types are performed before an invocation of this method. The type restriction in the code is added for safety.

Function *join* defined in *efficientImplTrueDistributive* is straightforward implementation of the new operator $\odot$ given in the theorem. Function *join* is applied to tuples. The first element of a tuple is the result of the nested reductions. The second element of the tuple is the result of a reduction with *mr*, which is used to improve efficiency by reuse of partial results. Value *zero* is the identity of the new operator, constructed from identities of the operators used in the original nested reductions. The reduction with the new operator is performed by *generate* of the original list (*list*() in the code). The body function given to *generate* is extended to apply a function *mf* before the reduction. The correctness of this extension is guaranteed by Theorem 14

To dispatch this implementation to a user program, we have to add an entry to `typecase` of the dispatching table (Section 5.1.2). The entry is shown below.

```
(InitsGenerator⟦E⟧, R, IdFunction⟦Y⟧, LeftDistributiveOver⟦R⟧, N, TrueListPredicate⟦E⟧)
  ⇒ g.efficientImplTrueDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
```

Since the condition for the theorem is that the inner reduction (*mr*) has left-distributivity over the outer reduction (*r*), the entry checks whether *mr* extends trait LeftDistributiveOver⟦*R*⟧. Also, it checks that *f* is the identity function and *p* is the true-predicate, since the theorem cannot deal with other functions and predicates.

5.3.3 Implementation for Filtering with Relational Predicates

Figure 12 shows the efficient implementation *efficientImplRelationalDistributive* for Corollary 17 (Theorem 16) in Section 4.3.3, which should be implemented in object InitsGenerator to be used in dispatching. A related example problem is found in Section 3.3.3.

```
efficientImplRelationalDistributive⟦R, F, M, N, P, Z, Y⟧
  (r : R, f : IdFunction⟦Y⟧, mr : LeftDistributiveOver⟦R⟧, mf : N, p : RelationalPredicate⟦E⟧)
 = do
  join(x, y) = do
    (i1, s1, h1, l1) = x
    (i2, s2, h2, l2) = y
    px = typecase (l1, h2) of
    (Just⟦E⟧, Just⟦E⟧) ⇒ p.related(l1.unJust(), h2.unJust())
    else⇒ true
    end
    if px then
      (r.join(i1, mr.join(s1, i2)), mr.join(s1, s2), takeleft(h1, h2), takeright(l1, l2))
    else
      (i1, r.empty(), takeleft(h1, h2), takeright(l1, l2))
    end
  end
  zero = (r.empty(), mr.empty(), Nothing⟦E⟧, Nothing⟦E⟧)
  (r1, r2, r3, r4) = list().generate⟦(Z, Z, Maybe⟦E⟧, Maybe⟦E⟧)⟧
                          (MapReduceReduction⟦(Z, Z, Maybe⟦E⟧, Maybe⟦E⟧)⟧(join, zero),
                           (fn a ⇒ do b = mf.apply(a); (b, b, Just⟦E⟧(a), Just⟦E⟧(a)) end))
  r1
end
```

Figure 12: Efficient implementation of nested reductions with distributive operators and filtering by relational predicate for GGenerator inits

The signature of *efficientImplRelationalDistributive* is almost the same as method *generate2* of trait GGenerator. The types of some arguments are restricted to guarantee that the applicable condition of the theorem is satisfied by the arguments. Basically, restriction on types here is not necessary because the checking of types are performed before an invocation of this method. The type restriction in the code is added for safety.

Function *join* defined in *efficientImplRelationalDistributive* is straightforward implementation of the new operator $\boxtimes$ given in the theorem. Function *join* is applied to quadruples. The first element of a quadruple is the result of the nested reductions. The second element of the quadruple is the result of a reduction with mr after filtering with p, which is used to improve efficiency by reuse of partial results. The third and the fourth elements are the edge elements of the input, which are used to check whether results from two recursions in divide-and-conquer computation can be combined to make a better solution. Value *zero* is the identity of the new operator, constructed from identities of the operators used in the original nested reductions. The reduction with the new operator is performed by *generate* of the original list (*list*() in the code). The body function given to *generate* is extended to apply a function mf before the reduction. The correctness of this extension is guaranteed by Theorem 14

