Saarland University

Department of Computer Science

Bachelor's Thesis

Implementation of a Method for Automatic Parallelization of Dynamic Programming in R-programs using Data-parallel Programming Patterns

submitted by
David Pfaff

submitted
July 25, 2012

Thesis Supervisor:
Prof. Dr. Sebastian Hack

Thesis Advisor:
Dr. Frank Padberg

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum/Date) (Unterschrift/Signature)

Contents

1. Introduction

R is a programming language and software environment for statistical computing and is widely used in statistics and bio-informatics. In these areas it is common to work with large sets of data. This makes scalability and processing speed an important factor. However, since R is an interpreted language, it does not natively support parallelism. While there are a number of extensions to R (cf. [19]) that allow a programmer to exploit parallelism, they are not easily applicable to existing projects, since they require the programmer to adapt the program. This restructuring process is prone to errors which are specific to the parallel extension and irrelevant to the original application domain. A different solution would be the application of a tool which automatically parallelizes sequential code. This approach has the advantage that the programmer only has to correctly apply the tool. In this case, the responsibility for providing a correct parallelization falls to the tool-developers.
Dynamic programming, in short DP, is a solution concept to a large number of problems appearing in bio-informatics, such as comparing DNA-sequences [35]. First described by R. Bellman, it is a method for solving complex problems whose solution is composed of (overlapping) solutions for smaller sub-problems [8]. This behaviour is described by a recursive equation: the so-called DP-equation (also called Bellman-equation). In order to avoid needless recomputation, the sub-results are saved in a matrix, called the DP-matrix. While dynamic programming is an efficient technique, the matrix often becomes huge when the concerned problem is large. A solution for this problem is to distribute the computation of the matrix. However, while dynamic programming shows potential for gains in speed by exploiting parallelism, it is also notoriously hard to parallelize [23].

1.1. Goal of this thesis

In this thesis we focus on parallelizing dynamic programming programs. There exist many parallel solutions for specific recurrence-relations and application problems, yet only few publications deal with research towards parallel dynamic programming in general [37] [1]. The main goal of this thesis is to *automatically* parallelize DP-programs written in R [3] using the approach given in [31]. More specifically, our primary goal is to develop a tool which automatically recognizes structures of dynamic programming in R code and transforms them into parallel versions using data-parallel list skeletons (cf. section 3.3). Our tool is embedded into the ALCHEMY framework [34], which allows researchers to experiment with different parallelization techniques for R. A secondary goal of this thesis is to evaluate feasibility and performance of the parallelization

[1] For a more detailed discussion of related work see section 3.4.

approach from [31] experimentally, using a set of dynamic programming R programs.

1.2. Structure of this thesis

Chapter 2 provides an overview of our approach and states which problems must be overcome. In chapter 3 we introduce the theoretical background and foundations of this thesis and give insight into related work. Chapter 4 establishes the theoretic base and describes how the approach outlined in chapter 2 is realized in detail. In the following chapter, chapter 5, we show how these requirements are realized and how MATSU is implemented. Afterwards, the results of an experimental evaluation are presented in chapter 6. The thesis concludes in chapter 7 with an overview of the current state of our tool and an outlook on further research opportunities.

2. Approach

This section presents a high-level overview of our parallelization tool, *MATSU*. Given a DP-program written in the R programming language, MATSU's goal is to convert this program to a semantically equivalent, but parallel version. We base our approach on the one developed by Kaheki, Matsuzaki et. al. [31].
The automation of the parallelization process poses multiple challenges. On the one hand, we want to realize this parallelization with an absolute minimum of user-input. That is, besides the program itself, the user does not have to submit additional meta-knowledge to aid MATSU. As a direct consequence of this, we are responsible for gathering necessary knowledge from the program representation itself. In particular, we are responsible to determine if the input is a valid DP-program and whether it can be handled by the approach.
On the other hand, as noted by the authors of [31], their approach still requires human insight to be useful. So an additional challenge is to make good parallelization choices in a generic fashion, thereby extending on the original approach. We will proceed to give a high-level overview of how the approach works and how it is realized.

2.1. Overview

The parallelization process can be divided in the following stages. A more detailed description can be found in chapter 4. Initially, MATSU will be fed a program representation of a R-program: the so called *Analysis Intermediate Representation* (*AIR*) of ALCHEMY.

Pattern matching for dynamic programming As input, we get a plain AIR of the input R-program. That is, we do not get any information in addition to the program. This implies that we have to find the occurrence of dynamic programming in the input program ourselves. To achieve this goal, we will first specify what the necessary ingredients for a DP-program are. In particular, we want to derive a informal notion of a *normal form* of DP-programs. Then we will specify this notion in a formal setting, called *attribute grammar*, and infer a algorithm which is capable of detecting dynamic programming.

Dependency Analysis Any program, which passed the previous step will now be analysed. An important requirement for deriving a correct program is to obey the data-dependencies. That is, we want to fix an *order of computation*, such that no element has any dependencies onto its successors. However, besides this order of computation, we also want to know which elements can be computed at the same time – i.e., can be efficiently computed using parallel programming patterns. To combine this idea with the notion of order, we specify and formalize the notion of *generations* introduced in [31]. Using this definition, we reformulate the original dependencies

and classify them into two types: First, dependencies into earlier generations, i.e. computations which refer to elements that were already computed in an earlier step. Those dependencies are referred to as *intergenerational*. Second, *intragenerational* dependencies, which are dependencies of elements within the same generation.
As the actual output of this stage we will then create an *oracle* over dependencies, called a *generation description*. It is able to answer queries about the type of dependency.

Function Splitting The function splitting routine fulfils two important steps. First, it takes the DP-Formula and derives a new intermediate representation: the split function tree. It divides the DP-formula in two parts: *summary functions*, which calculate a value out of multiple DP-matrix cells, and *inner functions*, which modify input from only one DP-matrix cell. This new representation lifts the semantics of the AIR to one more fitting for our purposes. Second, we will use the previously derived generation description to assign dependencies to the inner functions by analysing their induced dependencies.

Program Transformation In the final part, we will use the split function tree derived in the previous stage to derive a parallel computation according to the approach of [31]. In particular, we will use so called *data-parallel skeletons* (cf. section 3.3) to compute the members of each generation in parallel. In particular, this approach is focused on computing the dependencies to already computed values first. That is, we combine the values of previous generations in parallel using the *zipw*-skeleton. *Zipw* takes a function and two lists and outputs a list of pairwise combination of the list-elements using the given function. After this, the remaining dependencies will be resolved as efficiently as possible using the *scan*-skeleton. *Scan* iteratively combines all elements in a generation using a given binary-function or -operator.

3. Related Work

3.1. Lists and matrices

We adapt the List-construction from [27]. A *list* of length n over a set X is defined as a function $l : \{1,...,n\} \to X$. The set of all lists of length n over X is denoted by $L^n(X)$. The set of all lists over X is denoted by

$$L(X) = \bigcup_{i \in \mathbb{N}} L^i(X)$$

For a list $l \in L(X)$ with $l(1) = x_1, ..., l(n) = x_n$ we shortly write $[x_1, x_2, ..., x_n]$. By writing $l[i]$ we denote the i-th element of a list l. The list $l = [\, i \in X \mid C]$ with condition C is defined over the set $X' = \{i \in X | \, C\}$. We write $x \in l$ if there exists some i for which $l[i] = x$.
Furthermore, we will deal with matrices, or rather functions which are defined over sets of lists. That is, we have a function $e : \{1,...,n\} \to L(X)$. Then instead of writing $(e(i))[j] = e[i][j]$ to access the j-th element of list $e[i]$, we write $e[i, j]$. Finally, if we write $e[i,]$ we denote the list $e[i]$. Conversely, $e[, j]$ will return the list $[\, l[j] \mid l \in e\,] = [l[1], l[2], ...]$.

3.2. Dynamic Programming

Dynamic programming is a problem solving method mostly used for optimization problems. It was first described by R. Bellman [8]. The main idea of dynamic programming is to simplify a complicated problem by breaking it down into simpler subproblems in a recursive manner. Furthermore, to avoid recomputation of subproblems, the results of each subproblem is saved in a matrix. For optimization problems, these ingredients of dynamic programming are known as *optimal substructure* and *overlapping sub-problems* [16].

The relation of a problem to its subproblems is described by a recursive equation, the so called *DP-equation* (also known as DP-recurrence, DP-formula or Bellman equation). An example is shown in figure 3.1.

For a $N \times M$ matrix D, compute:

$$D[i,j] = \max \begin{cases} D[i, j-1] + 1 \\ D[i-1, j] + 2 \\ D[i-1, j-1] + 3 \end{cases} \quad \text{for} \quad \begin{matrix} 0 \leq i \leq N \\ 0 \leq j \leq M \end{matrix}$$

Figure 3.1.: A DP-formula

In the context of this thesis we are interested in the data-dependencies induced by the DP-equation. Formally, for a DP-equation ξ, we define the relation $\rightarrow_\xi$ to be the set of data-dependencies of ξ. I.e., we have $D[i,j] \rightarrow_\xi D[k,m]$ if and only if $D[k,m]$ is used to compute $D[i,j]$. Our example in figure 3.1 induces the following dependencies:

$$\rightarrow_\xi = \{(D[i,j], D[i-1,j-1]),\ (D[i,j], D[i,j-1]),\ (D[i,j], D[i-1,j]) \mid \forall i,j\}$$

3.3. Data-parallel Skeletons

Algorithmic skeletons are patterns of parallel programming. The term *skeleton* was coined by Murray Cole in 1991 [13]. Not unlike their more widely known counterparts, the *Software Design Patterns* (cf. [22]), they abstract from common patterns of computation and organization in programming. What separates skeletons from other models of parallel computation is that the manner of computation and synchronization is implicitly defined by the skeletons themselves. More complex skeletons can be created by combining basic ones. At present, there are a number of approaches and solutions to skeletal parallelism available. A recent summary and comparison of available frameworks can be found in [26].

The type of skeletons encountered here are called (data-parallel) *list skeletons* [14]. They are directly linked to the Bird-Meertens Formalism [9, 10, 41]. We can consider these skeletons to be higher-order functions, which take a function f as an argument. In the context of this thesis the following skeletons are of particular interest:

map($f(x)$, L) Given a function f and a list L, map applies f to every element contained in L. Formally:

$$\text{map}(f,L) = [\ f(x) \mid x \in L\ ]$$

zipw($f(x,y)$, L_1, L_2) Given a *binary* function f and two lists L_1 and L_2, zipw computes a component wise combination of both lists using f. Formally:

$$\text{zipw}(f,L_1,L_2) = [\ f(x,y) \mid x = L_1[i], y = L_2[i], 1 \leq i \leq n\ ]$$

scan($f(x,y)$, L) Given a *binary* function f and a list L, scan iteratively combines all elements in the list using f. If $f(a,b) = a+b$, this operation is also known as the *prefix sum* of L. Formally:

$$\text{scan}(f,L) = [\ f(b_{i-1}, b_i) \mid b_i = f(L[i], f(L[i-1], ..., f(L[2], L[1]))...), 2 \leq i \leq n\ ]$$

Finally, scan can be given an additional, optional parameter h. In this case the accumulative computation of b_i refers to any h-th preceding element.

3.4. Approaches to parallelization of dynamic programming

Developing automated versions of specific DP-programs for specific parallel architectures is the topic of many research advances in parallelization [11, 17, 21, 29, 40]. A survey of different methods can be found in [23]. Many of the more recent publications are concerned with dynamic programming problems that are frequently encountered in Bio-Informatics, such as String Matching [6, 7, 42]. However, there are only few publications about parallelization and formalization of dynamic programming in general [37]. One of the earliest advances into this territory was by [43], suggesting parallizations of certain classes of dynamic programming using parallel matrix chain product algorithms, which however led to inefficient solutions. An even earlier approach towards the formalization of dynamic programming was by Karp, who disassembled dynamic programming into discrete decision processes [32]. In 1989, Helman suggested an approach to the formalization of dynamic programming and branch-and-bound algorithms using dominance relations on the ordering of computations [28]. In [33], the authors introduce a categorization of dynamic programming and suggest parallel derivations. Gibbons and Rytter present yet another way of deriving parallel DP-programs based on a approach for expression evaluation in [24]. Finally, following the work of Ibaraki (cf. [30]), DPSkel provides a task-parallel skeleton based framework for the parallelization of dynamic programming [25, 37, 38].

3.5. The DAG matrix formalization

The main result of Kaheki, Matsuzaki et. al. in [31] was the definition of a uniform framework of dynamic programming, the *DAG matrix formalization*. The computation of DP-matrices is implicitly defined by the framework itself. The main task is therefore to re-express the DP-program in terms of the framework.

The DAG matrix formalization describes how DP-matrices can be computed using generations. In particular, it uses two 2-dimensional matrices, E and F. For both matrices, the index i describes the operation a thread i has to do during computation. The index j, however, describes the number of generations. Therefore the cells $E[i, j]$ and $F[i, j]$ will be computed by thread i in step j. Furthermore, the values both matrices are defined recursively: the computation of $E[i, j]$ only refers to $F[i, j]$ and any $E[i-h, j]$ with $h \in \mathbb{N}_+$. The computation of $F[i, j]$ refers only to any $E[g(i), j-k]$ with $k \leq j$. In terms of generations, this means that F will first compute all the intergenerational values, while E will use the values of F to add the intragenerational values to it.

More formally, framework computes the matrices as follows[1]:

$$E[i,j] = (E[i - l_g^*, j] \odot w[i,j]) \otimes F[i,j] \quad \text{where } l_g^* \in \mathbb{N}_+ \tag{b}$$

$$F[i,j] = \bigotimes_{k \neq g} f_k(E[i - l_k^*, j - g_k^*], i, j, k) \quad \text{where } g_k^* \in \mathbb{N}_+ \tag{a}$$

To apply this framework to a DP-program, one therefore has to determine the following parts:

- $\otimes$, which we call the *summary function*
- $\{f_k \mid k\}$, which we call the *inner functions*
- $\{l_k^* \mid k\}$, the *inner offsets*
- $\{g_k^* \mid k\}$, the *generational steps*
- and $x \odot w[i,j]$ is an inner function for the intragenerational dependencies

Finally, the framework is implemented with data-parallel list skeletons as follows:

Procedure 3.1 Computation of the DAG matrix formalization

1: $F[,j] = \text{zipw}(\otimes, c_{l_1^*, g_1^*}, ...(\text{zipw}(\otimes, c_{l_{k-1}^*, g_{k-1}^*}, c_{l_k^*, g_k^*})))$
2: $E[,j] = \text{scan}(\ominus, zipw(\lambda(a,b).(a,b), F[,j], w[,j])$
3: **where**
4: $(c_l, w_l) \ominus (c_r, w_r) = (c_l \odot w_l \otimes c_r, w_l \odot w_r)$
5: $c_{l_k^*, g_k^*} = \text{map}(\lambda(i,x).f_k(x,i,j,k), idx(\text{shift}(l_k^*, E[,j - g_k^*])))$
6: $idx(L) = zipw(\lambda(a,b).(a,b), [1,...,N], L)$

However, this computation turned out not to reliably produce correct results. We therefore adapted the manner of computation. For more information please refer to appendix B.

[1] For the sake of exposition, we describe a reduced version which omits some parameters. For the full version, please refer to [31].

3.6. Tree Grammars

Tree grammars work like regular grammars, except that they generate a tree instead of a string. A thorough introduction to this and related topics can be found in [15]. Tree grammars are well studied, in particular in relation to code generation and pattern matching (cf. [5, 20]) as well as their algebraic properties [39].
Formally, a tree grammar G is a 4-tuple (Σ, N, P, n_s), where:

1. Σ is a finite set of terminal symbols,
2. N is a finite set of non-terminal symbols with $N \cap \Sigma = \emptyset$,
3. P is a finite set of production rules. Production rules can have two forms:
   a) $n \to a(r)$, where n is a non-terminal in N, a is a symbol in Σ and r is either $w \in N \cup \Sigma$ or a *regular expression* over $N \cup \Sigma$ with $L(r) \subset (N \cup \Sigma)^*$.
   b) $n \to x$, where $n, x \in N$.
4. n_s is the starting non-terminal, or *axiom*, $n_s \in N$.

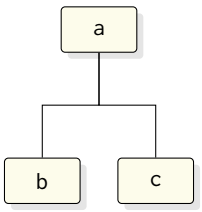
If r in the production $n \to a(r)$ is a symbol $w \in N \cup \Sigma$, the production replaces the non-terminal n with the tree $a(w)$. If, instead, r is a regular expression, the production replaces n with a tree $a(t_1 t_2 t_3 ... t_k)$, where $t_1 t_2 t_3 ... t_k \in L(r)$.

Consider the following tree grammar $G = (\Sigma, N, P, n_s)$:

- $\Sigma = \{a, b, c\}$,
- $N = \{X, Y, Z\}$,
- $n_s = X$,
- and P defined as:

```
X → a(Y,Z)
Y → b(ε)
Z → c(ε)
```

Then G would produce the tree:



Grammar rule:	X → a(Y,Z)
Synthesized attributes:	X.i := Y.i
	X.j := Z.i
Inherited attributes:	Z.k := X.k ∪ Y.i

Figure 3.2.: A structural definition the attributes for X → a(Y,Z) of (G,A).

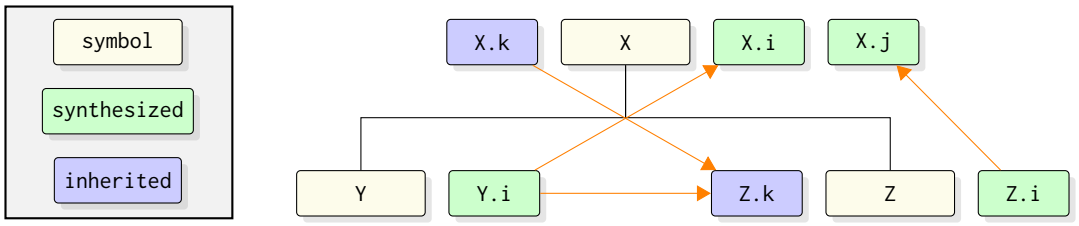


Figure 3.3.: Visualized version the attribute computation for X → a(Y,Z) of (G,A).

