MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

Generator-based GG Fortress Library

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

METR 2008–16 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

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

This report proposes a new library on Fortress to deal with computation with complex dependency such as prefix sums, which cannot be efficiently dealt with by simple comprehensions or generator-reduction patterns. The library provides a set of generator-of-generators that abstract generation of nested data structures, for allowing users to write their programs in an easy and uniform way. The library also provides an automatic optimization mechanism that dispatches correct and efficient implementation to those user programs. Thus, users can easily make correct parallel programs without losing efficiency. The proposed library is implemented on Fortress, and techniques used here can be reused for other libraries on Fortress.

1 Introduction

Consider the following simple problem "Maximum Prefix Sum Problem."

Given a sequence of numbers, find the maximum sum of all prefix segments of the sequence.

For example, the maximum prefix sum of a sequence $[2, -1, 3, -2, 1]$ is 4, because its prefix segments are $[2]$, $[2, -1]$, $[2, -1, 3]$, $[2, -1, 3, -2]$ and $[2, -1, 3, -2, 1]$, and their sums are 2, 1, 4, 2 and 3, respectively. This problem is an instance of the following computation often seen in various scientific computations.

$$\sum_{i \in [1,\ldots,n]}{}_{\oplus} \sum_{j \in f(i)}{}_{\otimes} x_j$$

Here, $f(i)$ returns some region depending on i, and $\sum_{\oplus}$ takes a summation with associative binary operator $\oplus$ instead of the usual plus operator $+$. Clearly, "Maximum Prefix Sum Problem" can be written in this form with maximum operator $\uparrow$ and the usual plus operator $+$:

$$\begin{aligned}&\sum_{i \in [1,2,3,4,5]}{}_{\uparrow} \sum_{j \in [1,\ldots,i]}{}_{+} x_j \\ &\quad = \quad (x_1) \uparrow (x_1 + x_2) \uparrow (x_1 + x_2 + x_3) \uparrow (x_1 + x_2 + x_3 + x_4) \uparrow (x_1 + x_2 + x_3 + x_4 + x_5)\ .\end{aligned}$$

Here, the dependency of j to i is a prefix $[1, \ldots, i]$.

This problem can be solved straightforwardly in Fortress. One way is to use two for-loops in Fortress [ACH+] as follows.

```
m : ℤ32 = −infinity
for i ← 0 # x.size() do
  s: ℤ32 = 0
  for j ← 0 # (i + 1) do
    atomic s += x_j
  end
  atomic m := m MAX s
end
```

Another way is to use comprehensions to take the maximum and summations:

$$\texttt{BIG MAX}\,[\,\textstyle\sum[\,x_j \mid j \leftarrow 0\,\#\,(i+1)]\mid i \leftarrow 0\,\#\,x.size()\,]$$

In both cases, the naive program is straightforward implementation of the problem statement, which is very easy to write and understand.

However, the above naive programs are inefficient. In fact, the following efficient implementation[1] using only one loop is known for "Maximum Prefix Sum Problem," while naive programs use two loops.

```
opr MPS(a, b) = do
  (m1, s1) = a
  (m2, s2) = b
  (m1 MAX (s1 + m2), s1 + s2)
end
(r1, r2) = BIG MPS [ (a, a) | a ← x]
r1
```

This program uses the dependency between prefixes and distributivity of the plus operator + over the maximum operator ↑, to compute the solution in one loop. Moreover, since associativity of the binary operator `MPS` used in this efficient program is guaranteed by the distributivity, this efficient program is a correct parallel program.

However, there exist some problems in asking users to write this efficient program.

1. It is difficult to understand what the efficient program computes in the loop. This difficulty results in increased occurrence of bugs in the program, so maintenance of this program becomes difficult and productivity gets lower.

2. It is difficult to change the efficient program to solve a slightly different problem. For example, consider a problem to find the maximum sum of prefixes in which numbers are ordered ascendingly. It is not clear how a user should change the efficient program to solve this slightly changed problem, since the resulting program requires associativity of a binary operator used in the program for parallel computation.