To dispatch this implementation to a user program, we have to add an entry to `typecase` of the dispatching table (Section 5.1.2). The entry is shown below.

```
(InitsGenerator⟦E⟧, R, IdFunction⟦Y⟧, LeftDistributiveOver⟦R⟧, N, RelationalPredicate⟦E⟧)
  ⇒ g.efficientImplRelationalDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
```

Since the condition for the theorem is that the inner reduction (*mr*) has left-distributivity over the outer reduction (*r*) and the predicate (*p*) is relational, the entry checks whether *mr* extends trait LeftDistributiveOver⟦R⟧ and *p* extends trait RelationalPredicate. Also, it checks that *f* is the identity function, since the theorem cannot deal with other functions.

```
object TailsGenerator⟦E⟧(arglist : List⟦E⟧) extends GGenerator⟦E⟧
  getter list() : List⟦E⟧ = arglist
  getter defaultImplementation() : List⟦E⟧ → List⟦List⟦E⟧⟧ = tailsImpl⟦E⟧
  generate2⟦R, F, M, N, P, Z, Y⟧(r : R, f : F, mr : M, mf : N, p : P) : Z
   = do
     dispatching⟦TailsGenerator⟦E⟧, R, F, M, N, P, Z, Y, E⟧(self, r, f, mr, mf, p)
  end
end

tailsImpl⟦E⟧(x : List⟦E⟧) : List⟦List⟦E⟧⟧ = do
  x.generate⟦List⟦List⟦E⟧⟧⟧(TailsReduction⟦E⟧, fn (a) ⇒ singleton⟦List⟦E⟧⟧(singleton⟦E⟧(a)))
end

object TailsReduction⟦E⟧ extends Reduction⟦List⟦List⟦E⟧⟧⟧
  empty(): List⟦List⟦E⟧⟧ = emptyList⟦List⟦E⟧⟧()
  join(a : List⟦List⟦E⟧⟧, b : List⟦List⟦E⟧⟧): List⟦List⟦E⟧⟧ = do
    h = b.left().generate⟦List⟦E⟧⟧(Concat⟦E⟧, fn (x) ⇒ x);
    a.map⟦List⟦E⟧⟧(fn (x) ⇒ x.append(h)).append(b);
  end
end

tails⟦E⟧(x : List⟦E⟧) : GGenerator⟦E⟧ = TailsGenerator⟦E⟧(x)
```

Figure 13: Base definition of GGenerator tails

5.4 Implementation of GGenerator tails

This section gives implementations for GGenerator tails: an object for base implementation of tails and methods for efficient implementations of nested reductions given by theorems in Section 4.4.

5.4.1 Base Implementation

The base implementation of GGenerator tails is object TailsGenerator given in Figure 13. A related example problem is found in Section 3.4.1. A related theory is found in Section 4.4.1.

Object TailsGenerator takes the original list as its argument, and getter *list*() returns the given original list. Default implementation of GGenerator tails is defined outside the object TailsGenerator as function *tailsImpl* shown below. Method *generate2* invokes function *dispatching* (Section 5.1.2) to dispatch suitable implementation to a user program specified by the given arguments.

Function *tailsImpl* generates a list of suffix (tail) segments using method *generate* of the given list with reduction object TailsReduction shown below.

TailsReduction takes two lists of tail segments (a and b in the code), and returns a list of tail segments of the concatenated list. Suppose a is a list of tail segments of a list x, and b is that of a list y. TailsReduction makes a list of tail segments of the concatenated list $x.append(y)$ from a and b as follows. Tail segments of y are also tail segments of the concatenated list. So, all elements in b remain in the result. Each tail segment of x needs to be concatenated with y to become a tail segment of the concatenated list. So, TailsReduction maps a function to concatenate the first (leftmost) element of b (y is the first element of b) to each element of a so that it can become an tail segment of the concatenated list.