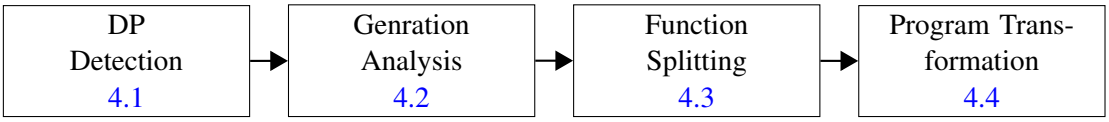
3.7. Attribute Grammars

Attribute grammars are a powerful description mechanism for semantic analysis of program trees [36]. An attribute grammar associates so-called *attributes* with productions of a formal grammar. That is, each node in the abstract syntax tree produced by a parser of this language will be associated with a value for such an attribute. More formally, for each symbol X of the grammar, $A(X)$ denotes the *set of attributes* of X. On the other hand, $X.a$ denotes the attribute $a \in A(X)$.

The computation of attributes is realized by traversal of the abstract syntax tree. We distinguish two types of attributes: *inherited* and *synthesized*. Inherited attributes are passed down from parent nodes. Synthesized attributes are computed from the evaluation rules of the current production, which may use values from its children and other attributes of the current production. Consider the attribute grammar (G,A), where G is taken from section 3.6 and with $A = \{X \to \{i,j,k\}, Y \to \{i\}, Z \to \{i,k\}\}$. A possible structural definition of the attribute-calculation for the rule $X \to a(Y,Z)$ can be found in figure 3.2. A visualized version is shown in figure 3.3.

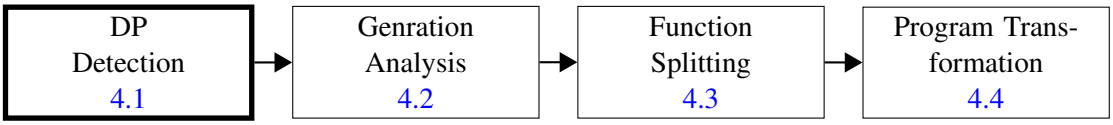
4. The Parallelization Process

In this chapter, we give an explanation on how the parallelization process works. It follows the structure of the translation, meaning that each section takes care of one step in the overall parallelization process outlined in chapter 2.
Each section will first give a very general explanation on what will be achieved in this particular stage. Then we will identify the problems in this respective stage, and briefly outline how each of them is encountered. Finally, we will give a detailed explanation of the algorithmic steps, using diagrams, pseudo code and illustrating examples as necessary.



4.1. Detection of Dynamic Programming

The detection of dynamic programming is the first step of the translation process. Given an input *AIR* abstract syntax tree (AST), the goal is to decide whether it constitutes a DP-program. Furthermore, if evidence of dynamic programming was found, it identifies the location of said evidence in the program for further processing. If not, an error will be reported and the parallization process will be aborted.



4.1.1. Problems and challenges

The challenged faced in this phase are two-fold: First, we need to find a pattern description of dynamic programming. This means that we need to give a plausible definition of how DP-programs are structured. Second, once we have identified a pattern, we need a mechanism, which is able to identify this pattern. Since pattern matching is undecidable in general (cf. [18]), we need to specify a mechanism that produces an acceptable output in finite time.

Concerning the identification of the pattern, we have the to reduce the set of possible DP-programs to those on which the approach of [31] is applicable. This approach has the following

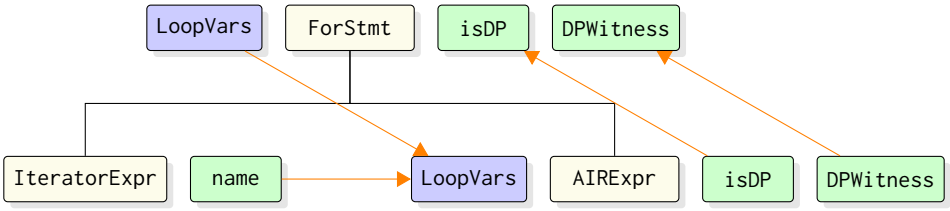


Figure 4.1.: A visualized version of the attribute grammar rule for `ForStmt`.

limiting factors:

- Any type of dynamic programming considered has a clearly defined *DP-equation*, which describes the calculation of values in a matrix.
- The DP-equation is a recursive modification of a matrix.
- The modification is done in a cell-per-cell fashion.

This means that the scope of our translation up to this stage is also limited by these factors. In fact, this means that we can deduce the pattern of dynamic programming problems we can cover as:

- A DP-program needs to have a recursive modification of a matrix.
- The modification is done in a cell-per-cell fashion.
- To ensure cell-per-cell modification of a matrix, a DP-program has to have nested loops whose iteration-variables are used to address specific cells.

In the remainder of this chapter, we will now specify an algorithm to check these conditions, and a formal setting, in which it will operate.

4.1.2. A formalism for DP-pattern-matching

As outlined in section 3.6, tree grammars are a powerful tool for pattern matching in trees. Since we do not have access to the source of the program and the AIR is already an intermediate representation build ontop of the actual R-AST, we will need to specify our approach as a grammar for the AIR-tree. A version of the actual AIR-grammar is given in [34].
On top of this grammar, we specify a so-called *attribute grammar*. An attribute grammar is usually used to ascertain semantic correctness of a input program [36]. For a formal definition refer to section 3.7.

Since the whole grammar is fairly large, we will restrict the explanation for the sake of readability. We will cover the most important attributes and the three mpst important specification rules: (for-)loops, assignments and matrix-accesses. The full grammar can be found in appendix A. In table 4.1, we classify and explain the use of the most important attributes.

Attribute	Type	Function
isDP	synthesized	If its value is true, this means that up to this point, the sub-tree of this node can be considered part of a DP-program.
DPWitness	synthesized	If a valid DP-assignment, i.e. an assignment fulfilling the properties of section 4.1.1 was found, this attribute will be a reference to its location in the tree. Otherwise, its null.
LoopVars	inherited	A set of variables that were defined in the context of a loop. Contains all such variables that are active at this node. That is, this attribute describes all variables that can be used in this nodes sub-tree.
MatrixToAccessID	synthesized	A map which assigns any matrix all variables which it was accessed by in the current sub-tree. E.g., if in the sub-tree of the current node was an access D[i,j], it would induce a mapping $D \rightarrow \{i, j\}$.

Table 4.1.: Some important attributes used in the DP-detection grammar.

4.1.3. Examination of the attribute rule for for-loops

We will examine how the attribute grammar is build using the example of the for-loop (`ForStmt`) in the AIR. The production-rule of `ForStmt` in the tree-grammar of the AIR is:

```
ForStmt → forstmt(IteratorExpr,AIRExpr)
```

The associated attribute grammar rule is:

Grammar rule:
`ForStmt → forstmt(IteratorExpr,AIRExpr)`

Synthesized attributes:
`ForStmt.isDP := AIRExpr.isDP`
`ForStmt.DPWitness := AIRExpr.DPWitness`

Inherited attributes:
`AIRExpr.LoopVars := ForStmt.LoopVars ∪ {IteratorExpr.name}`

We see that a `ForStmt`-node has two children: an `IteratorExpr` and a body, which can be any `AIRExpr`. The synthesized attributes of `ForStmt` are just the synthesized attributes of its children. In other words, if the body of a for-loop has a valid DP-assignment, then the for-loop is part of a DP-program. Hence `ForStmt.isDP` is assigned `AIRExpr.isDP`. For the same reason, the `DPWitness` of `ForStmt` is the same as its body.
However, the ForStmt itself is still important to identifying a DP-program. To re-iterate our results of the previous section, a DP-assignment must use variables used in a for-loop. That is, to modify a cell of the DP-matrix it uses the loop-variables. Conversely, this means that the body-part of the ForStmt has to have access to the variable that is used in its iterator. In terms of the grammar, this means that this attribute value has to be passed downwards. This explains the addition of iterator.name to the LoopVars set.

Besides the for-statement, there are still two other rules, which are of particular interest to us: subscript-expressions, which can constitute a DP-matrix access, and binary-operator expressions, which can constitute a DP-assignment. We will cover them in more detail in the following two sections.

4.1.4. Attribute-rules for Subscript-expressions

Grammar rule:
`SubscriptExpr` $\rightarrow$ `subscriptexpr(col=AIRExpr,AIRExpr+)`

Synthesized attributes:

let

K = `SubscriptExpr.loopVars` $\cap$ ($\bigcup_i$ `AIRExpr[i].usedVars`)

in

`SubscriptExpr.usedVars := {col.usedVars[1]}`
`SubscriptExpr.isDP :=` $K \neq \emptyset$

```
SubscriptExpr.matrixToAccessIDs :=
      if SubscriptExpr.isDP then
          { col.usedVars[1] → K }
      else
          ∅
```

end

Inherited attributes:
`col.LoopVars := SubscriptExpr.LoopVars`
$\forall i$: `AIRExpr[i].LoopVars := SubscriptExpr.LoopVars`

When using the attribute rule for a SubscriptExpr we are in the following situation:

- We are given a list of all loop-variables that are active at this point: `SubscriptExpr.LoopVars`
- We have knowledge on which variables were used in the expression which holds the col-

```
for i in 1:100
   for j in 1:100
       D[i,j] = ...  D[i,j-1]
                ...  M[i,j]
                ...  D[k,m]
```

Figure 4.2.: Example program

lection (`col.usedVars`) and which were used in the subscripts (`AIRExpr[i].usedVars`).

- We want to decide whether this matrix can be a DP-matrix.
- If this can be a DP-matrix, we want to add a mapping from the name of the collection to the indices used for accessing it to `matrixToAccessIDs`.

For a subscript expression to be a DP-access, it must be accessed by variables that were used in loops. Computing the union of the `usedVars`-list therefore gives us a list, which holds all the indices which were used to access said matrix. Finally, computing the intersection of said list with `SubscriptExpr.LoopVars` gives us the list K of all loop-variables that where used as indices in this subscript expression.
If K is non-empty, then this matrix constitutes a DP-matrix access. That is, we set `isDP=true`. Furthermore, we create a new member of the `matrixToAccessIDs`-mapping, which is

$$\texttt{col.usedVars[1]} \rightarrow K$$

where `col.usedVars[1]` gives us the name of the matrix. If K is empty, we can safely claim that it cannot be a DP-matrix.

Examples For the program in figure 4.2, the following subscript-expressions would be rejected (i.e. `isDP=false`):

- `D[k,m]`

The following subscript-expressions would be tentatively accepted (i.e. `isDP=true`) :

- `D[i,j]`
- `D[i,j-1]`
- `M[i,j]`

4.1.5. Attribute-rules for Assignments

Grammar rule:
`BinopExpr` → `binopexpr(assign, rhs=AIRExpr, lhs=AIRExpr)`

Synthesized attributes:

`BinopExpr.isDP` :=
`(lhs.matrixToAccessIDs(l.usedVars[1])`
`⊆ rhs.matrixToAccessIDs(l.usedVars[1]))`
`∧ lhs.isDP`

`BinopExpr.DPWitness` :=
if `BinopExpr.isDP` **then**
`{ (binopexpr,`
`l.usedVars[1],`
`l.matrixToAccessIDs(l.usedVars[1])`
`)`
`}`
else
∅

Inherited attributes:
`lhs.LoopVars := BinopExpr.LoopVars`
`rhs.LoopVars := BinopExpr.LoopVars`

When using the attribute rule for a BinopExpr where the operator is an assignment, we are in the following situation:

- We are given a list of all loop-variables that are active at this point: `SubscriptExpr.LoopVars`
- We know if the left-hand side (LHS) of the assignment is to a possible DP-matrix (`lhs.isDP`)
- We know if the right-hand side (RHS) of the assignment holds accesses to a possible DP-matrix (`rhs.isDP`)
- We have a map for both RHS and LHS, which maps any matrix-identifier to the indices it was accessed by (`lhs.matrixToAccessIDs` and `rhs.matrixToAccessIDs`)
- For this assignment to be a DP-assignment, the matrix assigned to on the LHS must be recursively accessed by the same indices on the RHS.
- If this is a DP-assignment, we want to create a DP-witness. A DP-witness is a triple consisting of a reference to the assignment, the name of the matrix and the indices used to access it.

So at this point, we already have full information on how the matrices are distributed in the assignment. The ultimate requirement, which the assignment has to fulfil in order to become a DP assignment, is that the assignment is recursive over the matrix, using the same variables. We can check for this by using the `matrixToAccessIDs` maps. By computing `lhs.usedVars[1]`

we get the name of the modified matrix. Computing `matrixToAccessIDs(lhs.usedVars[1])` conversely gives us all indices by which the matrix was modified by. So if

$$\texttt{lhs.matrixToAccessIDs(lhs.usedVars[1])} \subseteq \texttt{rhs.matrixToAccessIDs(lhs.usedVars[1])}$$

then we know that the RHS modifies the matrix on the LHS. The condition for this assignment to be a DP-assignment is therefore to the subset-condition shown above to hold, and furthermore, for both sides to be potential DP-parts. That is, for both sides `isDP` has to be true.
Due to the semantics of AIR, a BinopExpr can have multiple functions. The rule shown here is only part of a bigger rule. For example, an assignment can be used to define functions. In particular, the statement:

```
f <- funtion(x,y) { ... }
```

would be shown in the AIR as an assignment of a `FuncDef`-expression. Furthermore, a BinopExpr could also use non-assignment operators. For this reason, the real attribute rule is more expansive. For the sake of exposition we do not cover these cases in this section. However, a full version of this rule can be found in appendix A.

Examples Consider the programs listed in figure 4.3. The grammar rules for a `BinopExpr` with assignment operator would accept program (a) and reject program (b).
For program (a), we have that:

- `lhs.isDP=true, rhs.isDP=true`
- `lhs.usedVars[1] = D`
- `lhs.matrixToAccessIDs = { D` $\rightarrow$ `{i,j} }`
- `rhs.matrixToAccessIDs = { D` $\rightarrow$ `{i,j} }`

Therefore, we would evaluate:

- `lhs.matrixToAccessIDs = rhs.matrixToAccessIDs`
  $\Rightarrow$ `BinopExpr.isDP = true`
- `DPWitness = { (binopexpr,D,{i,j}) }`

For program (b), we have:

- `lhs.isDP=true, rhs.isDP=true`
- `lhs.usedVars[1] = D`
- `lhs.matrixToAccessIDs = { D` $\rightarrow$ `{i,j} }`
- `rhs.matrixToAccessIDs = { M` $\rightarrow$ `{i,j}, L` $\rightarrow$ `{i,j}, Q` $\rightarrow$ `{i,j} }`

And therefore:

- `lhs.matrixToAccessIDs` $\not\subset$ `rhs.matrixToAccessIDs`
  $\Rightarrow$ `BinopExpr.isDP = false`
- `DPWitness =` $\emptyset$

```
for i in 1:100
      for j in 1:100
           D[i,j] = max(D[i,j-1],D[i-1,j],D[i-1,j-1])
```

(a)

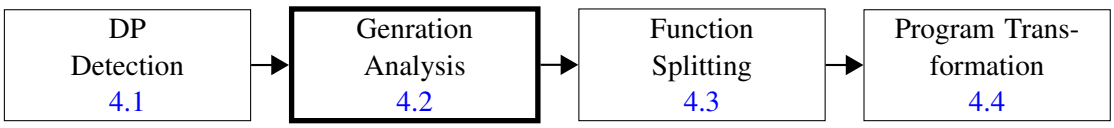
```
for i in 1:100
      for j in 1:100
           D[i,j] = max(M[i,j-1],L[i-1,j],Q[i-1,j-1])
```

(b)

Figure 4.3.: An accepted (a) and a rejected (b) program.

4.2. Generation Analysis

The generation analysis is the second step in the parallelization process and follows directly behind the detection of dynamic programming. The goal of this stage is to provide a so-called *generation description*, which is needed to express the dependencies between matrix accesses. This section first covers the theory behind generations and provides a formal definition. After that, we will apply the definition to the traversal we consider and consider dependencies between generations.



4.2.1. What are generations?

Before we start with the actual analysis, let us briefly introduce a formal notion of *generations*. The principle of generations relies on that of data-dependencies. Let D[i,j] $\rightarrow_\xi$ D[a,b] if the computation of D[i,j] under DP-formula ξ uses matrix-cell D[a,b]. We say *D[i,j] depends on D[a,b].*

Definition 4.2.1 *(Set-Partition)*
For a set X, $P \subset 2^X$ is called a partition of X *if and only if:*

1. $\bigcup P = X$
2. $\forall A \neq B \in P : A \cap B = \emptyset$
3. $\emptyset \notin P$

Then generations can be defined as:

Definition 4.2.2 *(Generations)*
Let $G = \{gen_k \mid k\}$ be a partition of $\{D[i,j] \mid i,j\}$. Let ξ be a DP-formula over D. Then G is the called a generation-set *if and only if:*

$$\forall i,j : i < j \Rightarrow \nexists D[a,b] \in gen_i, D[k,l] \in gen_j : D[a,b] \rightarrow_\xi D[k,l]$$

The elements $gen_k \in G$ are called generations*. We say x* can be computed at time *k if and only if $x \in gen_k$.*

Informally speaking, a generation is a set of matrix-cells, whose elements do not depend on *later* generations. The intent of generations is to define a partition of the whole matrix, such that the elements of this partition can be computed in order of their index.

4.2.2. Dependencies between generations

While the definition of generations effectively eliminates dependencies from one generation to later ones, we are still left with two other dependencies. We categorize them into two kinds:

1. **Intergenerational Dependency**, i.e. a dependency from the given matrix access into previously computed generations. Formally, a dependency $D[i,j] \rightarrow_\xi D[k,l]$ where $D[i,j] \in gen_a$ and $D[k,l] \in gen_b$ is called intragenerational if and only if $b < a$.

2. **Intragenerational Dependency**, i.e. a dependency into the same generation. Formally, a dependency $D[i,j] \rightarrow_\xi D[k,l]$ where $D[i,j] \in gen_a$ and $D[k,l] \in gen_b$ is called intragenerational if and only if $b = a$.

4.2.3. Generations for traversal-types

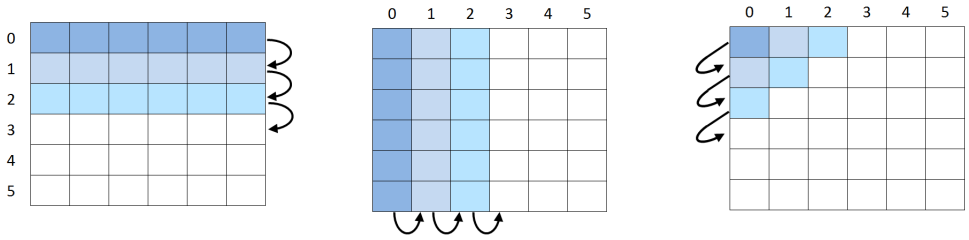
In this subsection we show how to derive generations from simple traversals. We consider three kinds of traversal options:

PerColumn Traversal on a *column-by-column* basis. The order of execution is then from left-to-right, computing one column at a time.