3. It is a heavy burden for a user to know such efficient implementation with its applicable condition, since there are many variants of implementation for a class of problems. So, when a user uses efficient implementation for his/her problem, the implementation may be neither the best implementation nor applicable to the problem by lacking some required condition.

Thus, asking users to write complicated efficient programs increases burdens on users and decreases productivity. The best way is to let users write clear programs, and programming environments dispatch the best efficient implementation for user programs.

Writing clear programs, on the other hand, needs abstraction mechanism for flexibility. The naive programs shown above are not flexible enough. For example, to find the maximum sum of any segments instead of prefix segments, a user has to add another for-loop to the naive program.

[1]The definition of `BIG MPS` is omitted here for readability. Please refer to Figure 2-(c) for its definition.

This is not a small change. Moreover, to find the maximum sum of any subsequences, a user has to change the program drastically because generation of all subsequences is difficult to write with a finite number of for-loops. Those changes of the structure of programs are expected to be hidden from users. Thus, an interface abstracting those generations of nested data structures is required for describing programs uniformly and for development of clear programs.

The ideal solution to solve these problems is to provide both an interface to describe programs easily and uniformly by abstracting generation of nested data structures, and a programming environment that dispatches the best and correct implementation to user programs written with the interface.

We propose *GG library* (*GG* stands for *Generator-of-Generators*) to achieve the ideal solution. For example, using our library, a user can write a naive program for "Maximum Prefix Sum Problem" clearly as follows[2].

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

This program is very clear, since generation of prefix segments is abstracted by GGenerator *inits* [3] (*inits* stands for initial segments). A user can easily change the program to find the maximum sum of any segments by replacing *inits* with another generator-of-generators (GGenerator for short) *segs* [4] that abstracts generation of all segments.

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

Also, a user can easily change the program to find the maximum sum of ascending prefixes by adding predicate *ascending*[5].

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

Once written with our library, those programs are provided with the best correct implementation in the collection of implementation in the library. So, the efficient implementation shown above is dispatched to the first program written with GGenerator *inits*.

Features of our library are shown below.

- 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. All examples shown above are covered by this specification.

- 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.

[2]Current implementation of GG library handles List that is denoted by $\langle|\ldots|\rangle$. But we use the notation $[\ldots]$ in this report for readability.

[3]For example, *inits* $[2,-1,3,-2,1]$ results in [[2], $[2,-1]$, $[2,-1,3]$, $[2,-1,3,-2]$, $[2,-1,3,-2,1]$]

[4]For example, *segs* $[2,-1,3,-2,1]$ results in [[2], $[2,-1]$, $[2,-1,3]$, $[2,-1,3,-2]$, $[2,-1,3,-2,1]$, $[-1]$, $[-1,3]$, $[-1,3,-2]$, $[-1,3,-2,1]$, [3], $[3,-2]$, $[3,-2,1]$, $[-2]$, $[-2,1]$, [1]]

[5]For example, *ascending* $[1,2,5]$ result in true, but *ascending* $[1,5,2]$ results in false

- 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.

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.

The rest of this report is organized as follows. Section 2 shows the structure and the behavior of our GG library. Section 3 shows the implementation details of the mechanism of GG library. Section 4 shows how GG library grows. Section 5 concludes this report.

2 Structure and Behavior of GG Library

In this section, we will explain the structure and the behavior of our library to dispatch efficient implementation to a user program. Details of implementation techniques used in the library are shown in Section 3 and Section 4.

2.1 Structure of GG Library

Figure 1 illustrates the structure of our GG Library. There are two kinds of collections in the library. One is a collection of GGenerators that are used to describe specification of problems. This collection of GGenerators provides an interface for easy and uniform description of user programs. Each GGenerator abstracts generation of a nested data structure. For example, generation of initial segments is abstracted by GGenerator *inits*, and generation of all segments is abstracted by GGenerator *segs*.

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 includes a default implementation that is dispatched when no efficient implementation can be used for the given user program.

Besides those two specific collections, the library has a collection of traits to describe mathematical properties of user programs. Some of them have already been defined in the Fortress specification [ACH+]. We add extra traits in our library to describe mathematical properties not given in the Fortress specification. Those traits are used to determine whether a given user program satisfies applicable conditions of theories.