5.4.2 Implementation for Nested Reductions with Distributive Operators

Figure 14 shows the efficient implementation *efficientImplTrueDistributive* for Theorem 15 in Section 4.4.2, which should be implemented in object TailsGenerator to be used in dispatching.

```
efficientImplTrueDistributive⟦R, F, M, N, P, Z, Y⟧
  (r : R, f : IdFunction⟦Y⟧, mr : RightDistributiveOver⟦R⟧, mf : N, p : TrueListPredicate⟦E⟧)
= do
  join(x, y) = do
    (t1, s1) = x
    (t2, s2) = y
    (r.join(mr.join(t1, s2), t2), mr.join(s1, s2))
  end
  zero = (r.empty(), mr.empty())
  (r1, r2) = list().generate⟦(Z, Z)⟧(MapReduceReduction⟦(Z, Z)⟧(join, zero),
                                     (fn a ⇒ do b = mf.apply(a); (b, b) end))
  r1
end
```

Figure 14: Efficient implementation of nested reductions with distributive operators for GGenerator tails

A related example problem is found in Section 3.4.2.

The signature of *efficientImplTrueDistributive* is almost the same as method *generate2* of trait GGenerator. The types of some arguments are restricted to guarantee that the applicable condition of the theorem is satisfied by the arguments. Basically, restriction on types here is not necessary because the checking of types are performed before an invocation of this method. The type restriction in the code is added for safety.

Function *join* defined in *efficientImplTrueDistributive* is straightforward implementation of the new operator $\circledast$ given in the theorem. Function *join* is applied to tuples. The first element of a tuple is the result of the nested reductions. The second element of the tuple is the result of a reduction with *mr*, which is used to improve efficiency by reuse of partial results. Value *zero* is the identity of the new operator, constructed from identities of the operators used in the original nested reductions. The reduction with the new operator is performed by *generate* of the original list (*list*() in the code). The body function given to *generate* is extended to apply a function *mf* before the reduction. The correctness of this extension is guaranteed by Theorem 20.

To dispatch this implementation to a user program, we have to add an entry to `typecase` of the dispatching table (Section 5.1.2). The entry is shown below.

```
(TailsGenerator⟦E⟧, R, IdFunction⟦Y⟧, RightDistributiveOver⟦R⟧, N, TrueListPredicate⟦E⟧)
  ⇒ g.efficientImplTrueDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
```

Since the condition for the theorem is that the inner reduction (*mr*) has right-distributivity over the outer reduction (*r*), the entry checks whether *mr* extends trait RightDistributiveOver⟦*R*⟧. Also, it checks that *f* is the identity function and *p* is the true-predicate, since the theorem cannot deal with other functions and predicates.

5.4.3 Implementation for Filtering with Relational Predicates

Figure 15 shows the efficient implementation *efficientImplRelationalDistributive* for Corollary 23 (Theorem 22) in Section 4.4.3, which should be implemented in object TailsGenerator to be used in dispatching. A related example problem is found in Section 3.4.3.

The signature of *efficientImplRelationalDistributive* is almost the same as method *generate2* of trait GGenerator. The types of some arguments are restricted to guarantee that the applicable condition of the theorem is satisfied by the arguments. Basically, restriction on types here is not necessary because the checking of types are performed before an invocation of this method. The type restriction in the code is added for safety.

```
efficientImplRelationalDistributive⟦R, F, M, N, P, Z, Y⟧
  (r : R, f : IdFunction⟦Y⟧, mr : RightDistributiveOver⟦R⟧, mf : N, p : RelationalPredicate⟦E⟧)
 = do
  join(x, y) = do
    (t1, s1, h1, l1) = x
    (t2, s2, h2, l2) = y
    px = typecase (l1, h2) of
    (Just⟦E⟧, Just⟦E⟧) ⇒ p.related(l1.unJust(), h2.unJust())
    else⇒ true
    end
    if px then
      (r.join(mr.join(t1, s2), t2), mr.join(s1, s2), takeleft(h1, h2), takeright(l1, l2))
    else
      (t2, r.empty(), takeleft(h1, h2), takeright(l1, l2))
    end
  end
  zero = (r.empty(), mr.empty(), Nothing⟦E⟧, Nothing⟦E⟧)
  (r1, r2, r3, r4) = list().generate⟦(Z, Z, Maybe⟦E⟧, Maybe⟦E⟧)⟧
                          (MapReduceReduction⟦(Z, Z, Maybe⟦E⟧, Maybe⟦E⟧)⟧(join, zero),
                           (fn a ⇒ do b = mf.apply(a); (b, b, Just⟦E⟧(a), Just⟦E⟧(a)) end))
  r1
end
```