PerRow Traversal on a row-by-row basis. The order of execution is from top to bottom, executing one row at a time.

WF Traversal in a *wavefront* pattern. The Wavefront pattern describes a traversal method whereby the computation starts in the upper-left corner of the matrix and a sweep will progress across the plane to the bottom-right corner following a diagonal trajectory.

Using the definition of generations from the previous section, we can define fitting generations. In table 4.2 we show how one generation gen_k is derived for each traversal for a matrix D of size $N \times M$. The set of all generations is then the union of all generations gen_k.



(a) Per-Row traversal (b) Per-Column traversal (c) Wavefront traversal

Figure 4.4.: Traversal options illustrated.

Traversal	Definition of one generation k	Number of generations
PerColumn	$\text{gen}_k = \{D[i,j] \mid j = k\}$	M
PerRow	$\text{gen}_k = \{D[i,j] \mid i = k\}$	N
WF	$\text{gen}_k = \{D[i,j] \mid (i+j) = k\}$	$N+M-1$

Table 4.2.: Definition of generations for each traversal method for a $N \times M$ matrix

4.2.4. The generation description

The generation description's main purpose is to serve as an *oracle for dependencies*. That is, after its creation in this phase, there will be no point at which we will need to know which kind of traversal option or generations was chosen. Therefore the whole logic regarding dependencies must be encoded within the description itself. In particular, the generation description will be given a matrix-access and has to decide whether this type of access would constitute an intergenerational or intergenerational dependency to the currently processed value.

In this thesis we consider the current generation to be the generation which contains $D[i,j]$. This directly corresponds to the original definition given in [31]. This means that deciding if a dependency is intragenerational can be reduced to deciding if the matrix access is in the same generation as $D[i,j]$.

To add the dependency logic into the description we have to consider how it can be computed. Since we already have the formal definitions for each type of generation (cf. table 4.2), this is easy to formalize. Assume we are given a matrix access $D[f(i),g(j)]$[1]. Let $D[i,j] \in \text{gen}_{D[i,j]}$. Then we can simply check if $D[f(i),g(j)] \in \text{gen}_{D[i,j]}$ by:

Traversal	Condition for Intragenerational Dependency
PerColumn	$g(j) = j$
PerRow	$f(i) = i$
WF	$f(i)+g(j) = i+j$

Table 4.3.: Condition for intragenerational dependency given $D[f(i),g(j)]$.

[1] where $f,g : \mathbb{N} -> \mathbb{N}$, are functions of the form $f(x) = x-a$ and $g(x) = x-b$, for some $a,b \geq 0$.

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

(a) Original matrix

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

(b) Per-Column embedding

(1,1)	(2,1)	(3,1)	(4,1)
(1,2)	(2,2)	(3,2)	(4,2)
(1,3)	(2,3)	(3,3)	(4,3)
(1,4)	(2,4)	(3,4)	(4,4)

(c) Per-Row embedding

(1,1)	(1,2)	(1,3)	(1,4)			
	(2,1)	(2,2)	(2,3)	(2,4)		
		(3,1)	(3,2)	(3,3)	(3,4)	
			(4,1)	(4,2)	(4,3)	(4,4)

(d) Wavefront embedding

Figure 4.5.: Embedding of matrix D[i,j] for different traversals.

The final output of this stage is therefore an instantiation of an generation description which contains the logic defined in table 4.3. I.e., the query isIntragen($D[f(i),g(j)]$) can be computed by computing the respective condition.

4.2.5. Generational step and inner offset

Another type of query, which a generation description has to answer to are offset-queries. Since we want to rewrite the original DP-formula in terms of generations, we need to figure out how the dependencies themselves can be re-formulated. For this purpose, we will express the generation-partition as a matrix. The matrix E is one of the two matrices of the *DAG matrix formalization* of [31]. E is the base matrix of the reformed DP-formula where $E[,j]$ (the j-th column of j) contains the elements of gen_k. The rows of E, i.e. the $E[i,]$, denote the so-called *computational pedigrees*. Basically, one thread will always take care of one such pedigree, while the calculation progresses from column to column.

This gives us the challenge to embed the original matrix D and the DP-formula ξ over D into this formalization. Once again this differs by type of traversal. The definitions of matrix E can be found in table 4.4. A visualized example of this embedding process can be found in figure 4.5.

Traversal	Definition of $E[i,j]$ given D
PerColumn	$\forall i,j : E[i,j] = D[i,j]$
PerRow	$\forall i,j : E[i,j] = D[j,i]$
WF	$\forall i,j : E[i,j] = D[j-i,i]$

Table 4.4.: Definitions of E for each traversal-option

This describes how the matrix D is embedded in E. However, we still have to adapt the

DP-formula ξ. Since we change the semantics of the matrix (meaning an index of D has another meaning than an index of E), we have to find out in which way we can change the matrix-accesses such that they still point to the same value in the new matrix E. That is, for a given access-pattern $D[f(i),g(j)]$ we need to return values l^*, g^* such that $E[i-l^*, j-g^*] = D[f(i),g(j)]$. We call l^* the *inner offset* and g^* the *generational step*. Table 4.5 summarizes how they are computed for our choices of traversal.

Traversal	Computation of Inner Offset	Computation of Generational Step
PerColumn	$l^* = f(i) - i$	$g^* = g(j) - j$
PerRow	$l^* = g(j) - j$	$g^* = f(i) - i$
WF	$l^* = g(j) - j$	$g^* = f(i) + g(j) - (i+j)$

Table 4.5.: Computation of inner offset and generational step given $D[f(i),g(j)]$.

4.2.6. Summary of the generation analysis

In this phase, we have created an oracle for dependencies: the generation description. The generation description will later be used as a query-able black-box. This means that later phases will only rely on the output of those queries to make a decision, and will not try to examine the exact contents of the black-box. To summarize, the generation description has to answer the following queries:

Generation Description

isIntragenerational(MatrixAccess *m*) :
Decides if $D[i,j] \rightarrow m$ is a intragenerational dependency using a formula from table 4.3.

isIntergenerational(MatrixAccess *m*) :
Decides if $D[i,j] \rightarrow m$ is a intergenerational dependency using a formula from table 4.3.

computeInnerOffset(MatrixAccess *m*) :
Computes the inner offset l^* of m according to table 4.5.

computeGenerationalStep (MatrixAccess *m*) :
Computes the generational step g^* of m according to table 4.5.

4.2.7. Illustrating Example

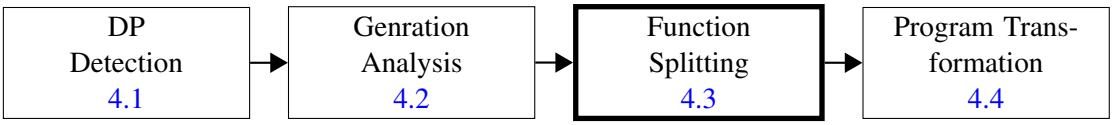
Assume we are given the traversal-option "PerRow". Then the produced generation description looks as follows:

PerRow Generation Description

```
isIntragenerational(MatrixAccess D[f(i),g(i)]):
    return f(i) == i;

isIntergenerational(MatrixAccess D[f(i),g(i)]) :
    return f(i) ≠ i;

computeInnerOffset(MatrixAccess D[f(i),g(i)]):
    return g(j) − j;

computeGenerationalStep(MatrixAccess D[f(i),g(i)]):
    return f(i) − i;
```

4.3. Function Splitting

Function Splitting is the third of four translation steps. In this phase, the goal is to identify and classify parts of the DP-formula. We want to split (or rather dissect) the formula into the building blocks for reordering them in the program translation step. In particular, we want to identify two kinds of functions: the so called *inner functions*, and the *summary functions*.



4.3.1. Splitting the AIR

The first goal of the function splitting process is to break down the DP-assignment into its important components. The intuition behind this is the following: A DP-formula usually takes multiple recursive values and then computes some optimal value[2]. It also sometimes applies functions to the recursive accesses.
This gives us the ability to identify two types of components:

Inner Functions An inner function describes how one individual matrix access is modified. Speaking in terms of the AIR-AST, an inner function is the maximal sub-tree, which has only one recursive access to the DP-tree as a leaf.
For example consider the DP-equation:

$$D[i,j] = max(D[i-1,j-1]+2, (D[i-1,j]-3)*2+4)$$

Then the inner functions of this equation are $f(x) = x+2$ and $g(x) = (x-3)*2+4$ respectively. Using these derived functions, we can rewrite the given formula as:

$$D[i,j] = max(f(D[i-1,j-1]), g(D[i-1,j]))$$

Summary Functions A summary function describes how values of multiple recursive matrix accesses are combined. In terms of the AIR-AST, a summary function is a node where the paths from at least two different recursive matrix-accesses to the assignment-root meet for the first time. Consider the DP equation that was modified in the previous paragraph. Then the summary function of:

$$D[i,j] = max(f(D[i-1,j-1]), g(D[i-1,j]))$$

would be $h(a,b) = max(a,b)$, since this is the first node in the AIR where the paths from D[i-1,j-1] and D[i-1,j] meet when going upwards. A summary node can potentially be any function or binary operator. However, the approach of [31] limits the options. In fact, it imposes two very strict requirements:

[2] usually for some particular notion of optimality, e.g. max.

- All summary functions of one DP-equation need to be the same.
- A summary function must be associative and commutative. We consider the following functions, which fulfil this requirement: addition($+$), multiplication($\times$), minimum ($\downarrow$) and maximum($\uparrow$)-functions. While logical-and ($\wedge$), logical-or ($\vee$), logical-xor ($\oplus$) and their negations also fulfil these requirements, they are currently not available in the AIR.

Finally, we will construct a new data structure: the *split function tree*. It will mimick the structure of the original AIR-AST, while consisting of bigger "building blocks": the inner functions and summary functions.

4.3.2. Algorithm for creation of the split-function tree

The algorithm to construct the new split function tree is based on the AIR-AST view of what these elements look like. As explained in the previous section, an inner function can be seen as the maximal subtree that has not more than one recursive matrix access node. The summary node is the exact opposite: a node whose subtree has at least two recursive matrix accesses.
Informally speaking, this means that we can state that the algorithm has to do the following subtasks:

- Construct the maximal upwards-path for any recursive assignment. That is, go upwards until you meet another path.
- Once you meet another path, this node must be a summary node.
- Convert all maximal sub-trees to inner functions. Then create a new summary node and assign the inner functions as its children.
- Repeat until root of assignment is reached.

Now that the individual tasks are clear, we will build the actual algorithm. Before we actually start with this, we can use two useful properties to make the construction principle clearer: First, we will need to visit all child-nodes of any node before we can make a decision. This is clear if we consider that there are functions (like max, min) which can have any number of parameters. Second, there are only two types of nodes which can constitute a summary node: binary-operators and function-calls.
Both factors make the description of an algorithm much easier. First, it fixes the tree-traversal order to be *post-order*, such that the children are visited first, and then the node computation itself is executed. Second, we can specify the algorithm in terms of *possible summary nodes* and *impossible summary nodes*.

Procedure 4.1 Visit function impossible summary nodes.

Input: AIRNode: node
Output: A new inner function node

```
 1: if node is a DP-assignment then
 2:         return produceInnerFuncNode(node);
 3: else
 4:         subnode ← visit(node.child);
 5:         if subnode = null then
 6:                 return null;
 7:         else if subnode.isSummaryNode() then
 8:                 return subnode;
 9:         else
10:                 return produceInnerFuncNode(node);
11:         end if
12: end if
```

Procedure 4.2 Visit function for possible summary nodes (BinopExpr, FuncCall).

Input: AIRNode: node
Output: A new split function tree node, which is either a summary or an inner function node

```
 1: sublist = new List();
 2: for all child ∈ node.children do
 3:         subnode ← visit(child);
 4:         if subnode ≠ null then
 5:                 sublist.add(subnode)
 6:         end if
 7: end for
 8: if sublist.size() ≥ 2 then
 9:         return produceSummaryNode(node,sublist);
10: else if sublist.size() = 1 then
11:         return produceInnerFuncNode(node);
12: end if
13: return null;
```

4.3.3. Illustrating example

We are given the AST of the DP-assignment shown in figure 4.6. Since the full AIR-AST is fairly large, we use a reduced version in this example, which is shown in the top part of figure 4.7.
The visitor will traverse the right-hand side of the assignment-node (`<-`) and first meet the upper max-node: `<MAX #1>` in the graph. Since this node is a `FuncCall`, it can possibly be a Summary node. Therefore both children are visited first. Since we do a post-order traversal, the left child is visited first. We visit `<MAX #2>`, and again visit its children first.
We again find a possible summary node: `<+ #1>`. And again, we visit the children first. This time we will visit the node `<D[i,j-1]>`. This is a subscript expression for which we already know that it constitutes a recursive access to the DP-matrix. Therefore we will return this node as a possible `InnerFuncNode`. We visit the next child of `<+ #1>`. This time we meet a a constant-expression, which cannot be an inner function on its own. Therefore we simply return. We are

```
D[i,j] = max(max(D[i,j-1]+2,D[i-1,j]+3),D[i-1,j-1]+1)
```

Figure 4.6.: An DP-assignment that will be split.

back at node <+ #1>. Since both children were visited, we will now evaluate the node itself. Since it had only one InnerFuncNode returned by its children, the node is part of the inner function as well. Therefore we throw the previous InnerFuncNode away and build a new one using the node <+ #1> as input, denoted as InnerFuncNode #1.
We visit the next child of <MAX # 1> : the node <+ #2>. The process is similar to the construction during our visit of <+ #1>. We visit both children and ultimately make a new InnerFuncNode: InnerFuncNode #2 consisting of node <+ #2>.
Now that both children of <MAX #2> are done, and both returned an InnerFuncNode, we know that this is a summary function. Therefore we build a new SummaryNode, SummaryNode #1. Its children are InnerFuncNode #1 and InnerFuncNode #2.
The second child of <MAX #1> is evaluated like the nodes <+ #1> and <+ #2> before, producing InnerFuncNode #3.
Finally, we have evaluated all the sub-trees of <MAX #1>. We obtain two results from those two sub-trees: SummaryNode #1 and InnerFuncNode #3. This means that <MAX #1> is a summary node as well, with those two returns as its children. This finishes the visit of the DP-assignment and returns the FunSplitTree as shown in the bottom part of figure 4.7.

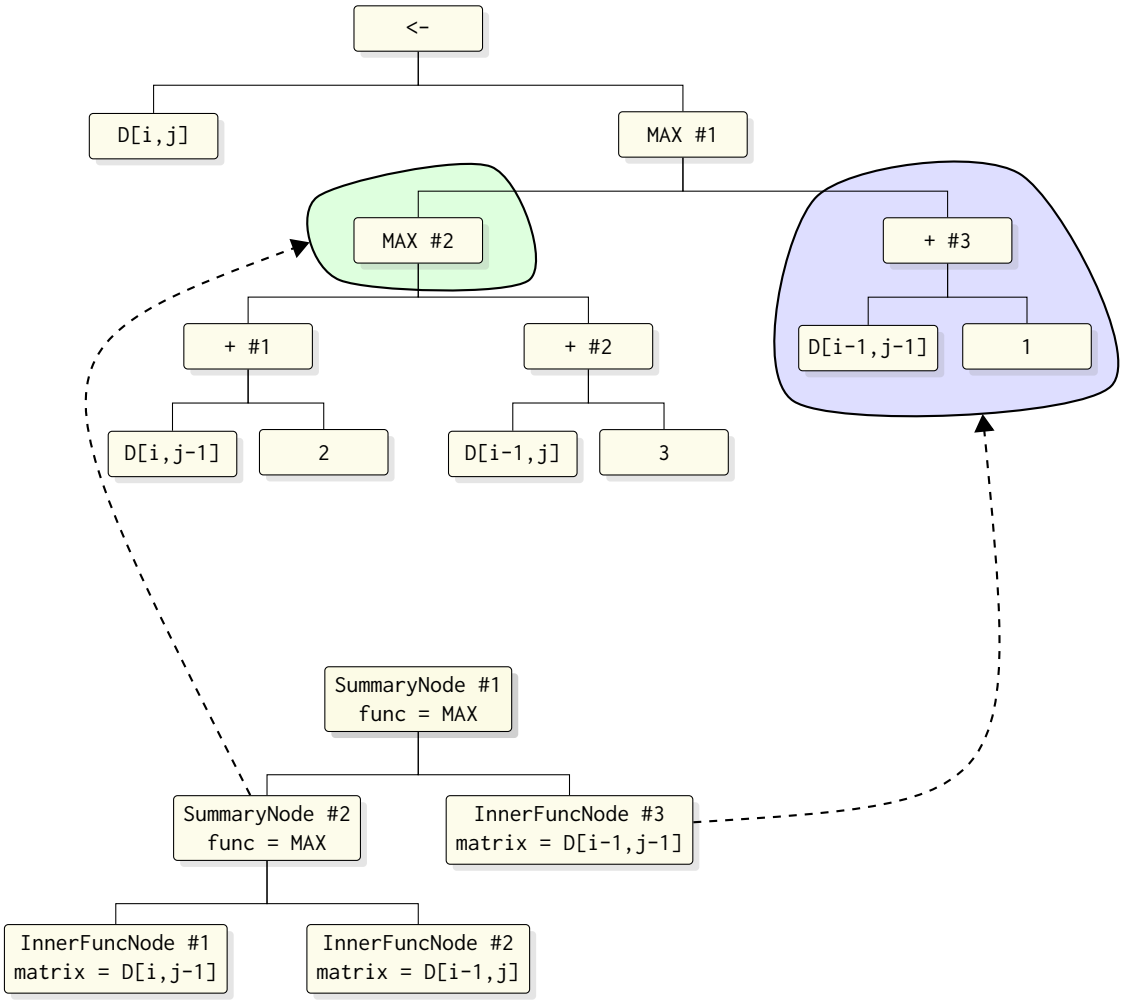
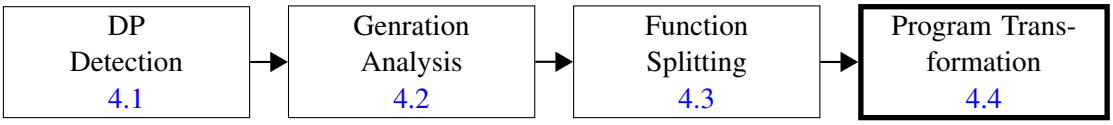


Figure 4.7.: Construction of the function split tree for the example DP-assignment in figure 4.6.