Figure 1 also shows 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." The behavior of GG Library for dispatching implementation to a user program is separated into two phases.

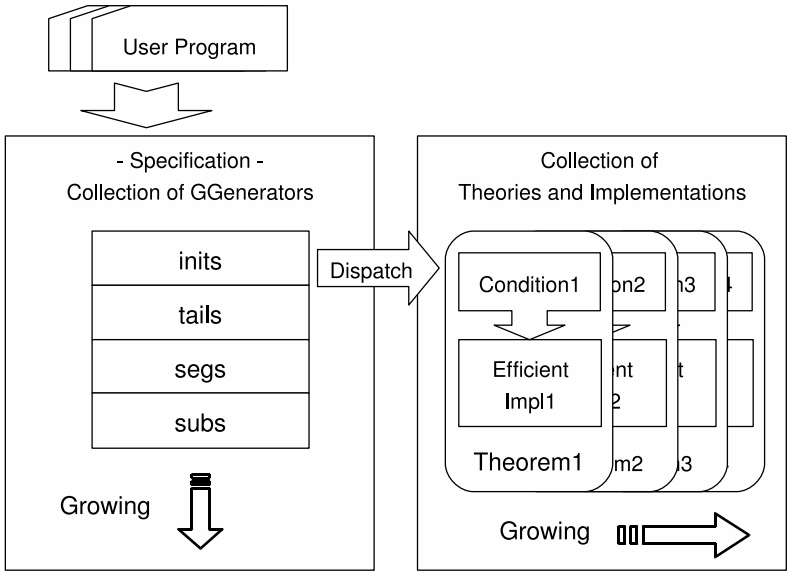


Figure 1: Structure of GG library

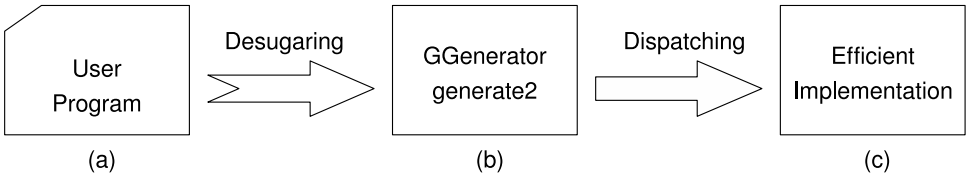
Phase 1. *Desugaring a user program into invocations of method generate2 of GGenerators*
In the first phase, the library desugars a user program written with for-loops or comprehensions into invocations of method *generate2* of GGenerators used in the program. Method *generate2* is the most important method of trait GGenerator, which is the base trait of all GGenerators, to perform nested reductions on generated nested data structures. Basically, users do not need to invoke method *generate2* directly to perform nested reductions. They have freedom to write programs for nested reductions with comprehensions or for-loops.

Figure 2-(a) shows an example of such user programs written with comprehensions. This program uses GGenerator *inits* to generate prefix segments of the input sequence x. The inner reduction of the nested reduction takes a summation of y that is a generated prefix of x. The outer reduction takes the maximum of those summations.

The desugaring process transforms the user program (Figure 2-(a)) into the program shown in Figure 2-(b). In the desugared program, nested reductions written with comprehensions are replaced with an invocation of method *generate2* of GGenerator *inits*. Two reduction operators + and `MAX` used in the original comprehensions are given to the method *generate2* as its arguments enclosed in objects: SumReductionZZ32 and MaxReductionZZ32. The other arguments of the method *generate2* are default values such as IdFunction (the identity function) and TrueListPredicate (the predicate that is always true).

Phase 2. *Dispatching implementation within the invocation of method generate2*
After the desugaring process, the library dispatches implementation within the invocation of method *generate2* of a GGenerator. 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 with the given arguments. Otherwise, the library will use the default naive implementation for the nested reductions.