Figure 15: Efficient implementation of nested reductions with distributive operators and filtering by relational predicate for GGenerator tails

Function *join* defined in *efficientImplRelationalDistributive* is straightforward implementation of the new operator ⊠ given in the theorem. Function *join* is applied to quadruples. The first element of a quadruple is the result of the nested reductions. The second element of the quadruple is the result of a reduction with mr after filtering with p, which is used to improve efficiency by reuse of partial results. The third and the fourth elements are the edge elements of the input, which are used to check whether results from two recursions in divide-and-conquer computation can be combined to make a better solution. Value *zero* is the identity of the new operator, constructed from identities of the operators used in the original nested reductions. The reduction with the new operator is performed by *generate* of the original list (*list*() in the code). The body function given to *generate* is extended to apply a function mf before the reduction. The correctness of this extension is guaranteed by Theorem 20.

To dispatch this implementation to a user program, we have to add an entry to `typecase` of the dispatching table (Section 5.1.2). The entry is shown below.

```
(TailsGenerator⟦E⟧, R, IdFunction⟦Y⟧, RightDistributiveOver⟦R⟧, N, RelationalPredicate⟦E⟧)
  ⇒ g.efficientImplRelationalDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
```

Since the condition for the theorem is that the inner reduction (mr) has right-distributivity over the outer reduction (r) and the predicate (p) is relational, the entry checks whether mr extends trait RightDistributiveOver⟦R⟧ and p extends trait RelationalPredicate. Also, it checks that f is the identity function, since the theorem cannot deal with other functions.

```
object SegsGenerator⟦E⟧(arglist : List⟦E⟧) extends GGenerator⟦E⟧
  getter list() : List⟦E⟧ = arglist
  getter defaultImplementation() : List⟦E⟧ → List⟦List⟦E⟧⟧ = segsImpl⟦E⟧
  generate2⟦R, F, M, N, P, Z, Y⟧(r : R, f : F, mr : M, mf : N, p : P) : Z
  = do
    dispatching⟦SegsGenerator⟦E⟧, R, F, M, N, P, Z, Y, E⟧(self, r, f, mr, mf, p)
  end
end
segsImpl⟦E⟧(x : List⟦E⟧) : List⟦List⟦E⟧⟧ = do
  concat(tailsImpl(x).map⟦List⟦List⟦E⟧⟧⟧(initsImpl⟦E⟧))
end
segs⟦E⟧(x : List⟦E⟧) : GGenerator⟦E⟧ = SegsGenerator⟦E⟧(x)
```

Figure 16: Base definition of GGenerator segs

5.5 Implementation of GGenerator segs

This section gives implementations for GGenerator segs: an object for base implementation of segs and methods for efficient implementations of nested reductions given by theorems in Section 4.5.

5.5.1 Base Implementation

The base implementation of GGenerator segs is object SegsGenerator given in Figure 16. A related example problem is found in Section 3.5. A related theory is found in Section 4.5.1.

Object SegsGenerator takes the original list as its argument, and getter *list*() returns the given original list. Default implementation of *segs* generator is defined outside the object SegsGenerator as function *segsImpl* shown below. Method *generate2* invokes function *dispatching* (Section 5.1.2) to dispatch suitable implementation to a user program specified by the given arguments.

Function *segsImpl* generates a list of all segments (continuous subsequences) using naive implementation *initsImpl* (Section 5.3.1) and *tailsImpl* (Section 5.4.1).

5.5.2 Implementation for Nested Reductions with Distributive Operators

Figure 17 shows the efficient implementation *efficientImplTrueDistributive* for Theorem 26 in Section 4.5.2, which should be implemented in object SegsGenerator to be used in dispatching. A related example problem is found in Section 3.5.2.