4.4. Program Transformation

The program transformation is the last stage of the MATSU translation process. In this phase, we are given the split function tree built in the previous phase. Our goal is to use its components as the building blocks for the final output. In particular, our goal is to use these components to fit them into *data-parallel list skeletons* using the approach of [31].



4.4.1. Computing DP with generations

For this approach to work in a generic fashion, we need to reformulate the original DP-formula to work over generations. In [31], the authors approach this by their *DAG matrix formalization*. The basic idea is to distribute the computation over two matrices, called E and F. F holds the values of the inter-generational part of the computation and takes values from previous generations of E, while E takes values from this generation of F to compute the intragenerational dependencies. Generations are encoded in this formalization as columns, i.e. $F[,k]$ holds all the intergenerational values of gen_k, while $E[,k]$ holds the values of gen_k of D.

More precisely, F is responsible for the computation of intergenerational dependencies. That means, $F[,k]$ has access to all $E[,j]$ where $j < k$. E is responsible for the computation of the intragenerational dependencies, which means that $E[,k]$ can access values in $F[,k]$. We will now show precisely how a generation gen_k can be computed in terms of E and F.

4.4.2. The adapted DAG matrix formalization

As noted in section 3.5, we had to adapt the DAG matrix formalization which was given by [31]. Procedure 4.3 summarizes the framework, i.e. how $F[,j]$ and $E[,j]$ are computed.

Procedure 4.3 Adapted computation of the DAG matrix formalization

1: $F[,j] = \text{zipw}(\otimes, c_{l_1^*,g_1^*}, ...(\text{zipw}(\otimes, c_{l_{k-1}^*,g_{k-1}^*}, c_{l_k^*,g_k^*})))$
2: $E[,j] = \text{map}(\lambda(a,b).b, \text{scan}(\lambda((i_1,x),(i_2,y)).(i_2, f_E(i_1,x) \oplus y), idx(F[,j])))$
3: **where**
4: $c_{l_k^*,g_k^*} = \text{map}(\lambda(i,x).f_k(x,i,j,k), idx(\text{shift}(l_k^*, E[,j-g_k^*])))$
5: $idx(L) = \text{zipw}(\lambda(a,b).(a,b), [1,...,N], L)$

Before we proceed with doing the actual program transformation, let us briefly summarize what we already computed and what we need to infer:

- Parts of the DP-program which constitute inner functions, i.e. the f_k and f_E
- $\otimes$, the summary function

- We inferred the generational steps g_k^* and the inner offsets l_k^*

So we already have all the parts that we need to express the given DP-program in the modified DAG matrix formalization.

4.4.3. Constructing the computation of F

F will represent the intergenerational dependencies. To do this in parallel using skeletons, we will have to fulfil two subtasks:

Applying the inner functions to one generation In particular, we will have to figure out how a computation of a inner function can be extended to one for the whole generation. We are given an inner function $f_{k,m}$ with offset l_k^* and generational step g_k^*. The derivation works as follows:

- Infer the *generational step* (g^*) for each inner function representation. Using a generation description *desc*, we can compute g^* for matrix access m as:
- Extract an *instrumented version* of the inner function $f_{k,m}$. An instrumented version f_k^{ins} is the function $f_{k,m}$ where the occurrence of matrix-access m is replaced by a parameter x. That is

$$f_k^{ins} = f_{k,m}[m/x]$$

- Then c_{l^*,g^*,f_m} is defined as:

$$c_{l^*,g^*,f_k} = \text{shift}(l^*, \text{zipw}(\lambda(i,x).f_k^{ins}, [1,..,N], E[,j-g^*]))$$

Note that our versions differs slightly from the one presented in [31]. The main reason for this change is that R does not natively support lists of tuples, as those would be interpreted as vectors of tuples – which does not work. Therefore, instead of using map and the function *idx*, we use zipw over the generation and an index-list, keeping the definition of the function used for the skeleton intact.

Combining the previous generations After completing the computation for the paragraph above, we obtain a list $\overline{c_{l,g,f}} = [\ c_{l_i,g_i,f_i} \mid i\ ]$. Furthermore, let $s(x,y)$ be the summary function which is used to combine the inner functions in the split-function tree[3]. Then we can derive the computation of $F[,j]$ as

$$F[,j] = zipw(\ s(x,y),\ \overline{c_{l,g,f}}[1],\ zipw(\ s(x,y),\ \overline{c_{l,g,f}}[2],\ ...))$$

That is, the values of $F[,j]$ can be computed by iteratively combining the contents of the inner-function skeleton-evaluations using zipw with the summary function as function-argument.

[3] As described in section 4.3, this function needs to be the same for all inner functions.

Construction Algorithm To construct the expression, we use the two algorithms described in the following. The output of these algorithms is an `SkeletonBuilder`, which can build an `AIRExpr` using a given `SkeletonFactory` (cf. section 5.5.4). The construction method for inner functions is implemented as procedure 4.4. The construction method for summary functions is implemented in procedure 4.5.

Procedure 4.4 Intergen Function Builder Visitor for `InnerFuncNode`

Input: An `InnerFuncNode`: node
Output: A builder which builds the intergen. skeleton for an inner function according to section 4.4.3

```
1: if ¬(node.isIntergen()) then return ∅;
2: end if
3: g* = node.getGenerationalStep();
4: l* = node.getInnerOffset();
5:
6: // Builds 1:dim(E)[1]
7: idx = buildIdxAsAIR();
8: // Builds E[, j − g*]
9: previousgen = buildPreviousGenAsAIR(g*);
10:
11: // create zipw-Skeleton builder with subparts
12: zipw = ZipWSkeletonBuilder(node,idx,previousgen);
13:
14: if l*>0 then
15:         return {ShiftSkeletonBuilder(l*,zipw)};
16: else
17:         return {zipw};
18: end if
```

Procedure 4.5 Intergen Function Builder Visitor for `SummaryNode`

Input: An `SummaryNode`: node
Output: A builder which builds the skeleton of a summary node according to section 4.4.3

```
1: list = ∅;
2: for all child ∈ node.children do
3:         childskel = visit(child);
4:         list = list ∪ childskel;
5: end for
6: if list.size() == 1 then
7:         return list;
8: else if list.size() ≥ 2 then
9:         zipw = list.removeFirst();
10:        // Build nested zipw-skeletons
11:        for all skel ∈ list do
12:                zipw = ZipwSkeletonBuilder(node,skel,zipw);
13:        end for
14:        return {zipw};
15: end if
16: throw ERROR();
```

4.4.4. Constructing the computation of E

Before we start with the explanation of the actual algorithm, we would like to stress that we diverge from the original formulation by [31] at this point. In the computation of $E[,j]$, we will now combine the values stored in $F[,j]$ according to the intra-generational dependencies. After this computation is finished, $E[,j]$ should hold the values of gen_j. We are given an intragenerational inner function $f_{E,m}$ with matrix access m and the inner offset l^*. The derivation process works as follows:

- Extract an *instrumented version* of the inner function $f_{E,m}$. An instrumented version f_E^{ins} is the function f_m where the occurrence of matrix-access m is replaced by a parameter x. That is:

$$f_E^{ins} = f_m[m/x]$$

- Combine f_E^{ins} with the summary function s as:

$$s'(i,x,y) = s(f_E^{ins}(i,x),y)$$

- The scan function is then:

$$g((i_1,x),(i_2,y)) = (i_2,\ s'(i_1,x,y))$$

- The function $idx(L)$ for list L, where the length of L is N:

$$idx(L) = \text{zipw}(\lambda(a,b).(a,b),[1,...,N],L)$$

- Then, according to the DAG matrix formalization, the computation of $E[,j]$ can be realized as:

$$E[,j] = \text{map}(\lambda(a,b).b,\text{scan}(g,idx(F[,j],l^*))$$

This construction process is implemented in processes 4.6 and 4.7 respectively.

Procedure 4.6 Intragen Function Builder Visitor for `InnerFuncNode`

Input: An `InnerFuncNode`: node
Output: If the node is inragen., returns set with node. Else returns the empy set. 4.4.4

```
1: if ¬(node.isIntragen()) then
2:         return ∅;
3: else
4:         return {node};
5: end if
```

Procedure 4.7 Intragen Function Builder Visitor for `SummaryNode`

Input: An `SummaryNode`: node
Output: A builder which builds the scan-skeleton of a summary node according to section 4.4.4

```
1:  list = ∅;
2:  for all child ∈ node.children do
3:          childskel = visit(child);
4:          if child.NodeType == SummaryNode and childskel.size()≥ 1 then
5:                  // then scan-skeleton is already built
6:                  return childskel;
7:          end if
8:          list = list ∪ childskel;
9:  end for
10: if list.size() ≥ 2 then
11:         throw ERROR("Too many intragenerational dependencies.");
12: else if list.size() = 1 then
13:         ifnode = list.removeFirst(); // Obtain the inner function f_E
14:         l* = ifnode.getInnerOffset();
15:         thisgen = buildAsAIR(F[,j]);
16:         indices = buildAsAIR(1 : dim(E)[0]); // [1,..,N]
17:         mkpair = buildAsAIR(function(a,b)c(a,b));
18:         zipw = ZipWSkeletonBuilder(mkpair,indices,thisgen);
19:
20:         // Instruments ifnode, combines it with node and constructs the scan skeleton
21:         scan = ScanSkeletonBuilder(node,ifnode,zipw,l*);
22:         return {scan};
23: end if
24: throw ERROR();
```

4.4.5. The final program

Now that we have constructed how the computations of E and its auxiliary matrix F are realized, let us construct the final program:

Procedure 4.8 Computation of a DP-formula using skeletons

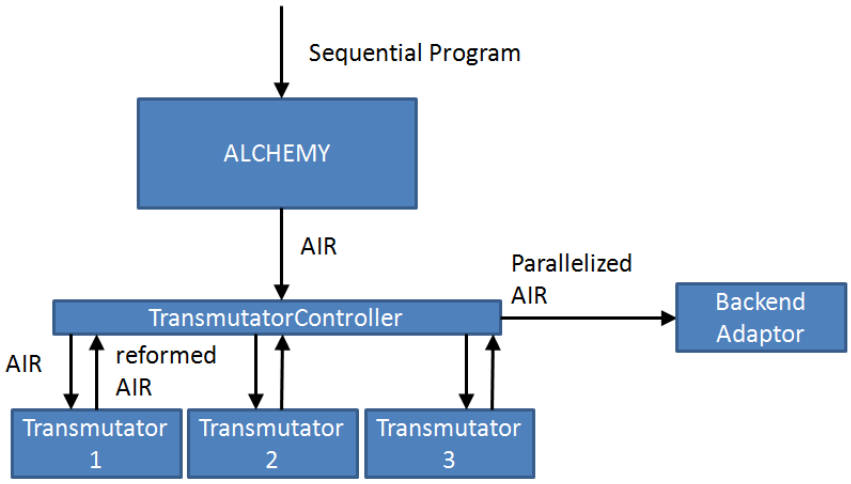
```
1:  for all j ∈ [1,..,dim(E)[2]] do
2:          F[,j] = zipw(s(x,y), c_{l,g,f}[1],
3:                       zipw(s(x,y), c_{l,g,f}[2],
4:                        ...
5:                       ) );
6:          E[,j] = map(function(a) a[2] ,
7:                       scan(function(d,e) c(e[1],s((f_E^ins)(d[1],d[2]),e[2])    ,
8:                          zipw(function(a,b) c(a,b)    ,
9:                                  1 : dim(E)[1],F[,j]
10:                                 ),
11:                         l*)
12:                      );
13: end for
```

5. Software Design and Implementation

This section gives an overview over the software design of MATSU. Section 5.2 depicts the major building blocks of this software. Section 5.3 describes how the parallelization pipeline is realized. The following sections are concerned with the design of the individual pipeline units. The description will always start with the structural and logical layout, and will go into as much detail as is needed to explain the functionality individual classes should provide.

5.1. Interfacing with Alchemy

MATSU is implemented as a `Transmutator` within the ALCHEMY framework. That is, MATSU will be scheduled by ALCHEMY for a translation process, where it will receive an AIR-tree as input.



In order to communicate with ALCHEMY, MATSU therefore implements ALCHEMY's Transmutator interface. The controller will then call the `transmutate`-method, which starts the translation process of MATSU.

5.2. Overall System Architecture

MATSU is split into multiple sub-packages. We identify six different packages, whereby the packages encapsulating the individual translation stages are responsible for the main functional-

ity of MATSU. The access point from ALCHEMY and the start-up of the translation process can be found in `Main`. The translation itself is split into five phases, where each phase only depends on the output of the previous one (cf. chapter 2). Therefore, the system is divided into packages, each of which represents one phase in the translation process. To unify these packages, we have one additional package `Commons`, which encapsulates the shared resources.

5.3. The MATSU Parallelization Pipeline

The parallelization process of MATSU is realized with a pipeline, following the pipes-and-filters design pattern [12].

The pipeline is set up using three generic parts: A `Pipeline`, a `PipelineUnit` and a `PipelineContext`. A `PipelineUnit` encapsulates a stage in a number of processes, which are executed in some order. In our case, each `PipelineUnit` takes care of one step of the translation process outlined in section 2. Informally, the `Pipeline` itself provides answers to two questions: *what* is executed, and *how* it is executed. In particular, the interface `Pipeline` has a method to add `PipelineUnits`. Any class, which implements this interface then decides in what order and fashion these units are executed. In our case we use a sequential pipeline. This means that all `PipelineUnits` contained in the pipeline are executed sequentially, in the order, in which they were added. A `PipelineContext` allows different pipeline units to collaborate by sharing data. A `PipelineUnit` will receive a context along with the execute-method, and will then perform the associated operations. After the execution is finished, the pipeline unit will put its results into the context and terminate, allowing the next pipeline unit (as specified by the pipeline) to take over.

The use of a pipeline offers multiple advantages to a regular computation. First, the parallelization process naturally decomposes into multiple processing stages. Second, the pipeline pattern forces loose interlocking between subsystems. Forcing separation between units makes it easier to change, remove or replace individual ones. In particular, it is easy to insert additional stages that can instrument the program to enable optimization (cf. section 7). Another possible option is to give different kinds of inputs, which then only need to add one pre-processing stage to the pipeline to translate to the expected input. The same is true for different kinds of outputs. Finally, the way the pipeline is set up makes the processing of the algorithm oblivious to how the pipeline is actually set up, which makes it easier to integrate more advanced features such as parallelism.

Last but not least, a very important aspect of ALCHEMY is to enable a user to review a translation process. This means, in particular, that both error-handling and notifications during translation are important. Enabling these kinds of features in a pipeline in a generic fashion is often problematic [12]. Since we want to handle each `PipelineUnit` as a generic part of the pipeline, we need one individual point where error-messages emerge. The solution we use is to use the `PipelineContext` as an aggregator for those messages. After a `PipelineUnit` has finished, either by an exception or normally, the `Pipeline` may check for errors. The implementation of `SequentialPipeline` mimics a try-catch block (cf. figure 5.3).

5.4. Configuring MATSU

Before the pipeline is executed, the user configuration is parsed. For this configuration, MATSU currently provides two options: First, the user can choose one traversal method out of three possible methods: PerRow, PerColumn or Wavefront. Second, she can specify the output type of MATSU. That is, how the AIR should interpret the skeletons: either as `FuncCalls`, or as `SkeletonExprs` [1]. In general, the `matsu/matsu_config`-file looks like this:

```
traversal= (PerRow | PerColumn | Wavefront)
output= (FuncDef | SkeletonExpr)
```

where $x|y$ denotes an option of either x or y. If no configuration-file is provided, MATSU will default to the standard configuration:

```
traversal=PerColumn
output=FuncCall
```

A `MATSUConfig` class will be instantiated with these choices and added to the `DPContext`.

5.5. Design and Implementation of the Parallelization Process

While the previous sections were concerned with the general architecture and layout of MATSU, this section will take a closer look at the parallelization process itself. Each section will be concerned with one unit of the parallelization pipeline, namely the:

- `DPDetector`
- `GenerationAnalyzer`
- `FunctionSplitter`
- `ProgramTransformator`

Each section will start with giving a medium- to high-level explanation, going into as much detail as necessary to explain the topic. After that, we show how this approach is implemented using appropriate diagrams or pseudocode.

5.5.1. DP Detection

The `DPDetector` ist the first `PipelineUnit` that is used. Its intent is to decide whether the input program, which is a tree of `AIRNodes`, is indeed a DP-program. For this it uses the `AIRNodeVisitor` interface, which follows the visitor pattern. This pattern is often used to traverse diverse data structures. Here, our goal is to verify a tree grammar whose rules are defined by subclasses of the AIRNode. A detailed diagram of the structure of AIRNodes and its sub-classes can be found in [34].
The `DPGrammarChecker` will implement visit-methods according to the attribute grammar specified in section 4.1. A pseudo-code version of a generic visit method is given in procedure 5.1.

[1] Currently, not all `SkeletonExprs` are supported by ALCHEMY.

Procedure 5.1 DPGrammarChecker visit method to evaluate attributes

Input: An `AIRNode`: node

```
1: // Map of synthesized attributes of children
2: catt = ∅
3:
4: // Compute the synthesized attributes of children
5: for all child ∈ node.children do
6:         compute the inherited attributes inh for child;
7:         childvisitor = DPGrammarChecker(inh);
8:         child.accept(childvisitor);
9:         csynth = childvisitor.getSynthesizedAttributes();
10:        catt = catt ∪ { child → csynth };
11: end for
12:
13: // Compute the synthesized attributes of node
14: compute the synthesized attributes synth using csynth;
15: setSynthesizedAttributes(synth);
16: return ;
```

5.5.2. Generation Analysis

The `GenerationAnanylzer` is the second `PipelineUnit`. The intent of this stage is to build as specific `GenerationDescription` based on the choice of the user. Any type of generation-description is a part of the `DPContext`. The associated package consists of the following classes:

GenerationDescription An interface that provides methods to decide if a dependency is intra- or intergenerational. Furthermore, it can extract the offsets of those dependencies. Since the exact nature of those generations depends on the choice of traversal, we need classes corresponding to a type of traversal.

PerColumnGenerationDescription A class, which encodes the information of a per-column traversal.

PerRowGenerationDescription A class, which encodes the information of a per-row traversal.

WFGenerationDescription A class, which encodes the information of a (top-left to bottom-right) wavefront-traversal.

The `GenerationAnalyzer` itself implements the `PipelineUnit` interface. Its execute method will instantiate a `GeneratioDescription` depending on the traversal option specified in the `MATSUConfig` of the provided `DPContext`.