To tell properties of the arguments to the library, the operator/predicate/function given to the method *generate2* should extend a set of suitable traits provided by the library or Fortress when it has some mathematical properties. For example, the usual



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

(a) User program of nested reductions with comprehensions

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

(b) Desugared program

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

(c) Efficient implementation for distributive operators

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

plus operator + has distributivity over the maximum operator `MAX`, so SumReduction should extend LeftDistributiveOver[[MaxReduction]] to tell its distributivity. Since properties of the given arguments are specified by traits, checking of applicable conditions of theories are performed by checking types of the arguments. Basically, the checking of types is realized by overloading or `typecase`.

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 trait LeftDistributiveOver[[MaxReductionZZ32]] to indicate its distributivity. Thus, the library uses the efficient implementation shown in Figure 2-(c) to perform the nested reduction of the program. Conversely, if the object SumReduction did not extend trait LeftDistributiveOver[[MaxReductionZZ32]], the library would use the naive implementation, shown in the previous section, as the default implementation for the nested reduction.

3 Implementation Details of GG Library

In this section, we will explain the details of implementation of GG Library. First, we will explain trait GGenerator and the dispatching process that are the core of GG Library. After that, we will explain the desugaring process.

3.1 **Trait** GGenerator

Trait GGenerator is the base trait of all GGenerators. Figure 3 shows the definition of trait GGenerator (the figure contains only essential methods). Currently, GGenerator$[\![E]\!]$ is defined as a subtrait of trait List$[\![$List$[\![E]\!]]\!]$, which is a subtrait of Fortress's Generator shown in Figure 4. 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.

3.1.1 **Methods of Trait** GGenerator

Figure 3 defines trait GGenerator that has five methods.

Getter *list*() returns the original list given to the GGenerator. Basically, values in 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

```
(* base trait of generator-of-generators *)
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: The base trait of GGenerators

```
trait Generator⟦E⟧ excludes {Number}
    generate⟦R⟧(r: Reduction⟦R⟧, body: E → R): R
end
```

Figure 4: The core of trait Generator in Fortress

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.

```
generate2⟦R,F,M,N,P,Z,Y⟧(r : R, f : F, mr : M, mf : N, p : P) : Z
  where {R extends Reduction⟦Z⟧, F extends Function⟦Y,Z⟧,
         M extends Reduction⟦Y⟧, N extends Function⟦E,Y⟧,
         P extends ListPredicate⟦E⟧}
  = defaultgeneration⟦R,F,M,N,P,Z,Y⟧(r, f, mr, mf, p)
```

3.1.2 An Example GGenerator *inits*

As a concrete example of GGenerators, Figure 5 gives the definition of GGenerator *inits* as object InitsGenerator.

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 object InitsGenerator as function *initsImpl* shown below. Method *generate2* invokes function *dispatching* (explained in the next section) to dispatch suitable implementation to a user program specified by the given arguments. If there is no efficient implementation available for the arguments, the default implementation explained below will be used.

```
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 5: Implementation of GGenerator *inits*

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.

Finally, function *inits* is defined for shortcuts to constructing an instance of InitsGenerator.

3.2 Dispatching Efficient Implementation

In this section, we first introduce an efficient implementation of the nested reduction for GGenerator *inits*. After that, we explain dispatching process to execute a user program with the efficient implementation.

3.2.1 Preparing Efficient Implementation

We introduce a theorem that gives us efficient implementation of nested reductions for GGenerator *inits*. The theorem says the nested reductions with two operators can be performed by one reduction with another new operator if the operator of the inner reduction has distributivity over the operator of the outer reduction.

Theorem 1 (Maximum Initial-segment Sum) *Provided that* $\oplus$ *is associative, and* $\otimes$ *is*

```
efficientImplTrueDistributive⟦R, F, M, N, P, Z, Y⟧
        (r : R, f : IdFunction⟦Y⟧,
         mr : DistributiveOver⟦R⟧, mf : IdFunction⟦E⟧,
         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 ⇒ (a, a)))
  r1
end
```

Figure 6: Efficient implementation of the nested reductions for GGenerator *inits*

associative and left-distributive over $\oplus$*, the following equation holds.*

$$\bigoplus [\ \bigotimes [\ a \mid a \leftarrow y\ ] \mid y \leftarrow \mathsf{inits}\, x\ ] \quad = \quad \mathit{first}\ (\bigodot [\ (a,a) \mid a \leftarrow x\ ]\ ) \quad \textbf{where}\ \ (i_1,s_1) \odot (i_2,s_2) = (i_1 \oplus (s_1 \otimes i_2), s_1 \otimes s_2),\ \ \mathit{first}\ (a,b) = a$$

Please refer to the complementary report [EHK+08] for proof of this theorem and theoretical backgrounds.