The signature of *efficientImplTrueDistributive* is almost the same as method *generate2* of trait GGenerator. The types of some arguments are restricted to guarantee that the applicable condition of the theorem is satisfied by the arguments. Basically, restriction on types here is not necessary because the checking of types are performed before an invocation of this method. The type restriction in the code is added for safety.

Function *join* defined in *efficientImplTrueDistributive* is straightforward implementation of the new operator ⊙ given in the theorem. Function *join* is applied to quadruples. The first element of a quadruple is the result of the nested reductions. The second and the third elements of the quadruple are the results of the nested reductions on suffixes only and prefixes only, which are used to improve efficiency by reuse of partial results. The fourth element is the result of a reduction with *mr*. Value *zero* is the identity of the new operator, constructed from identities of the operators used in the original nested reductions. The reduction with the new operator is performed by *generate* of the original list (*list*() in the code). The body function

```
efficientImplTrueDistributive⟦R, F, M, N, P, Z, Y⟧
  (r : Commutative, f : IdFunction⟦Y⟧,
   mr : DistributiveOver⟦R⟧, mf : N, p : TrueListPredicate⟦E⟧)
 = do
  join(x, y) = do
    (m1, i1, t1, s1) = x
    (m2, i2, t2, s2) = y
    t = r.join(mr.join(t1, s2), t2)
    i = r.join(i1, mr.join(s1, i2))
    (r.join(r.join(m1, m2), r.join(t, i)), i, t, mr.join(s1, s2))
  end
  zero = (r.empty(), r.empty(), r.empty(), mr.empty())
  (r1, r2, r3, r4) = list().generate⟦(Z, Z, Z, Z)⟧(MapReduceReduction⟦(Z, Z, Z, Z)⟧(join, zero),
                                                   (fn a ⇒ do b = mf.apply(a); (b, b, b, b) end))
  r1
end
```

Figure 17: Efficient implementation of nested reductions with distributive operators for GGenerator segs

given to *generate* is extended to apply a function *mf* before the reduction. The correctness of this extension is guaranteed by Theorem 25.

To dispatch this implementation to a user program, we have to add an entry to `typecase` of the dispatching table (Section 5.1.2). The entry is shown below.

```
(SegsGenerator⟦E⟧, R, IdFunction⟦Y⟧, DistributiveOver⟦R⟧, N, TrueListPredicate⟦E⟧)
  ⇒ g.efficientImplTrueDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
```

Since the condition for the theorem is that the inner reduction (mr) has distributivity over the outer reduction (r) and the outer reduction is commutative, the entry checks whether mr extends trait DistributiveOver⟦R⟧ and r extends trait Commutative. Also, it checks that f is the identity function and p is the true-predicate, since the theorem cannot deal with other functions and predicates.

5.5.3 Implementation for Filtering with Relational Predicates

Figure 18 shows the efficient implementation *efficientImplRelationalCommutativeDistributive* for Theorem 27 in Section 4.5.3, which should be implemented in object SegsGenerator to be used in dispatching. A related example problem is found in Section 3.5.3.

The signature of *efficientImplRelationalCommutativeDistributive* is almost the same as method *generate2* of trait GGenerator. The types of some arguments are restricted to guarantee that the applicable condition of the theorem is satisfied by the arguments. Basically, restriction on types here is not necessary because the checking of types are performed before an invocation of this method. The type restriction in the code is added for safety.