5.5.3. Function Splitting

The `FunctionSplitter` is the third `PipelineUnit` involved in the translation process. Its goal is to classify parts of the AIR input based on the output of the `GenerationAnalyzer`. First, the function splitter deploys a visitor for the `AIRNode`: the `FunSplitVisitor`. The `FunSplitVisitor` traverses the AIR and builds a new representation, called the split-function tree. This tree is structured as follows:

SplitFunctionTreeNode An interface for all nodes of the split-function tree. It gives interfaces for a abstract visitor, the `SplitFunctionTreeVisitor`.

SummaryNode One of the two node types of the split-function tree. It constitutes an inner node, i.e. it has child elements of type `SplitFunctionTreeNode`.

InnerFuncNode A leaf node of the split-function tree.

SplitFunctionTreeVisitor An interface for a visitor of this data-structure.

How this tree is derived from the AIR-AST is described in section 4.3.2. The benefit of building a new data structure like this is not immediately apparent. The first major benefit is one in terms of abstraction level: instead of only classifying parts of the AIR-AST, we can actually traverse this new structure on the level of our new classification. That is, since we do not care about how the inner functions are actually working, we can use `SummaryNodes` and `InnerFuncNodes` as the basic "building blocks" in the next phase. Even better, we can hide the intricacies of building an AST behind multiple, higher-level factory methods. This makes changes on the intended level easier to realize.

After the split-function tree has been built, it is immediately used again along with the previously produced `GenerationDescription` to assign dependencies to the `InnerFuncNodes`. That is, the nodes are assigned the type of dependency their encapsulated matrix-access evaluates when querying the `GenerationDescription`. For this, we once again use a visitor, only this time a visitor for the `FunSplitTreeNodes`: the `DependencySetter`. Its visit method for an `InnerFuncNode` is shown in procedure 5.2.

Procedure 5.2 Dependency Setter

Input: An `InnerFuncNode`: node, a `GenerationDescription`: desc

```
1: matrix = node.getMatrixReference();
2: node.isIntergen = desc.isIntergen(matrix);
3: node.generationalStep = desc.computeGenerationalStep(matrix);
4: node.innerOffset = desc.computeInnerOffset(matrix);
5: return ;
```

5.5.4. Program Transformation

The `ProgramTransformator` is the last `PipelineUnit` to be executed. Its goal is to build the final program out of the `SplitFunctionTreeNode`-tree produced by the previous stage. For this, it first uses two `SplitFunctionTreeVisitors`:

IntergenFunctionBuilder Builds the intergenerational part of the translation process, i.e. the computation for F. Implements procedures 4.4 and 4.5.

IntragenFunctionBuilder Builds the intragenerational part of the translation process, i.e. the computation for E. Implements procedures 4.6 and 4.7.

SkeletonBuilder A builder for skeletons. Given the neccesary elements, a builder will produce a concrete version. The `build`-method requires a `SkeletonFactory`.

ScanSkeletonBuilder, ZipWSkeletonBuilder, ShiftSkeletonBuilder Builder methods for the skeletons scan, shift and zipw. `ScanSkeletonBuilder` and `ZipwSkeletonBuilder` will execute the instrumentation process for the nodes that represent their function-argument. The instrumentation was outlined in sections 4.4.4 and 4.4.3 respectively.

BaseSkeletonBuilder A wrapper for non-skeleton sub-trees to use for other `SkeletonBuilders`.

FunctionInstrumentor A class, which encapsulates the instrumentation-process for `AIRExprs` that is necessary for the `ScanSkeletonBuilder` and `ZipWSkeletonBuilder`.

SkeletonFactory An abstract factory for skeletons.

FuncDefSkeletonFactory Used for the `SkeletonBuilder` `build`-methods. Will produce a `FuncCall` to represent the respective skeleton.

SkeletonExprSkeletonFactory Used for the `SkeletonBuilder` `build` methods. Will produce a `SkeletonExpr` to represent the respective skeleton. Since not all skeletons are supported by ALCHEMY at the moment, this factory is not functional.

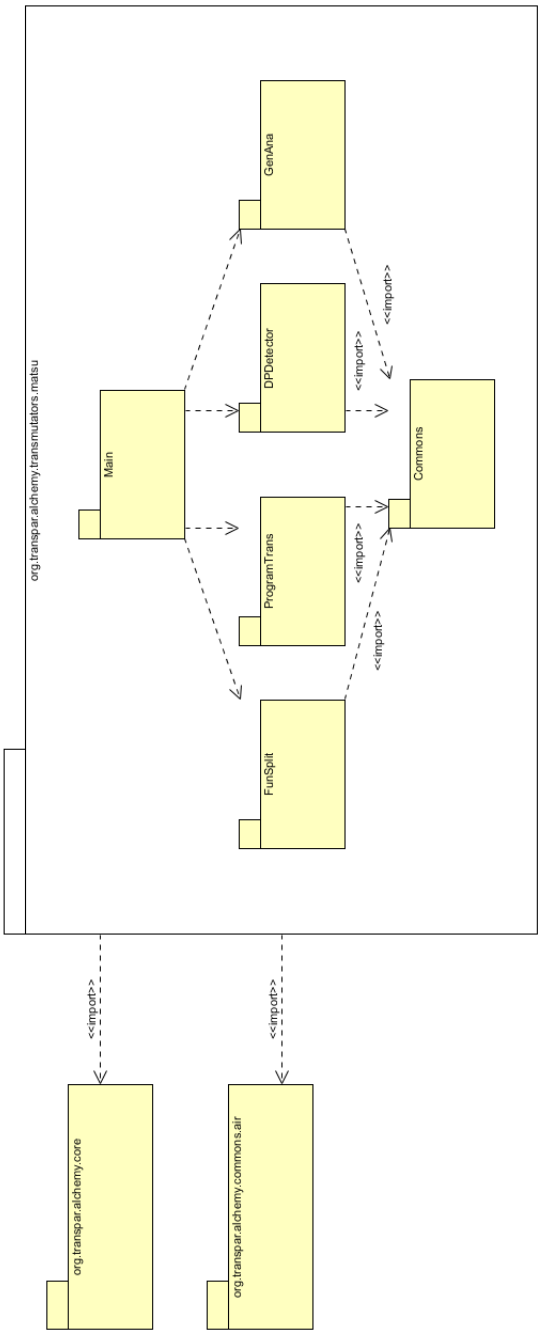


Figure 5.1.: Relationship between MATSU packages

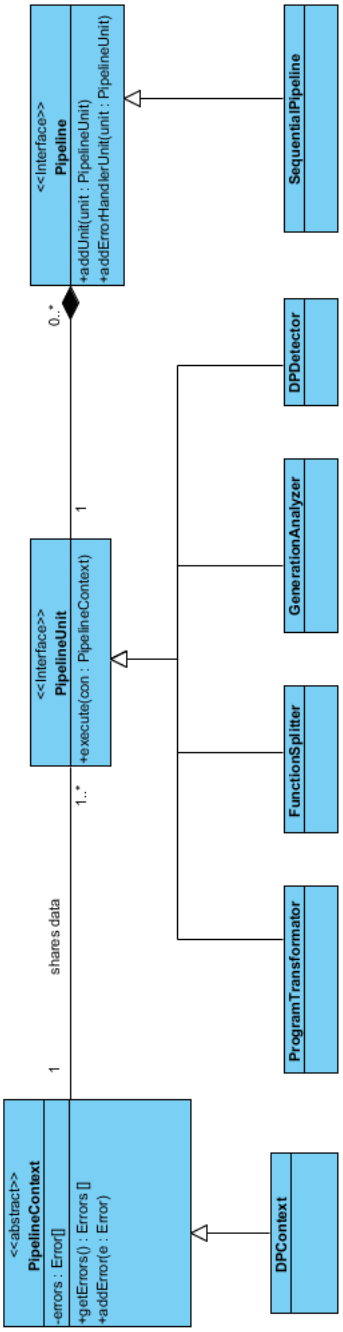


Figure 5.2.: Class diagram of the MATSU pipeline

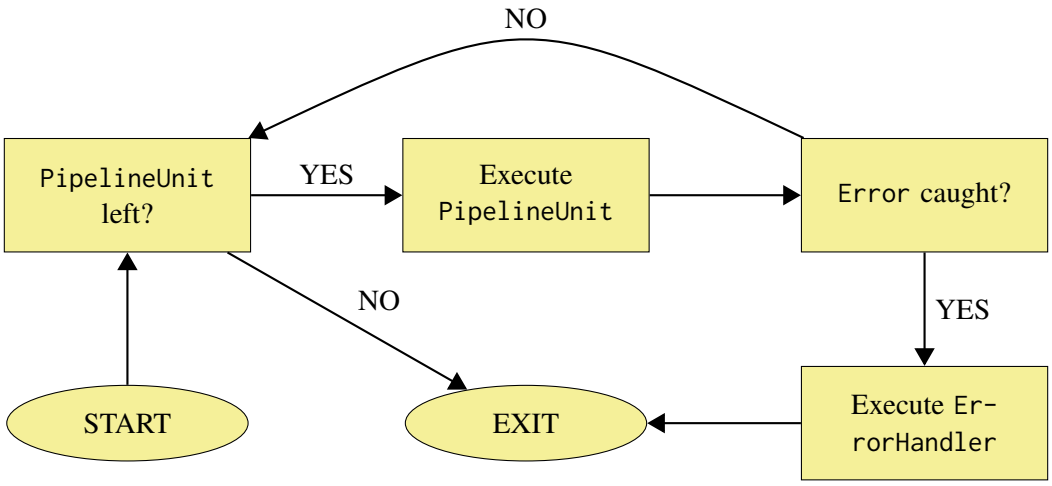


Figure 5.3.: Execution of the `SequentialPipeline`

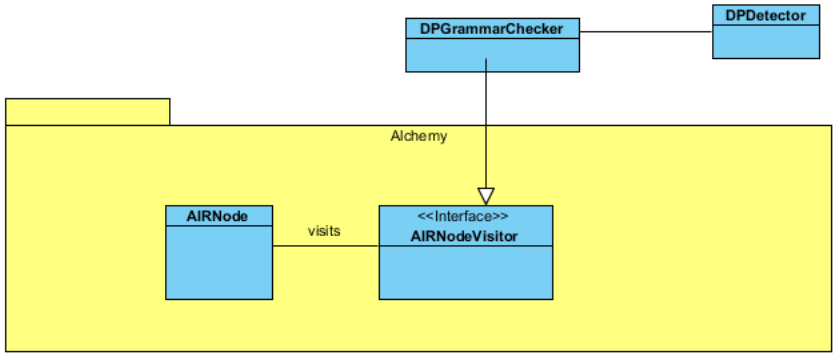


Figure 5.4.: Class Diagram of the DPDetector and its auxiliary classes

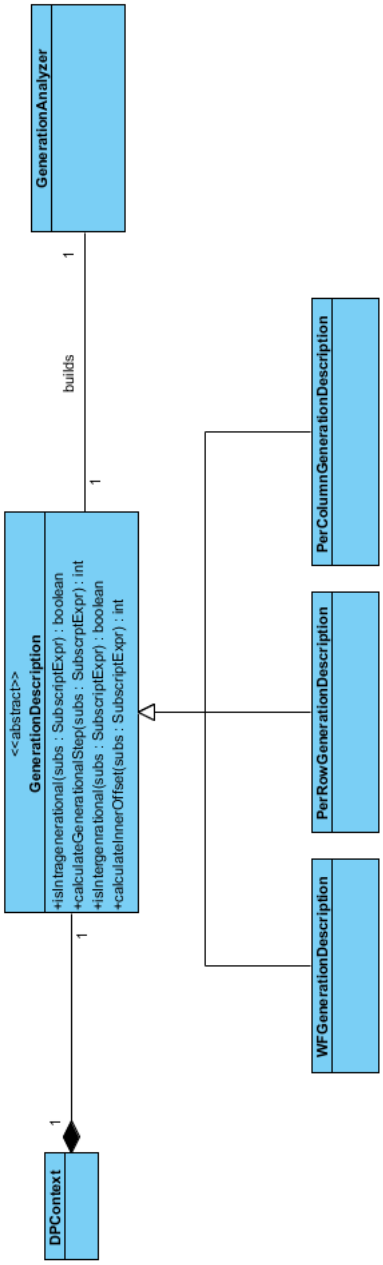


Figure 5.5.: Class Diagram of the Generation Analysis and its auxiliary classes

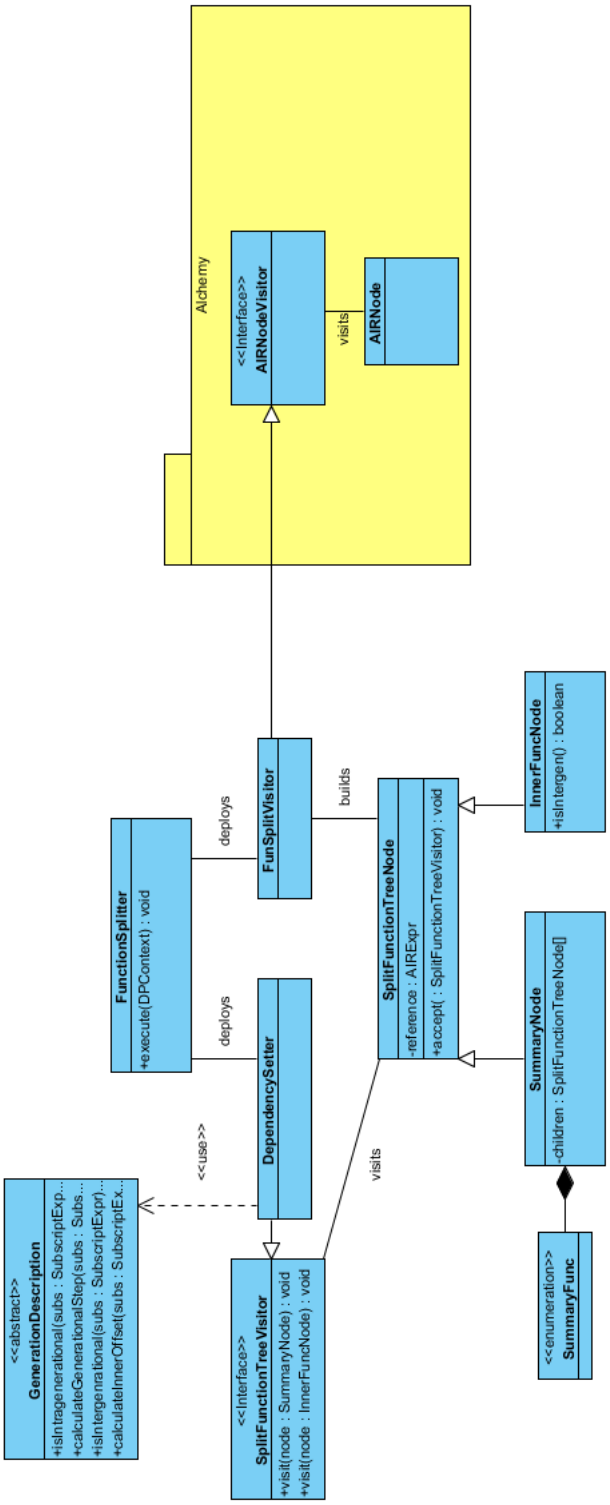


Figure 5.6.: Class Diagram of the Function Splitter and its auxiliary classes

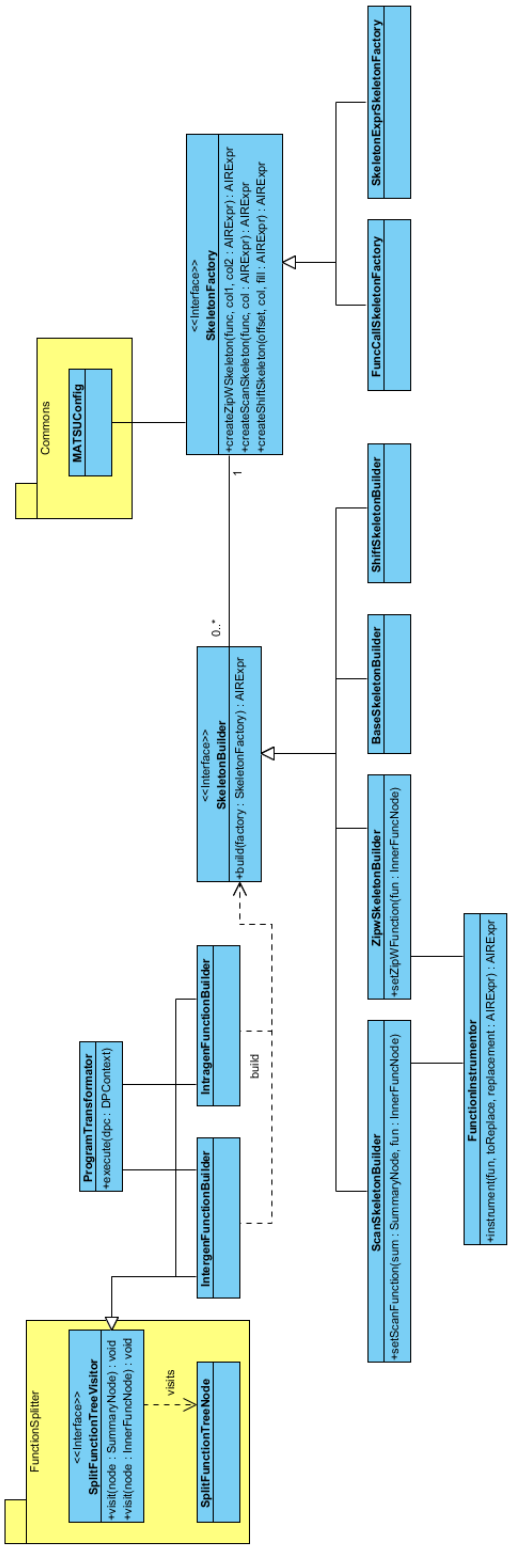


Figure 5.7.: Class Diagram of the ProgramTransformator and its auxiliary classes

6. Evaluation

The following experiments were used to find out if the parallelization approach for dynamic programming implemented in this thesis yields reasonable results. Furthermore, these results also give an insight into the performance of the resulting, parallelized programs.

6.1. On the scope of this evaluation

Originally, our intent was to perform a large-scale evaluation of multiple programs, which represent the classes of dynamic programming outlined in [33], chapter 12. However, the back-end adaptor for ALCHEMY was not finished in time. The back-end adaptor would have been responsible for translating the AIR, especially the skeletons, into an equivalent program for a parallel back-end[1].
Therefore, we could not run the parallelized programs in ALCHEMY, but had to implement them manually. Since this caused a much higher workload, we had to reduce the number of tested programs drastically.