The new reduction with the new operator ($\odot$) is taken on pairs of values. Each pair records the result of the nested reductions of an input and the result of the inner reduction on the input, which is used to reuse partial results effectively in the divide-and-conquer parallel computation. For example, when we apply this theorem to the program for "Maximum Prefix Sum Problem," the first value of a pair is the maximum prefix sum of an input sequence, and the second value is the summation of the input. The computational cost (work) of the new reduction is $O(n)$ for an input of length n, while the cost of the naive nested reductions is $O(n^2)$. Thus, the use of the new reduction instead of the naive nested reduction improves efficiency of a user program.

To use the knowledge of the theorem, we have to implement the optimization given by the theorem. Figure 6 shows the efficient implementation *efficientImplTrueDistributive* for the theorem, which should be implemented in object InitsGenerator to be used in dispatching. 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 in the dispatching shown in the next section. 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. Value *zero* is the identity of the operator, constructed from identities of the operators used in the naive nested reductions. The reduction with the new operator is performed by *generate* of the original list (*list*() in the code) stored in the GGenerator *inits*. The body function given to *generate* makes a tuple as shown in the theorem.

Now, we have prepared an efficient implementation for dispatching. The next section shows how to dispatch this efficient implementation to a user program.

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

Figure 7: Dispatching table

3.2.2 Dispatching Table

The dispatching process dispatches efficient implementation given by a theorem to an invocation of *generate2* of a GGenerator, when the arguments satisfy the applicable condition of the theorem. Basically, checking of applicable conditions of theorems is performed by checking types of the arguments, since properties of the arguments should be specified by traits.

Figure 7 shows the dispatching table that is the core of the dispatching process. The dispatching table is defined as function *dispatching* that checks types of the given arguments and selects 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* as in Figure 5.

Function *dispatching* checks types of arguments by `typecase` of Fortress. Basically, one case of `typecase` corresponds to one theorem, and the type constraints correspond to the applicable condition of the theorem. The default (`else`) case corresponds to naive implementation of nested reductions. Figure 7 contains a case for Theorem 1 and the default case. For the default case, the dispatching table invokes method *defaultgeneration* of GGenerator g. In the case for Theorem 1, it checks whether properties of the arguments satisfy the condition by checking types of the arguments. Since the condition for the theorem is that the inner reduction (mr) has distributivity over the outer reduction (r), function *dispatching* checks whether mr extends trait LeftDistributiveOver⟦R⟧. Also, function *dispatching* checks whether f is the identity function and p is the true-predicate, 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 (shown in the previous section) to perform the nested reductions of a user program efficiently by the implementation.

Note that every function given to *dispatching* and *generate2* should extend trait Function shown in Figure 8 to be checked whether the function is the identity function or not. Also note that every predicate given to *dispatching* should extend trait ListPredicate shown in Figure 8 to be checked whether the predicate is the true-predicate or not. Basically, a function or a predicate (a function returning a Boolean value) 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.

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

```
trait Function[[X, Y]]
  apply(x : X) : Y
end

object IdFunction[[X]] extends Function[[X, X]]
  apply(x : X) : X = x
end

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
```

Figure 8: Traits for functions and predicates

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 the applicable condition of the theorem, and by adding 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.

3.3 Desugaring

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.

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

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.

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

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

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

Here, $ys \leftarrow \mathit{filter}\mathit{ascending}\ (\mathit{inits}\ xs)$ 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., $fgg\ xs = \mathit{filter}\ p\ (gg\ xs)$. 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.

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

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*.