Function *join* defined in *efficientImplRelationalCommutativeDistributive* is straightforward implementation of the new operator ⊡ given in the theorem. Function *join* is applied to hextuples. The first element of a hextuple is the result of the nested reductions. The second and the third elements of the hextuple are the results of the nested reductions on suffixes only and prefixes only, which are used to improve efficiency by reuse of partial results. The fourth element is the result of a reduction with *mr* after filtering with p. The fifth and the sixth elements are the edge elements of the input, which are used to check whether results from two recursions in divide-and-conquer computation can be combined to make a better solution. Value *zero* is the identity of the new operator, constructed from identities of the operators used in

```
efficientImplRelationalCommutativeDistributive⟦R, F, M, N, P, Z, Y⟧
  (r : Commutative, f : IdFunction⟦Y⟧,
   mr : DistributiveOver⟦R⟧, mf : N, p : RelationalPredicate⟦E⟧)
 = do
  join(x, y) = do
    (m1, t1, i1, s1, h1, l1) = x
    (m2, t2, i2, s2, h2, l2) = y
    px = typecase (l1, h2) of
    (Just⟦E⟧, Just⟦E⟧) ⇒ p.related(l1.unJust(), h2.unJust())
    else⇒ true
    end
    if px then
      (r.join(r.join(m1, m2), mr.join(t1, i2)), r.join(mr.join(t1, s2), t2),
       r.join(i1, mr.join(s1, i2)), mr.join(s1, s2),
       takeleft(h1, h2), takeright(l1, l2))
    else
      (r.join(m1, m2), t2, i1, r.empty(), takeleft(h1, h2), takeright(l1, l2))
    end
  end
  zero = (r.empty(), r.empty(), r.empty(), mr.empty(), Nothing⟦E⟧, Nothing⟦E⟧)
  (r1, r2, r3, r4, r5, r6) = list().generate⟦(Z, Z, Z, Z, Maybe⟦E⟧, Maybe⟦E⟧)⟧
                              (MapReduceReduction⟦(Z, Z, Z, Z, Maybe⟦E⟧, Maybe⟦E⟧)⟧(join, zero),
                               (fn a ⇒ do b = mf.apply(a); (b, b, b, b, Just⟦E⟧(a), Just⟦E⟧(a)) end))
  r1
end
```

Figure 18: Efficient implementation of nested reductions with distributive operators and filtering by relational predicate for GGenerator segs

the naive nested reductions. The reduction with the new operator is performed by *generate* of the original list (*list*() in the code). The body function given to *generate* is extended to apply a function *mf* before the reduction. The correctness of this extension is guaranteed by Theorem 25.

To dispatch this implementation to a user program, we have to add an entry to `typecase` of the dispatching table (Section 5.1.2). The entry is shown below.

```
(SegsGenerator⟦E⟧, Commutative, IdFunction⟦Y⟧, DistributiveOver⟦R⟧, N, RelationalPredicate⟦E⟧)
  ⇒ g.efficientImplRelationalCommutativeDistributive⟦R, F, M, N, P, Z, Y⟧(r, f, mr, mf, p)
```

Since the condition for the theorem is that the inner reduction (*mr*) has distributivity over the outer reduction (*r*), the outer reduction is commutative, and the predicate (*p*) is relational, the entry checks whether *mr* extends trait DistributiveOver⟦*R*⟧, *r* extends trait Commutative and *p* extends trait RelationalPredicate. Also, it checks that *f* is the identity function, since the theorem cannot deal with other functions.

(MIS)	$mis = \texttt{BIG MAX} \langle \sum y \mid y \leftarrow inits\ x \rangle$
(MAIS)	$mais = \texttt{BIG MAX} \langle \sum y \mid y \leftarrow inits\ x, ascending(y) \rangle$
(MTS)	$mts = \texttt{BIG MAX} \langle \sum y \mid y \leftarrow tails\ x \rangle$
(MATS)	$mats = \texttt{BIG MAX} \langle \sum y \mid y \leftarrow tails\ x, ascending(y) \rangle$
(MSS)	$mss = \texttt{BIG MAX} \langle \sum y \mid y \leftarrow segs\ x \rangle$
(MDSS)	$mdss = \texttt{BIG MAX} \langle \sum y \mid y \leftarrow segs\ x, descending(y) \rangle$

Figure 19: Programs for experiment

5.6 Experimnt Results

This section shows the power of automatic optimization with dispatching implementation. Figure 19 shows the target programs of the experiment. Since the desugaring process does not work well, we desugared comprehensions by hand.