6.2. Testing Environment

We conducted the experiments on a standard PC with the following characteristics:

- Processor: Intel® Core™ 2 Quad CPU Q9400 @ 2.66GHz
- Ram: 4GB (2 × 2GB DIMM DDR-3 1333MHz)
- Operating System: Ubuntu 12.04 (64Bit)

During the tests, no other programs were running with the exception of system processes. In order to test the program, we needed a parallel back-end which provides the necessary skeletons. To implement the translated programs, we used SkeTo [4], a C++ library providing implementations of skeletons using MPI [2]. In particular, we used the following software versions:

- C++ compiler: `g++ 4.5.2`
- MPI: `openmpi 1.6`
- SkeTo `1.10`

Listing 6.1: Input program A.

```
1 ... // initialize
2 for i in 2:N
3     for j in 2:M
4         D[i,j] <- max(max(D[i,j-1]+2, D[i-1,j]+3), D[i-1,j-1]+1)
```

6.3. Input programs and their implementation

Lisiting 6.1 shows input program A. Using MATSU, we will get a parallel version of this program represented by an AIR-tree. In particular, we derived a program for each type of traversal. Since the AIR-tree is fairly large and hard to read, we give an R program interpretation of the AIR in listing 6.2. Afterwards, we implemented the AIR of the programs in C++. The C++ version of the per-column traversal-option translation of A can be found in listing 6.3. For both programs we give the AIR-tree, represented with XML, in appendix C.

Listing 6.2: The result of the MATSU-parallelization of of A.

```
1 ... //initialize E and F
2 for j in 2:dim(E)[1]
3     F[,j]<- zipw( function(a,b) <- max(a,b),
4                   zipw(function(i,x) <- x+2, 1:dim(E)[1], E[,j-1]),
5                   shift(1,
6                         zipw(function(i,x) <- x+1, 1:dim(E)[1], E[,j-1]),
7                         0),
8                 )
9     E[,j]<- map( function(a,b) <- b,
10                        scan(function(d,e) <- (e[1], max(d[2]+3,e[2]))
11                                zipw(function(a,b) <- c(a,b), 1:dim(E)[1],
                                     F[,j])))
```

Let us briefly discuss some details of the C++ implementation. First, we use a new skeleton called `map_with_index` to implement several instances of `zipw`-skeleton computations which used `1:dim(E)[1]` as one of their input lists. However, the behavior of both skeletons is identical; `map_with_index` just automatically infers the required index. Furthermore, we distributed the computation into multiple subresults for the sake of exposition. We also do not use another matrix F, as F is only used to store the intermediate results of one computation. Hence, we do not need to store all the subresults. Finally, let us note that the implementation uses more skeleton computations than needed. Ideally, we can remove the `map` and `map_with_index` skeletons of the computation of E and only keep the `postscan`. We would then use a scan-function which does not operate on pairs, but on integer values. We briefly touch on that subject in chapter 7 and appendix B.

[1] In this case, this back-end would have been the Intel® ArBB parallelization library [1].

Listing 6.3: C++ implementation of the per-column traversal parallelization of A.

```
#include <sketo/list_skeletons.h>
//other includes

typedef std::pair<int, int> ipair;

using namespace sketo::list_skeletons;

int sketo::main(int argc, char** argv)
{

  const int size_N = std::atoi(argv[1]);
  const int size_M = std::atoi(argv[2]);

  //generate matrix E...

  for (int j=2; j<size_M; j++)
  {

    //Compute F[j]

    auto m1 = map_with_index( [](int i, int x) {return x+1;},
                  E[j-1]);
    auto m2 = shiftr(0,
                     map_with_index(  [](int i, int x) {return x+2;},
                                      E[j-1]));

    auto fj = zipwith(
                [](int a, int b){return (std::max(a,b));},
                m1, m2);

    // Compute E[j]

    auto idx_fj = map_with_index([](int i, int x){return ipair(i,x);},fj);

    auto ej_pairs = postscan(
                         [](ipair d, ipair e)
                             { return ipair(e.first,
                                       std::max( d.second+3, e.second));
                             },
                               idx_fj
                             );

     E[j] = map([](ipair p){ return p.second;}, ej_pairs);

  }
  return 0;
}
```

6.4. Measuring time

For this evaluation, we are interested in measuring the performance of the resulting programs. This means, in particular, that we measured the wall-clock time each algorithm needs to evaluate the DP-matrix. For this purpose we use the `std::chrono::high_precision_clock` in the C++11 standard library. The exact manner of integration is shown in listing 6.4.

Listing 6.4: Integration of chrono into the DP-programs.

```
1  ... //initialize
2
3  auto t1 = std::chrono::high_resolution_clock::now();
4
5  ... //do DP-matrix calculation
6
7  auto t2 = std::chrono::high_resolution_clock::now();
8  auto delta = std::chrono::duration_cast<std::chrono::milliseconds>(t2-t1)
9
10 sketo::cout << "run finished. time = "
11             << delta.count()
12             << "ms"
13             << std::endl;
```

6.5. Comparing different traversal orders

This experiment verifies the MATSU parallelization for program A using all three traversal types. That is, the performance of the resulting parallelized programs will be measured and compared.

6.5.1. Setup

First, we derived another variation of program A. These variations featured more expensive inner functions. Instead of being constant time functions such as additions, the cost of these inner functions scale linearly with the size of the input-matrix. We refer to this modified programs as `complex`, while the original was is referred to as `simple`.
Each program is input in ALCHEMY using a transmutation sequence only consisting of MATSU. The output factory of MATSU was set to `FuncCall` for all runs. This means that the skeleton-functions are visualized in the AST using `FuncCalls`. The traversal order was set to `PerColumn` for the first translation, `PerRow` for the second, `WF` for the last. In the following, we will refer to the different translation results by their traversal-choice. That is, the translation for A using traversal order `PerColumn` will be referred to as `PerColumn`, and so on.
For each translation, we implemented a corresponding C++ program using SkeTo. The source files will be titled `<simple/complex>_dp_<col/rw/wf>.cpp` depending on the implemented

translation. That is, an implementation of a `simple` program translation using `PerColumn` traversal is called `simple_dp_col.cpp`.
The source files were compiled using the `sketocxx` script[2]. The resulting executable files are named according to their source files, i.e. the same name without the `.cpp` ending.
Overall, we obtained 6 program instances. Each program is invoked using the command

```
$ sketorun -np 4 <program> <N> <M>
```

where `N` and `M` determine the number of columns and rows of the matrix to be processed. The parameter `-np 4` specifies the number of threads to be used as 4. We invoked each program thrice, first with `N=M=1000`, then `N=M=10000` and finally `N=M=20000`. This corresponds to matrices consisting of 10^6, 10^8 and 4×10^8 cells each.

6.5.2. Results

The running times of the processes were collected as described in section 6.4. The numerical results are shown in table 6.1. Figures 6.1 and 6.2 visualize these results.

Program	1000×1000	10000×10000	20000×20000
`simple_dp_col`	6	380	1295
`simple_dp_rw`	6	385	1340
`simple_dp_wf`	12	313	855
`complex_dp_col`	220	131,397	1,040,329
`complex_dp_rw`	226	135,870	1,041,681
`complex_dp_wf`	211	121,587	946,594

Table 6.1.: Running times (in ms) of parallelized programs.

6.6. Comparing parallel and sequential programs

This experiments tries to evaluate the effectiveness of the implemented approach.The running times of the programs that resulted from the previous experiment will be compared to a sequential version of the same DP-program.

6.6.1. Setup

In addition to the parallel versions obtained, we developed a sequential version, which implements the DP-algorithm of the input program. Since we wanted to evaluate the effect of the parallelization, we will implement a new version in C++. We will not go into details, as the implementation is straight-forward. The sequential versions are implemented in the programs `simple_dp_seq` and `complex_dp_seq` respectively. The programs were compiled and run using SkeTo in order to minimize the differences between programs and to focus on the computation

[2]The exact input parameters are system and compiler dependent. Please refer to the SkeTo manual [4].

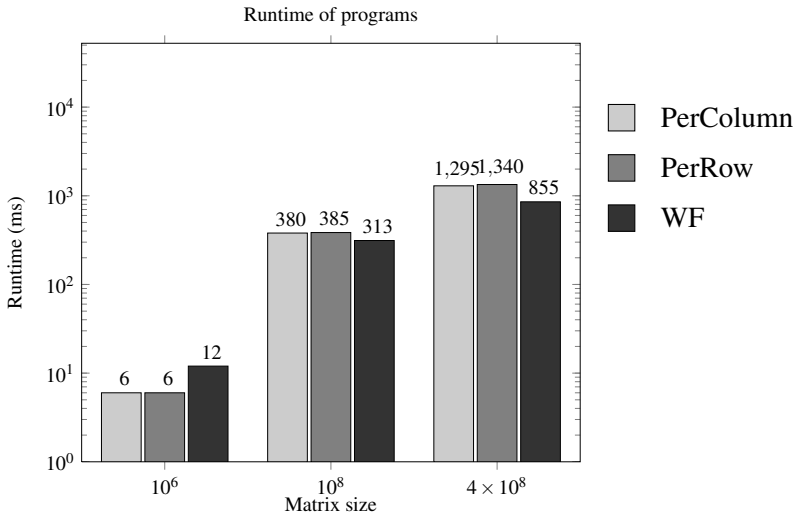


Figure 6.1.: Runtime of translations with simple inner functions.

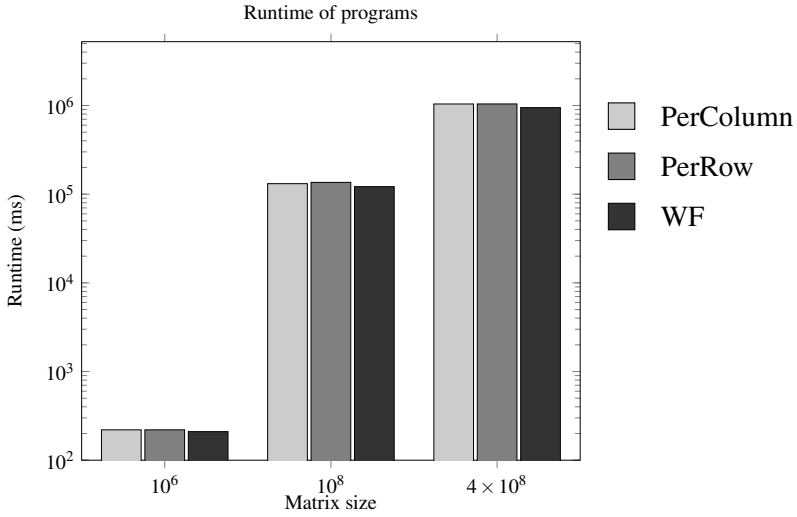


Figure 6.2.: Runtime of translations with complex inner functions.

Program	1000×1000	10000×10000	20000×20000
`simple_dp_seq`	4	361	1488
`complex_dp_seq`	390	189,501	1,424,976

Table 6.2.: Running times (in ms) of seqential programs.

Program	1000×1000	10000×10000	20000×20000
`simple_dp_col`	0.67	0.95	1.12
`simple_dp_rw`	0.67	0.94	1.08
`simple_dp_wf`	0.33	1.15	1.69
`complex_dp_col`	1.77	1.42	1.37
`complex_dp_rw`	1.72	1.42	1.36
`complex_dp_wf`	1.84	1.56	1.51

Table 6.3.: Speedup of parallelized programs

of the DP-recurrence. That is, we compiled the sequentials programs versions using `sketocxx` and ran them using

```
$ sketorun -np 1 <program> <N> <M>
```

Afterwards, we compared the running time of those programs with the ones from the previous experiment. To put the the parallelized execution into relation with a completely sequential one we use the notion of *speedup*:

$$Speedup = \frac{ExecutionTime(P_{sequential})}{ExecutionTime(P_{parallel})}$$

6.6.2. Results

The runtime results of the sequential programs are shown in table 6.2. The corresponding speedup values are summarized in table 6.3 and visualized in figures 6.3 and 6.4.

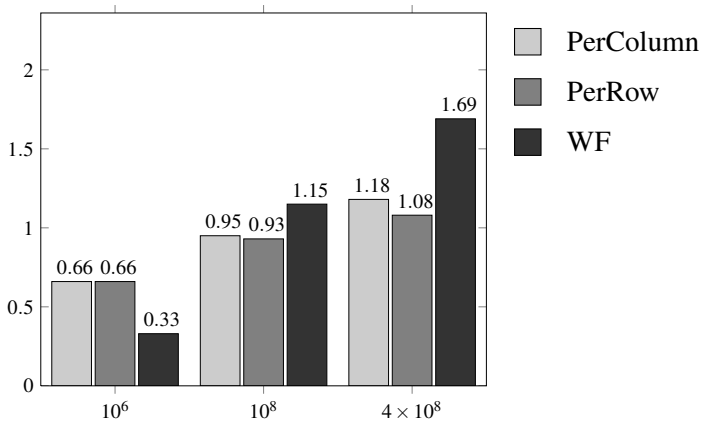


Figure 6.3.: Speedup for program A with simple inner functions.

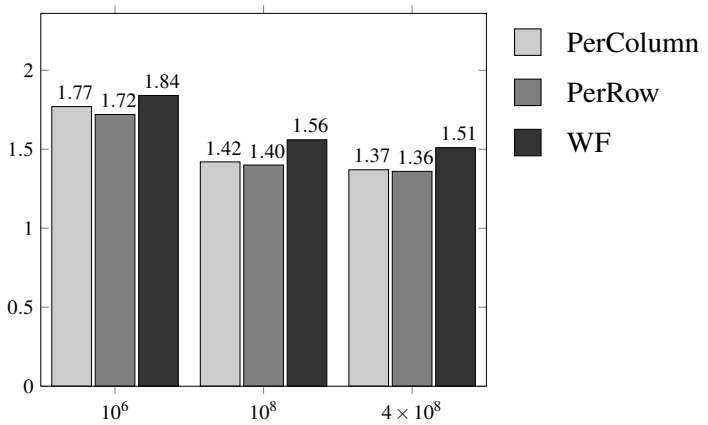


Figure 6.4.: Speedup for program A with complex inner functions.

7. Conclusions and Outlook

This thesis describes the concepts, design, theoretical foundations and realization of MATSU and its integration into ALCHEMY. MATSU is a tool capable of recognizing DP-structures and automatically parallelizing them using the approach of [31].
To facilitate the automatic recognition of dynamic programming, we derived a *normal form* of DP-programs. That is, we defined a lowest common denominator of DP-programs for which our approach is applicable. We showed how this *normal form* can be detected in an AST program-representation using a formal description mechanism called *attribute grammars*.
We continued to show how the *DAG matrix-formalization* of [31] can be automatically constructed. That is, we showed how the approach can be implemented such that a user does not have to give additional input besides the program. In order to express dependencies in a generic manner, we showed how generations can be implemented as an oracle, the *generation description*. We continued to derive a new intermediate representation for DP-equation expressions, the split function tree. This form allows our tool to work on a higher semantic level closer to the functionality of dynamic programming. Finally, we show how these individual parts can be used to derive skeleton-form programs in a generic fashion, i.e. without inspecting the source-program but only its derived components.
The experiments we performed showed two things: the approach works well if either the matrix is big, or if the inner-functions are expensive to compute. Using a four-core machine, we reached an average speedup of about 1.5. However, in other cases the overhead of more iterations (wavefront-traversal) or the set-up of the skeletons is overwhelming.

7.1. Future Work

Future work in this area could include the following topics:

- The evaluation of the approach is not finished by a large margin. In particular, we only considered three types of generations. Furthermore, as already mentioned in chapter 6, the back-end of ALCHEMY was not finished in time to be used for this thesis. Therefore we could only test a small fraction of the DP-problems we wanted to evaluate. Gaining more knowledge on the performance would be very beneficial to gain a better assessment of the approach's performance.
  We also only used one kind of distributed computing environment to evaluate the approach. Parallel computing often requires a good balance between communication and computation to perform well. Different environment place different weights on each of those two factors. For example, a node cluster consisting of multiple different machines suffer a larger draw-back from a communication intensive application than a shared-

memory environment. For an extensive evaluation this would be another factor to consider.

- When defining generations, we noted that we only cover tree possible types of traversal: by-row, by-column and wavefront traversal. However, the definition is generic to the actual form of those generations, and so is its implementation. Therefore MATSU can be easily extended by implementing other types of generations. Evaluating different kinds of generations by their performance impact could lead to interesting discoveries pertaining to data-dependencies and patterns in parallel programming. Of particular interest are generations, which split the matrix into *clusters* of computation and embedding higher-dimensional matrices from DP-problems (i.e. matrices with more than two dimensions). Furthermore, as seen in the evaluation, the kind of generation used has a non-trivial impact on the performance. To find the optimal generation choice for a DP-equation is therefore another interesting problem.

- The output of MATSU is a program which uses a very generic form of skeletons. Our implementation was designed with the intent to work on as few different skeletons as possible to maximize compatibility to existing parallel back-ends. The skeletons that were used might therefore not be optimal. For example, in the evaluation we used the `map_with_index` skeleton to replace our more complicated construction with `zipw`. Furthermore, we noted that the computation of E can be simplified if the inner function does not use the pedigree-index variable. Implementing an optimizer on skeleton-level is therefore another interesting direction, in which MATSU could be extended.

Bibliography

[1] Intel® array building blocks parallelization library (intel® arbb). http://software.intel.com/en-us/articles/intel-array-building-blocks/. Accessed: 07/07/2012.

[2] The message passing interface (mpi) standard. http://www.mcs.anl.gov/research/projects/mpi/. Accessed: 07/07/2012.

[3] The r project for statistical computing. http://www.r-project.org. Accessed: 07/07/2012.

[4] Sketo project. http://sketo.ipl-lab.org/. Accessed: 07/07/2012.

[5] A. V. Aho and M. Ganapathi. Efficient tree pattern matching (extended abstract): an aid to code generation. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '85, pages 334–340, New York, NY, USA, 1985. ACM.

[6] C. E. R. Alves, E. N. Cáceres, and F. Dehne. Parallel dynamic programming for solving the string editing problem on a cgm/bsp. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 275–281, New York, NY, USA, 2002. ACM.

[7] C. E. R. Alves, E. N. Cáceres, and S. W. Song. A bsp/cgm algorithm for the all-substrings longest common subsequence problem. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 57.1–, Washington, DC, USA, 2003. IEEE Computer Society.