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.

4 Growing Library

This section shows how the library grows in two directions. Also, we will give some experiment results to demonstrate the power of the library.

Further discussion on generators and theories for efficient implementations is found in the complementary report [EHK+08].

4.1 Growing in Expressiveness

One directions of growing is extension of the expressiveness to cover a wider range of problems. This is done by adding a new GGenerator.

For example, we can make a new GGenerator *tails* that abstracts generation of suffix segments[6], as shown in Figure 9. Behavior of default implementation of *tails* is similar to that of *inits*. The default implementation of *tails* is given as function *tailsImpl* that performs a single reduction with object TailsReduction. The operator of the reduction is almost the same as that of InitsReduction, except that it concatenates the first element of the second argument to each element of the first argument. Function *tailsImpl* is set to the getter *defaultImplementation* to be used in method *defaultgeneration* for the default naive implementation of nested reductions. Method *generate2* invokes function *dispatching* to make dispatching process work. Arguments of *dispatching* are GGenerator *tails* itself and the given arguments of *generate2*. Now, users can use GGenerator *tails* to describe their programs for suffix computations. For example, a program for "Maximum Suffix Sum Problem," of which objective is to find the maximum sum of a suffix of an input, is given as follows.

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

Similarly, we can add another GGenerator *segs* that abstracts generation of all segments (continuous subsequences) of the input. Its definition is shown in Figure 10. The default implementation of *segs* is given as function *segsImpl*. Function *segsImpl* uses default implementation

[6] For example, $\mathit{tails}\,[2,-1,3,-2,1]$ results in [$[2,-1,3,-2,1]$, $[-1,3,-2,1]$, $[3,-2,1]$, $[-2,1]$, $[1]$]

```
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 9: Base implementation of GGenerator *tails*

```
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
```

Figure 10: Base implementation of GGenerator *segs*

of *inits* and *tails* since each segment is a prefix of a suffix of an input sequence. Similar to GGenerator *tails*, method *generate2* invokes function *dispatching* to make dispatching process work. For example, a program for "Maximum Segment Sum Problem" is given as follows.

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

Since the default implementation of *tails* or *segs* is written with method *generate* of Fortress' Generator with associative operators, those programs are correct parallel programs.

Theories of GGenerator *tails* and GGenerator *segs* for optimization are found in the complementary report [EHK+08].

4.2 Growing in Optimization Power

The other direction of growing is extension of the power of automatic optimization by adding new knowledge of theories.

For example, we can extend GGenerator *inits* by the following theorem to support the case of filtering with non-true predicate p. Readers do not need to understand completely this

```
efficientImplRelationalDistributive⟦R, F, M, N, P, Z, Y⟧
        (r : R, f : IdFunction⟦Y⟧,
         mr : DistributiveOver⟦R⟧, mf : IdFunction⟦E⟧,
         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 ⇒ (a, a, Just⟦E⟧(a), Just⟦E⟧(a))))
  r1
end
```

Figure 11: Efficient implementation of reduction with a predicate in GGenerator *inits*

theorem, but this theorem says that we can perform the nested reductions with filtering by one reduction, if operators have distributivity and the filtering predicate p is of specific class called "relational predicate"[7]. The new operator used in the replaced reduction manipulates quadruples that are extension of tuples used by efficient implementation for "Maximum Prefix Sum Problem" to keep edge elements.

Theorem 2 (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.*

$$\bigoplus [\ \bigotimes [\ a \mid a \leftarrow y\ ] \mid y \leftarrow \mathsf{inits}\, x,\ p\ y\ ] \ = \ \mathit{first}\ (\bigodot [\ (a,a,a,a) \mid a \leftarrow x\ ])$$

where $(i_1, s_1, h_1, l_1) \odot (i_2, s_2, h_2, l_2) = (i_1 \oplus (s_1 \otimes i_2)_{l_1,h_2}, (s_1 \otimes s_2)_{l_1,h_2}, h_1 \ll h_2, l_1 \gg l_2)$

$\mathit{first}\ (a, b, c, d) = a$

$(a)_{l,h} = \mathbf{if}\ p\ ([l, h])\ \mathbf{then}\ a\ \mathbf{else}\ \imath_\oplus$

Please refer to the complementary report [EHK+08] for proof of this theorem and theoretical backgrounds.

Simply coding the new operator, we have a new efficient implementation in GGenerator *inits* shown in Figure 11. The new operator is defined as function *join* in the code. Value *zero* is the identity of the new operator, in which identities of the original operators are used. The reduction with the new operator is performed by method *generate* of the original list.

To check the applicable condition of the theorem, the library has to know the given predicate is of "relational predicate" or not. So, we introduce trait RelationalPredicate to indicate that a predicate extending RelationalPredicate is a relational predicate. The definition and an

[7] A relational predicate returns true if every pair of consecutive elements in the input satisfies a given relation.