Execution time of each program is measured on the current Fortress interpreter [For] in two cases. In the first case (case "naive"), the library does not know any theorems and dispatches naive implementations to programs. In the second case (case "efficient"), the library knows theorems and dispatches accompanying efficient implementations to programs.

Measured execution time (an average of eight executions) is shown in Table 1. The measurement is performed on a PC with two quadcore CPUs (Intel®Xeon®E5430, total 8 cores), 8GB memory, and Linux 2.6.22. The execution time contains startup time of a Fortress program, as well as the execution time of the program code.

The ratio of execution time of two cases shows the power of the optimization by dispatching, although the absolute speed is not meaningful because the interpreter is still under active development. Dispatching the efficient implementation, the library succeeded in improving the efficiency of the user program to achieve ten times faster execution time. So, a user program naively written with our GG library can run efficiently.

Table 1: Execution time and relative speed of dispatched naive/efficient implementation

problem	case	input size	execution time (s)	relative speed (naive / x)
MIS	naive	1000	19.72	1.00
		2000	71.49	1.00
	efficient	1000	0.40	49.30
		2000	0.46	155.41
MAIS	naive	1000	31.53	1.00
		2000	117.86	1.00
	efficient	1000	0.50	63.06
		2000	0.61	193.21
MTS	naive	1000	20.88	1.00
		2000	76.54	1.00
	efficient	1000	0.40	52.20
		2000	0.51	150.07
MATS	naive	1000	32.92	1.00
		2000	127.47	1.00
	efficient	1000	0.49	67.18
		2000	0.63	202.33
MSS	naive	100	17.51	1.00
		200	92.00	1.00
	efficient	100	0.35	50.02
		200	0.35	262.85
MDSS	naive	100	22.02	1.00
		200	125.37	1.00
	efficient	100	0.35	62.91
		200	0.35	358.20

6 Conclusion

This report provides collections for enriching the power of "GG library." It provides a collection of GGenerators with example problems, a collection of theories on GGenerators for efficient implementations with formal discussion, and a collection of their actual implementation in GG library on Fortress.

The library will grow with further collections of GGenerators and theories.

Acknowledgments

This report is a partial result of a joint research project "Development of a library based on skeletal parallel programming in Fortress" with Sun Microsystems. We would like to thank the members of Project Fortress, especially, Guy L. Steele Jr. and Jan-Willem Maessen for fruitful discussions on this research.

References

[ACH+] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 1.0 β. available on web: http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf.

[Bir87] Richard S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[Bir98] Richard S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Series in Computer Science. Prentice Hall, 2nd edition, April 1998.

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

[Ble90] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[EHK+08] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki, and Masato Takeichi. Generator-based GG Fortress library. Technical Report METR2008–16, Department of Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo, 2008.
available on web: http://www.keisu.t.u-tokyo.ac.jp/research/techrep/.

[For] Project Fortress. The reference interpreter for the Fortress language.
available on web: http://projectfortress.sun.com/Projects/Community.

[Gor97] Sergei Gorlatch. Optimizing compositions of scans and reductions in parallel program derivation. Technical Report MPI–9711, Universität Passau, 1997.

[Jeu93] Johan Theodoor Jeuring. *Theories for Algorithm Calculation*. PhD thesis, Faculty of Science, Utrecht University, 1993.

[SHT01] Isao Sasano, Zhenjiang Hu, and Masato Takeichi. Generation of efficient programs for solving maximum multi-marking problems. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 72–91, London, UK, 2001. Springer-Verlag.

[SHTO00] Isao Sasano, Zhenjiang Hu, Masato Takeichi, and Mizuhito Ogawa. Make it practical: a generic linear-time algorithm for solving maximum-weightsum problems. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 137–149, New York, NY, USA, 2000. ACM.

[Ski90] David B. Skillicorn. Architecture-independent parallel computation. *Computer*, 23(12):38–50, 1990.

[Zan92] Hans Zantema. Longest segment problems. *Science of Computer Programming*, 18(1):39–66, 1992.

[Zha02] Haiyan Zhao. *A Compositional Approach to Mining Optimal Ranges*. PhD thesis, Department of Information Engineering, University of Tokyo, 2002.