[8] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[9] R. 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.

[10] R. S. Bird. Algebraic identities for program calculation. *Comput. J.*, 32:122–126, April 1989.

[11] P. G. Bradford, G. J. E. Rawlins, and G. E. Shannon. Efficient matrix chain ordering in polylog time. *SIAM J. Comput.*, 27(2):466–490, 1998.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.

[13] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[14] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. In G. R. Joubert, D. Trystram, F. J. Peters, and D. J. Evans, editors, *PARCO*, pages 489–492. Elsevier, 1993.

[15] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2007.

[16] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[17] A. Czumaj. Parallel algorithm for the matrix chain product and the optimal triangulation problems (extended abstract). In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *STACS*, volume 665 of *Lecture Notes in Computer Science*, pages 294–305. Springer, 1993.

[18] G. Dowek. The undecidability of pattern matching in calculi where primitive recursive functions are representable. *Theor. Comput. Sci.*, 107(2):349–356, Jan. 1993.

[19] D. Eddelbuettel, H. Yu, L. Tierney, U. Mansmann, M. Schmidberger, and M. Morgan. State-of-the-art in parallel computing with r. Technical Report 47, Department of Statistics, University of Munich, 2009.

[20] C. Ferdinand, H. Seidl, and R. Wilhelm. Tree automata for code selection. *Acta Inf.*, 31(8):741–760, 1994.

[21] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than o(1) dependency. *J. Parallel Distrib. Comput.*, 21:213–222, May 1994.

[22] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[23] M. Gengler. An introduction to parallel dynamic programming. In A. Ferreira and P. M. Pardalos, editors, *Solving Combinatorial Optimization Problems in Parallel*, volume 1054 of *Lecture Notes in Computer Science*, pages 87–114. Springer, 1996.

[24] A. Gibbons and W. Rytter. *Efficient parallel algorithms*. Cambridge University Press, 1988.

[25] D. González-Morales, F. Almeida, F. Garcia, J. Gonzalez, J. L. Roda, and C. Rodríguez. A skeleton for parallel dynamic programming. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, Euro-Par '99, pages 877–887, London, UK, UK, 1999. Springer-Verlag.

[26] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40:1135–1160, November 2010.

[27] S. Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, October 2007.

[28] P. Helman. A common schema for dynamic programming and branch and bound algorithms. *J. ACM*, 36(1):97–128, 1989.

[29] S. H. S. Huang, H. Liu, and V. Viswanathan. Parallel dynamic programming. *IEEE Trans. Parallel Distrib. Syst.*, 5:326–328, March 1994.

[30] T. Ibaraki. Enumerative approaches to combinatorial optimization - part ii. *Ann. Oper. Res.*, 11:345–602, January 1988.

[31] K. Kakehi, K. Matsuzaki, A. Morihata, K. Emoto, and Z. Hu. Parallel dynamic programming using data-parallel skeletons. In *Proceedings of the 22nd JSSST Conference*, 2005.

[32] R. M. Karp and M. H. Held. Finite-State Processes and Dynamic Programming. *SIAM Journal of Applied Mathematics*, 15:693–718, 1967.

[33] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[34] M. Mirold. An experimentation laboratory for the automatic parallelization of programs written in the r language (alchemy). Master's thesis, 2011.

[35] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[36] J. Paakki. Attribute grammar paradigms – a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, June 1995.

[37] I. Peláez, F. Almeida, and D. González. High level parallel skeletons for dynamic programming. *Parallel Processing Letters*, 18(1):133–147, 2008.

[38] I. Peláez, F. Almeida, and F. Suárez. Dpskel: a skeleton based tool for parallel dynamic programming. In *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, PPAM'07, pages 1104–1113, Berlin, Heidelberg, 2008. Springer-Verlag.

[39] W. C. Rounds. Mappings and grammars on trees. *Mathematical Systems Theory*, 4(3):257–287, 1970.

[40] W. Rytter. On efficient parallel computations for some dynamic programming problems. *Theor. Comput. Sci.*, 59:297–307, 1988.

[41] D. B. Skillicorn. The bird-meertens formalism as a parallel model. In *Software for Parallel Computation, volume 106 of NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.

[42] G. Tan, N. Sun, and G. R. Gao. A parallel dynamic programming algorithm on a multi-core architecture. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 135–144, New York, NY, USA, 2007. ACM.

[43] B. W. Wah, G. jie Li, and C. F. Yu. Multiprocessing of combinatorial search problems. *Computer*, 18:93–108, June 1985.

A. The full attribute grammar

First, the full grammar of the AIR, G_{AIR}:

AIRExpr	→	ForStmt | IfExpr | BinopExpr | UnaryExpr | FuncCall | FuncDef | SubscriptExpr | SymbolExpr | ConstantExpr | ExprList
Program	→	program(AIRExpr+)
ParamExpr	→	paramexpr(name?, AIRExpr)
ExprList	→	exprlist(AIRExpr+)
ForStmt	→	forstmt(IteratorExpr, AIRExpr)
IteratorExpr	→	iteratorexpr(SymbolExpr, AIRExpr)
IfExpr	→	ifexpr(*condition*=AIRExpr, *body*=AIRExpr)
BinopExpr	→	binopexpr(OP, *lhs*=AIRExpr,*rhs*=AIRExpr)
UnaryExpr	→	unaryexpr(OP, AIRExpr)
FuncCall	→	funccall(*funcexpr*=AIRExpr, (ParamExpr)+)
FuncDef	→	funcdef((ParamExpr)+, airexpr)
SubscriptExpr	→	subscriptexpr(AIRExpr, AIRExpr+)
SymbolExpr	→	symbolexpr(name)
ConstantExpr	→	constantexpr(airvalue)

And the attribute grammar rules. Any rule which is not computed is assumed to be $\emptyset$ or $false$:

Grammar rule:
`Program` $\rightarrow$ `program(AIRExpr+)`

Synthesized attributes:
`Program.isDP` := $\bigvee_i$ `AIRExpr[i].isDP`
`Program.DPWitness` := $\bigcup_i$ `AIRExpr[i].DPWitness`

Grammar rule:
`ExprList` $\rightarrow$ `exprlist(AIRExpr+)`

Synthesized attributes:
`ExprList.isDP` := $\bigwedge_i$ `AIRExpr[i].isDP`
`ExprList.DPWitness` := $\bigcup_i$ `AIRExpr[i].DPWitness`

Inherited attributes:
$\forall$i: `AIRExpr[i].LoopVars := ExprList.LoopVars`

Grammar rule:
`ForStmt` $\rightarrow$ `forstmt(IteratorExpr,AIRExpr)`

Synthesized attributes:
`ForStmt.isDP := AIRExpr.isDP`
`ForStmt.DPWitness := AIRExpr.DPWitness`

Inherited attributes:
`AIRExpr.LoopVars` := `ForStmt.LoopVars` $\cup$ `IteratorExpr.usedVars`

Grammar rule:
`IteratorExpr := iteratorexpr(SymbolExpr,AIRExpr)`

Synthesized attributes:
`IteratorExpr.usedVars := SymbolExpr.usedVars`

Grammar rule:
`IfExpr` $\rightarrow$ `ifexpr(condition=AIRExpr,body=AIRExpr)`

Synthesized attributes:
```
IfExpr.isDP := body.isDP
IfExpr.DPWitness := body.DPWitness
IfExpr.matrixToAccessIDs := body.matrixToAccessIDs
IfExpr.usedVars := body.usedVars
```

Inherited attributes:
```
body.LoopVars := IfExpr.LoopVars
```

Grammar rule:
`UnaryExpr` $\rightarrow$ `unaryexpr(op, AIRExpr)`

Synthesized attributes:
```
UnaryExpr.isDP := AIRExpr.isDP
UnaryExpr.matrixToAccessIDs := AIRExpr.matrixToAccessIDs
UnaryExpr.usedVars := AIRExpr.usedVars
```

Inherited attributes:
```
AIRExpr.LoopVars := UnaryExpr.LoopVars
```

Grammar rule:
`FuncCall` $\rightarrow$ `funccall(funcexpr=AIRExpr, ParamExpr+)`

Synthesized attributes:
$\texttt{FuncCall.isDP} := \bigvee_{\texttt{i}} \texttt{ParamExpr[i].isDP}$
$\texttt{UnaryExpr.matrixToAccessIDs} := \bigcup_{\texttt{i}} \texttt{ParamExpr[i].matrixToAccessIDs}$
$\texttt{UnaryExpr.usedVars} := \bigcup_{\texttt{i}} \texttt{ParamExpr[i].usedVars}$

Inherited attributes:
$\forall \texttt{i}$: `ParamExpr[i].LoopVars := FuncCall.LoopVars`

Grammar rule:
`FuncDef → funcdef(ParamExpr+, AIRExpr)`

Synthesized attributes:
```
FuncDef.isDP := AIRExpr.isDP
FuncDef.DPWitness := AIRExpr.DPWitness
FuncDef.isFuncDef := true
```

Grammar rule:
`SymbolExpr → symbolexpr(name)`

Synthesized attributes:
```
SymbolExpr.usedVars := {name}
```

Grammar rule:
`ConstantExpr → constantexpr(airvalue)`

No attribute changes.

Grammar rule:
`SubscriptExpr` $\rightarrow$ `subscriptexpr(col=AIRExpr,AIRExpr+)`

Synthesized attributes:

let

K = `SubscriptExpr.loopVars` $\cap$ ($\bigcup_i$ `AIRExpr[i].usedVars`)

in

`SubscriptExpr.usedVars := {col.usedVars[1]}`

`SubscriptExpr.isDP :=` $K \neq \emptyset$

`SubscriptExpr.matrixToAccessIDs :=`
if `SubscriptExpr.isDP` **then**
`{ col.usedVars[1]` $\rightarrow K$ `}`
else
$\emptyset$

end

Inherited attributes:

`col.LoopVars := SubscriptExpr.LoopVars`

$\forall i$: `AIRExpr[i].LoopVars := SubscriptExpr.LoopVars`

And finally, the full rules for BinopExpr:

Grammar rule:
`BinopExpr` $\rightarrow$ `binopexpr(assign, rhs=AIRExpr, lhs=AIRExpr)`

Synthesized attributes:

$$\texttt{BinopExpr.isDP} := \begin{array}{l}(\texttt{lhs.matrixToAccessIDs(l.usedVars[1])} \\ \subseteq \texttt{rhs.matrixToAccessIDs(l.usedVars[1])} \\ \wedge\ \texttt{lhs.isDP} \wedge \texttt{rhs.isDP}) \\ \vee\ (\texttt{rhs.isDP} \wedge \texttt{rhs.isFuncDef})\end{array}$$

$$\texttt{BinopExpr.DPWitness} := \begin{array}{l}\textbf{if}\ \texttt{BinopExpr.isDP} \wedge \neg\ \texttt{rhs.isFuncDef}\ \textbf{then} \\ \quad \{\ (\texttt{binopexpr}, \\ \qquad \texttt{l.usedVars[1]}, \\ \qquad \texttt{l.matrixToAccessIDs(l.usedVars[1])} \\ \quad) \\ \quad \} \\ \textbf{elseif}\ \texttt{BinopExpr.isDP} \wedge \texttt{rhs.isFuncDef}\ \textbf{then} \\ \quad \texttt{rhs.DPWitness} \\ \textbf{else} \\ \quad \emptyset\end{array}$$

Inherited attributes:
`lhs.LoopVars := BinopExpr.LoopVars`
`rhs.LoopVars := BinopExpr.LoopVars`

Grammar rule:
`BinopExpr` $\rightarrow$ `binopexpr(OP`$\neq$`assign,lhs=AIRExpr,rhs=AIRExpr)`

Synthesized attributes:
`BinopExpr.isDP := lhs.isDP` $\vee$ `rhs.isDP`
`BinopExpr.usedVars := lhs.usedVars` $\cup$ `rhs.usedVars`

$$\texttt{BinopExpr.matrixToAccessIDs} := \begin{array}{l}\texttt{lhs.matrixToAccessIDs} \cup \\ \texttt{rhs.matrixToAccessIDs}\end{array}$$

Inherited attributes:
`lhs.LoopVars := BinopExpr.LoopVars`
`rhs.LoopVars := BinopExpr.LoopVars`

B. Problems with the approach of Kaheki, Matsuzaki et. al.

We use example (2) from the authors' own paper [31]. That is, the DP formula is the function:

$$D[i,j] = \min \begin{cases} D[i,j-1] + w[i,j] \\ D[i-1,j] + nw[i,j] \\ D[i-1,j-1] + n[i,j] \end{cases} \quad \text{for } \begin{matrix} 0 \le i \le N \\ 0 \le j \le M \end{matrix}$$

They define $\odot = +$, and $\otimes = \oplus = \downarrow = min$ and $g^* = h = 1$. I.e., $E[i,j]$ will be evaluated as:

$$E[i,j] = min((E[i-1,j] + w[i,j]), F[i,j])$$

And $E[,j]$ will be evaluated with skeletons as:

$$E[,j] = scan(\ominus, zipw(\lambda(a,b).(a,b), F[,j], w[,j]))$$
$$\text{where} (c_l, w_l) \ominus (c_r, w_r) = (min(c_l + w_l, c_r),\ w_l + w_r)$$

Furthermore, assume we already resolved the intergenerational dependencies of the j-th generation. That is, the matrix F currently looks as follows:

$$F = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & \cdots & j & \cdots \end{array} \\ \begin{pmatrix} F[1,1] & \cdots & 3 & \cdots \\ \vdots & \cdots & 5 & \cdots \\ \vdots & \cdots & 2 & \cdots \\ \vdots & \cdots & 17 & \cdots \\ \vdots & \cdots & 9 & \cdots \\ \vdots & \cdots & 2 & \cdots \\ \vdots & \cdots & 10 & \cdots \end{pmatrix} \end{array}$$

And $w[,j]$ as follows:

$$w[,j] = [2,5,1,1,9,2,3]$$

Let us first compute the values $E[,j]$ manually. We get the following matrix as result:

$$
E = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & \cdots & j & \cdots \end{array} \\
\begin{pmatrix}
F[1,1] & \cdots & 3 & \cdots \\
\vdots & \cdots & min(3+2,5) & \cdots \\
\vdots & \cdots & ? & \cdots \\
\vdots & \cdots & ? & \cdots \\
\vdots & \cdots & ? & \cdots \\
\vdots & \cdots & ? & \cdots \\
\vdots & \cdots & ? & \cdots
\end{pmatrix}
\end{array}
$$

$$
= \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & \cdots & j & \cdots \end{array} \\
\begin{pmatrix}
F[1,1] & \cdots & 3 & \cdots \\
\vdots & \cdots & 5 & \cdots \\
\vdots & \cdots & 2 & \cdots \\
\vdots & \cdots & 3 & \cdots \\
\vdots & \cdots & 4 & \cdots \\
\vdots & \cdots & 2 & \cdots \\
\vdots & \cdots & min(2+2,10) & \cdots
\end{pmatrix}
\end{array}
$$

$$
= \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & \cdots & j & \cdots \end{array} \\
\begin{pmatrix}
F[1,1] & \cdots & 3 & \cdots \\
\vdots & \cdots & 5 & \cdots \\
\vdots & \cdots & 2 & \cdots \\
\vdots & \cdots & 3 & \cdots \\
\vdots & \cdots & 4 & \cdots \\
\vdots & \cdots & 2 & \cdots \\
\vdots & \cdots & 4 & \cdots
\end{pmatrix}
\end{array}
$$

However, the computation using scan delivers different results. First, we evaluate the zipw-skeleton expression $zipw(\lambda(a,b).(a,b), F[,j], w[,j])$:

$$E = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & \cdots & j & \cdots \end{array} \\ \begin{pmatrix} F[1,1] & \cdots & (3,2) & \cdots \\ \vdots & \cdots & (5,5) & \cdots \\ \vdots & \cdots & (2,1) & \cdots \\ \vdots & \cdots & (17,1) & \cdots \\ \vdots & \cdots & (9,9) & \cdots \\ \vdots & \cdots & (2,2) & \cdots \\ \vdots & \cdots & (10,3) & \cdots \end{pmatrix} \end{array}$$

Then, we evaluate $E[,j] = scan((min(c_l + w_l, c_r),\ w_l + w_r), E[,j])$:

$$
E = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & \cdots & j & \cdots \end{array} \\
\begin{pmatrix}
F[1,1] & \cdots & (3,2) & \cdots \\
\vdots & \cdots & (min(3+2,5),2+5) & \cdots \\
\vdots & \cdots & (2,1) & \cdots \\
\vdots & \cdots & (17,1) & \cdots \\
\vdots & \cdots & (9,9) & \cdots \\
\vdots & \cdots & (2,2) & \cdots \\
\vdots & \cdots & (10,3) & \cdots
\end{pmatrix}
\end{array}
$$

$$
= \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & \cdots & j & \cdots \end{array} \\
\begin{pmatrix}
F[1,1] & \cdots & (3,2) & \cdots \\
\vdots & \cdots & (5,2+5=7) & \cdots \\
\vdots & \cdots & (2,7+1=8) & \cdots \\
\vdots & \cdots & (min(2+8,17),8+1=9) & \cdots \\
\vdots & \cdots & (9,9) & \cdots \\
\vdots & \cdots & (2,2) & \cdots \\
\vdots & \cdots & (10,3) & \cdots
\end{pmatrix}
\end{array}
$$

$$
= \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & \cdots & j & \cdots \end{array} \\
\begin{pmatrix}
F[1,1] & \cdots & (3,2) & \cdots \\
\vdots & \cdots & (5,7) & \cdots \\
\vdots & \cdots & (2,8) & \cdots \\
\vdots & \cdots & (\mathbf{10},9) & \cdots \\
\vdots & \cdots & (9,9) & \cdots \\
\vdots & \cdots & (2,2) & \cdots \\
\vdots & \cdots & (10,3) & \cdots
\end{pmatrix}
\end{array}
$$

At this point, we see that the matrix evaluates differently. Instead of having $E[4, j] = 3$, it evaluates to $E[4, j] = (10,9)$. Since the first value of the tuple represents the value that $E[4, j]$ should have, the computation is wrong.