```
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
object Ascending⟦E⟧ extends RelationalPredicate⟦E⟧
  related(a:E, b:E): Boolean = a < b
end
```

Figure 12: Properties of predicates and an example predicate

example of RelationalPredicate is shown in Figure 12. A relational predicate returns true if every pair of consecutive elements in the input satisfies the given relation *related*. An example of relational predicates is *ascending* that returns true if the given list is sorted in ascending order. Since a relational predicate has many properties, trait RelationalPredicate extends other traits SegmentClosedPredicate and OverlapClosedPredicate. Formal definitions of those properties are found in the complementary report [EHK+08].

To dispatch the efficient implementation to user programs, we need to modify the dispatching table as shown in Figure 13. In the dispatching table, the applicable condition of the theorem is checked against the arguments by `typecase`. Since the required condition is that the predicate is relational and operators have distributivity, the new entry (the second case) of the table checks whether the predicate p extends trait RelationalPredicate and the reduction operator mr extends trait LeftDistributiveOver⟦R⟧.

Now, a user program using *inits* with filtering by a predicate can receive the efficient implementation shown above. For example, the following program to find the maximum sum of ascending prefixes is executed by the single reduction, since the predicate *ascending* belongs to the specific class "relational predicate."

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

The effect of dispatching the efficient implementation is shown in the next section by experiment results.

4.3 Experiment Results

This section shows the power of automatic optimization with dispatching implementation. The following program is the target of the experiment.

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

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

Figure 13: Extended dispatching table

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

dispatched impl.	size of input	execution time (s)	relative speed (naive impl. / x)
naive	1000	35.3	1.00
	2000	126.0	1.00
efficient	1000	2.6	13.57
	2000	3.1	40.64

Since the desugaring process does not work well, we desugared this comprehension by hand. Execution time of the program is measured on the current Fortress interpreter [For] in two cases. In the first case, the library dispatches the naive implementation to the program by using the non-modified dispatching table shown in Figure 7. In the second case, the dispatching of the efficient implementation shown in Figure 11 works by using the modified dispatching table shown in Figure 13.

Measured execution time 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.

5 Conclusion

In this report, we proposed "GG library" that supports easy development of correct and efficient parallel programs. Users can write naive generate-and-test programs easily and uniformly with *GGenerators* (generator-of-generators) that abstracts generation of nested data structures. The library has an automatic optimization mechanism that automatically dispatches efficient implementation to a user program written with GGenerators based on a collection of theories. The library itself can easily grow up to cover a wider range of problems and to achieve better optimization power.

GGenerator is a natural extension of Generator, which is one of the core features of Fortress, to support efficient nested reductions with different operators. By checking whether operators/functions/predicates used in a user program extend a set of traits for required properties, the library determines whether applicable condition of the efficient implementation is satisfied or not. When the library finds that the user program satisfies the applicable condition, it dispatches the efficient implementation to the user program. The power of optimization by dispatching was shown by experimental results with prototype implementation.

Currently, the special desugaring process for GGenerators does not completely work. It is a part of our future work. Another part of future work is to grow the library, by enriching the collection of GGenerators and the collection of 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.

[EHK+08] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki, and Masato Takeichi. Generator-based GG Fortress library —collection of GGs and theories—. Technical Report METR2008–17, 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.