B.1. Changes in our version

To deal with this issue, we only use inner functions for scan which do not depend on the index i. In particular, we cannot evaluate the function $w[i,j]$. We will then re-define the DAG matrix formalization as:

$$E[i,j] = f_E(E[i-l_g^*, j]) \otimes F[i,j] \quad \text{where } l_g^* \in \mathbb{N}_+ \tag{b}$$

$$F[i,j] = \bigotimes_{k \neq E} f_k(E[i-l_k^*, j-g_k^*], i, j, k) \quad \text{where } g_k^* \in \mathbb{N}_+ \tag{a}$$

And, consequently, its implementation with skeletons to:

Procedure B.1 Computation of the DAG matrix formalization (Unrestricted)

1: $F[,j] = \text{zipw}(\otimes, c_{l_1^*,g_1^*}, ...(\text{zipw}(\otimes, c_{l_{k-1}^*,g_{k-1}^*}, c_{l_k^*,g_k^*})))$
2: $E[,j] = \text{map}(\lambda(a,b).b, \text{scan}(\lambda((i_1,x),(i_2,y)).(i_2, f_E(i_1,x) \oplus y), idx(F[,j])))$
3: **where**
4: $c_{l_k^*,g_k^*} = \text{map}(\lambda(i,x).f_k(x,i,j,k), idx(\text{shift}(l_k^*, E[,j-g_k^*])))$
5: $idx(L) = \text{zipw}(\lambda(a,b).(a,b), [1,...,N], L)$

Finally, we would like to note that it is possible to optimize in this scenario if f_E does not have any dependencies to i:

Procedure B.2 Computation of the DAG matrix formalization

1: $F[,j] = \text{zipw}(\otimes, c_{l_1^*,g_1^*}, ...(\text{zipw}(\otimes, c_{l_{k-1}^*,g_{k-1}^*}, c_{l_k^*,g_k^*})))$
2: $E[,j] = \text{scan}(\lambda(a,b).(f_E(a) \oplus b), F[,j])$
3: **where**
4: $c_{l_k^*,g_k^*} = \text{map}(\lambda(i,x).f_k(x,i,j,k), idx(\text{shift}(l_k^*, E[,j-g_k^*])))$
5: $idx(L) = \text{zipw}(\lambda(a,b).(a,b), [1,...,N], L)$

C. AIR-XML trees of program inputs and their translations

C.1. Input program

Listing C.1: Input for parallelization.

```
<AIR>
 <Program>
  <BinopExpr op="&lt;-">
   <lhs>
    <SymbolExpr name="a"/>
   </lhs>
   <rhs>
    <FuncDef>
     <params/>
     <body>
      <ExprList>
       <ForStmt>
        <condition>
         <IteratorExpr>
          <itervar>
           <SymbolExpr name="i"/>
          </itervar>
          <collection>
           <FuncCall>
            <funcexpr>
             <SymbolExpr name=":"/>
            </funcexpr>
            <params>
             <ParamExpr>
              <ConstantExpr type="real">
               <RealValue data="1.000000"/>
              </ConstantExpr>
             </ParamExpr>
             <ParamExpr>
              <SymbolExpr name="N"/>
             </ParamExpr>
            </params>
           </FuncCall>
          </collection>
         </IteratorExpr>
        </condition>
        <body>
```

```
38          <ExprList>
39           <ForStmt>
40            <condition>
41             <IteratorExpr>
42              <itervar>
43               <SymbolExpr name="j"/>
44              </itervar>
45              <collection>
46               <FuncCall>
47                <funcexpr>
48                 <SymbolExpr name=":"/>
49                </funcexpr>
50                <params>
51                 <ParamExpr>
52                  <ConstantExpr type="real">
53                   <RealValue data="1.000000"/>
54                  </ConstantExpr>
55                 </ParamExpr>
56                 <ParamExpr>
57                  <SymbolExpr name="M"/>
58                 </ParamExpr>
59                </params>
60               </FuncCall>
61              </collection>
62             </IteratorExpr>
63            </condition>
64            <body>
65             <ExprList>
66              <BinopExpr op="=">
67               <lhs>
68                <SubscriptExpr>
69                 <collection>
70                  <SymbolExpr name="D"/>
71                 </collection>
72                 <subscripts>
73                  <SymbolExpr name="i"/>
74                  <SymbolExpr name="j"/>
75                 </subscripts>
76                </SubscriptExpr>
77               </lhs>
78               <rhs>
79                <FuncCall>
80                 <funcexpr>
81                  <SymbolExpr name="max"/>
82                 </funcexpr>
83                 <params>
84                  <ParamExpr>
85                   <FuncCall>
86                    <funcexpr>
87                     <SymbolExpr name="max"/>
88                    </funcexpr>
89                    <params>
90                     <ParamExpr>
91                      <BinopExpr op="+">
```

```
                        <lhs>
                         <SubscriptExpr>
                          <collection>
                           <SymbolExpr name="D"/>
                          </collection>
                          <subscripts>
                           <SymbolExpr name="i"/>
                           <BinopExpr op="-">
                            <lhs>
                             <SymbolExpr name="j"/>
                            </lhs>
                            <rhs>
                             <ConstantExpr type="real">
                              <RealValue
data="1.000000"/>
                             </ConstantExpr>
                            </rhs>
                           </BinopExpr>
                          </subscripts>
                         </SubscriptExpr>
                        </lhs>
                        <rhs>
                         <ConstantExpr type="real">
                          <RealValue data="2.000000"/>
                         </ConstantExpr>
                        </rhs>
                       </BinopExpr>
                      </ParamExpr>
                      <ParamExpr>
                       <BinopExpr op="+">
                        <lhs>
                         <SubscriptExpr>
                          <collection>
                           <SymbolExpr name="D"/>
                          </collection>
                          <subscripts>
                           <BinopExpr op="-">
                            <lhs>
                             <SymbolExpr name="i"/>
                            </lhs>
                            <rhs>
                             <ConstantExpr type="real">
                              <RealValue
data="1.000000"/>
                             </ConstantExpr>
                            </rhs>
                           </BinopExpr>
                           <SymbolExpr name="j"/>
                          </subscripts>
                         </SubscriptExpr>
                        </lhs>
                        <rhs>
                         <ConstantExpr type="real">
                          <RealValue data="3.000000"/>
```

```
                        </ConstantExpr>
                       </rhs>
                      </BinopExpr>
                     </ParamExpr>
                    </params>
                   </FuncCall>
                  </ParamExpr>
                  <ParamExpr>
                   <BinopExpr op="+">
                    <lhs>
                     <SubscriptExpr>
                      <collection>
                       <SymbolExpr name="D"/>
                      </collection>
                      <subscripts>
                       <BinopExpr op="-">
                        <lhs>
                         <SymbolExpr name="i"/>
                        </lhs>
                        <rhs>
                         <ConstantExpr type="real">
                          <RealValue data="1.000000"/>
                         </ConstantExpr>
                        </rhs>
                       </BinopExpr>
                       <BinopExpr op="-">
                        <lhs>
                         <SymbolExpr name="j"/>
                        </lhs>
                        <rhs>
                         <ConstantExpr type="real">
                          <RealValue data="1.000000"/>
                         </ConstantExpr>
                        </rhs>
                       </BinopExpr>
                      </subscripts>
                     </SubscriptExpr>
                    </lhs>
                    <rhs>
                     <ConstantExpr type="real">
                      <RealValue data="1.000000"/>
                     </ConstantExpr>
                    </rhs>
                   </BinopExpr>
                  </ParamExpr>
                 </params>
                </FuncCall>
               </rhs>
              </BinopExpr>
             </ExprList>
            </body>
           </ForStmt>
          </ExprList>
         </body>
```

```
        </ForStmt>
       </ExprList>
      </body>
     </FuncDef>
    </rhs>
   </BinopExpr>
  </Program>
</AIR>
```

C.2. Output of MATSU

Listing C.2: Output of MATSU.

```
<AIR>
 <Program>
  <BinopExpr op="&lt;-">
   <lhs>
    <SymbolExpr name="a"/>
   </lhs>
   <rhs>
    <FuncDef>
     <params/>
     <body>
      <ExprList>
       <ForStmt>
        <condition>
         <IteratorExpr>
          <iterator>
           <SymbolExpr name="j"/>
          </iterator>
          <collection>
           <FuncCall>
            <funcexpr>
             <SymbolExpr name="dim"/>
            </funcexpr>
            <params>
             <ParamExpr>
              <SymbolExpr name="E"/>
             </ParamExpr>
            </params>
           </FuncCall>
          </collection>
         </IteratorExpr>
        </condition>
        <body>
         <ExprList>
          <BinopExpr op="&lt;-">
           <lhs>
            <SubscriptExpr>
             <collection>
              <SymbolExpr name="F"/>
```

```
39              </collection>
40              <subscripts>
41               <SymbolExpr name=""/>
42               <SymbolExpr name="j"/>
43              </subscripts>
44             </SubscriptExpr>
45            </lhs>
46            <rhs>
47             <FuncCall>
48              <funcexpr>
49               <SymbolExpr name="zipw"/>
50              </funcexpr>
51              <params>
52               <ParamExpr>
53                <SymbolExpr name="max"/>
54               </ParamExpr>
55               <ParamExpr>
56                <FuncCall>
57                 <funcexpr>
58                  <SymbolExpr name="shift"/>
59                 </funcexpr>
60                 <params>
61                  <ParamExpr>
62                   <ConstantExpr type="real">
63                    <RealValue data="1.0"/>
64                   </ConstantExpr>
65                  </ParamExpr>
66                  <ParamExpr>
67                   <FuncCall>
68                    <funcexpr>
69                     <SymbolExpr name="zipw"/>
70                    </funcexpr>
71                    <params>
72                     <ParamExpr>
73                      <FuncDef>
74                       <params>
75                        <ParamExpr name="i"/>
76                        <ParamExpr name="_x"/>
77                       </params>
78                       <body>
79                        <BinopExpr op="+">
80                         <lhs>
81                          <SymbolExpr name="_x"/>
82                         </lhs>
83                         <rhs>
84                          <ConstantExpr type="real">
85                           <RealValue data="1.0"/>
86                          </ConstantExpr>
87                         </rhs>
88                        </BinopExpr>
89                       </body>
90                      </FuncDef>
91                     </ParamExpr>
92                     <ParamExpr>
```

```
                      <FuncCall>
                       <funcexpr>
                        <SymbolExpr name="dim"/>
                       </funcexpr>
                       <params>
                        <ParamExpr>
                         <SymbolExpr name="E"/>
                        </ParamExpr>
                       </params>
                      </FuncCall>
                     </ParamExpr>
                     <ParamExpr>
                      <SubscriptExpr>
                       <collection>
                        <SymbolExpr name="E"/>
                       </collection>
                       <subscripts>
                        <SymbolExpr name=""/>
                        <BinopExpr op="-">
                         <lhs>
                          <SymbolExpr name="j"/>
                         </lhs>
                         <rhs>
                          <ConstantExpr type="real">
                           <RealValue data="1.0"/>
                          </ConstantExpr>
                         </rhs>
                        </BinopExpr>
                       </subscripts>
                      </SubscriptExpr>
                     </ParamExpr>
                    </params>
                   </FuncCall>
                  </ParamExpr>
                  <ParamExpr>
                   <ConstantExpr type="real">
                    <RealValue data="4.9E-324"/>
                   </ConstantExpr>
                  </ParamExpr>
                 </params>
                </FuncCall>
               </ParamExpr>
               <ParamExpr>
                <FuncCall>
                 <funcexpr>
                  <SymbolExpr name="zipw"/>
                 </funcexpr>
                 <params>
                  <ParamExpr>
                   <FuncDef>
                    <params>
                     <ParamExpr name="i"/>
                     <ParamExpr name="_x"/>
                    </params>
```

```
                    <body>
                     <BinopExpr op="+">
                      <lhs>
                       <SymbolExpr name="_x"/>
                      </lhs>
                      <rhs>
                       <ConstantExpr type="real">
                        <RealValue data="2.0"/>
                       </ConstantExpr>
                      </rhs>
                     </BinopExpr>
                    </body>
                   </FuncDef>
                  </ParamExpr>
                  <ParamExpr>
                   <FuncCall>
                    <funcexpr>
                     <SymbolExpr name="dim"/>
                    </funcexpr>
                    <params>
                     <ParamExpr>
                      <SymbolExpr name="E"/>
                     </ParamExpr>
                    </params>
                   </FuncCall>
                  </ParamExpr>
                  <ParamExpr>
                   <SubscriptExpr>
                    <collection>
                     <SymbolExpr name="E"/>
                    </collection>
                    <subscripts>
                     <SymbolExpr name=""/>
                     <BinopExpr op="-">
                      <lhs>
                       <SymbolExpr name="j"/>
                      </lhs>
                      <rhs>
                       <ConstantExpr type="real">
                        <RealValue data="1.0"/>
                       </ConstantExpr>
                      </rhs>
                     </BinopExpr>
                    </subscripts>
                   </SubscriptExpr>
                  </ParamExpr>
                 </params>
                </FuncCall>
               </ParamExpr>
              </params>
             </FuncCall>
            </rhs>
           </BinopExpr>
           <BinopExpr op="&lt;-">
```

```
201             <lhs>
202              <SubscriptExpr>
203               <collection>
204                <SymbolExpr name="E"/>
205               </collection>
206               <subscripts>
207                <SymbolExpr name=""/>
208                <SymbolExpr name="j"/>
209               </subscripts>
210              </SubscriptExpr>
211             </lhs>
212             <rhs>
213              <FuncCall>
214               <funcexpr>
215                <SymbolExpr name="map"/>
216               </funcexpr>
217               <params>
218                <ParamExpr>
219                 <FuncDef>
220                  <params>
221                   <ParamExpr name="a"/>
222                  </params>
223                  <body>
224                   <SubscriptExpr>
225                    <collection>
226                     <SymbolExpr name="a"/>
227                    </collection>
228                    <subscripts>
229                     <ConstantExpr type="real">
230                      <RealValue data="2.0"/>
231                     </ConstantExpr>
232                    </subscripts>
233                   </SubscriptExpr>
234                  </body>
235                 </FuncDef>
236                </ParamExpr>
237                <ParamExpr>
238                 <FuncCall>
239                  <funcexpr>
240                   <SymbolExpr name="scan"/>
241                  </funcexpr>
242                  <params>
243                   <ParamExpr>
244                    <FuncDef>
245                     <params>
246                      <ParamExpr name="_d"/>
247                      <ParamExpr name="_e"/>
248                     </params>
249                     <body>
250                      <FuncCall>
251                       <funcexpr>
252                        <SymbolExpr name="c"/>
253                       </funcexpr>
254                       <params>
```

```
255                         <ParamExpr>
256                          <SymbolExpr name="_e"/>
257                         </ParamExpr>
258                         <ParamExpr>
259                          <FuncCall>
260                           <funcexpr>
261                            <SymbolExpr name="max"/>
262                           </funcexpr>
263                           <params>
264                            <ParamExpr>
265                             <FuncCall>
266                              <funcexpr>
267                               <ClosureExpr>
268                                <body>
269                                 <BinopExpr op="+">
270                                  <lhs>
271                                   <SymbolExpr name="_x"/>
272                                  </lhs>
273                                  <rhs>
274                                   <ConstantExpr type="real">
275                                    <RealValue data="3.0"/>
276                                   </ConstantExpr>
277                                  </rhs>
278                                 </BinopExpr>
279                                </body>
280                                <params>
281                                 <ParamExpr name="i"/>
282                                 <ParamExpr name="_x"/>
283                                </params>
284                               </ClosureExpr>
285                              </funcexpr>
286                              <params>
287                               <ParamExpr>
288                                <SubscriptExpr>
289                                 <collection>
290                                  <SymbolExpr name="_d"/>
291                                 </collection>
292                                 <subscripts>
293                                  <ConstantExpr type="real">
294                                   <RealValue data="1.0"/>
295                                  </ConstantExpr>
296                                 </subscripts>
297                                </SubscriptExpr>
298                               </ParamExpr>
299                               <ParamExpr>
300                                <SubscriptExpr>
301                                 <collection>
302                                  <SymbolExpr name="_d"/>
303                                 </collection>
304                                 <subscripts>
305                                  <ConstantExpr type="real">
306                                   <RealValue data="2.0"/>
307                                  </ConstantExpr>
308                                 </subscripts>
```

```
                                </SubscriptExpr>
                               </ParamExpr>
                              </params>
                             </FuncCall>
                            </ParamExpr>
                            <ParamExpr>
                             <SubscriptExpr>
                              <collection>
                               <SymbolExpr name="_e"/>
                              </collection>
                              <subscripts>
                               <ConstantExpr type="real">
                                <RealValue data="2.0"/>
                               </ConstantExpr>
                              </subscripts>
                             </SubscriptExpr>
                            </ParamExpr>
                           </params>
                          </FuncCall>
                         </ParamExpr>
                        </params>
                       </FuncCall>
                      </body>
                     </FuncDef>
                    </ParamExpr>
                    <ParamExpr>
                     <FuncCall>
                      <funcexpr>
                       <SymbolExpr name="zipw"/>
                      </funcexpr>
                      <params>
                       <ParamExpr>
                        <FuncDef>
                         <params>
                          <ParamExpr>
                           <SymbolExpr name="_a"/>
                          </ParamExpr>
                          <ParamExpr>
                           <SymbolExpr name="_b"/>
                          </ParamExpr>
                         </params>
                         <body>
                          <FuncCall>
                           <funcexpr>
                            <SymbolExpr name="c"/>
                           </funcexpr>
                           <params>
                            <ParamExpr>
                             <SymbolExpr name="_a"/>
                            </ParamExpr>
                            <ParamExpr>
                             <SymbolExpr name="_b"/>
                            </ParamExpr>
                           </params>
```

```
                        </FuncCall>
                       </body>
                      </FuncDef>
                     </ParamExpr>
                     <ParamExpr>
                      <FuncCall>
                       <funcexpr>
                        <SymbolExpr name="dim"/>
                       </funcexpr>
                       <params>
                        <ParamExpr>
                         <SymbolExpr name="E"/>
                        </ParamExpr>
                       </params>
                      </FuncCall>
                     </ParamExpr>
                     <ParamExpr>
                      <SubscriptExpr>
                       <collection>
                        <SymbolExpr name="F"/>
                       </collection>
                       <subscripts>
                        <SymbolExpr name=""/>
                        <SymbolExpr name="j"/>
                       </subscripts>
                      </SubscriptExpr>
                     </ParamExpr>
                    </params>
                   </FuncCall>
                  </ParamExpr>
                  <ParamExpr>
                   <ConstantExpr type="real">
                    <RealValue data="1.0"/>
                   </ConstantExpr>
                  </ParamExpr>
                 </params>
                </FuncCall>
               </ParamExpr>
              </params>
             </FuncCall>
            </rhs>
           </BinopExpr>
          </ExprList>
         </body>
        </ForStmt>
       </ExprList>
      </body>
     </FuncDef>
    </rhs>
   </BinopExpr>
  </Program>
</AIR>
```