**Автономная некоммерческая организация высшего образования**
**«Университет Иннополис»**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**
**(БАКАЛАВРСКАЯ РАБОТА)**
**по направлению подготовки**
**09.03.01 - «Информатика и вычислительная техника»**

**GRADUATION THESIS**
**(BACHELOR'S GRADUATION THESIS)**
**Field of Study**
**09.03.01 – «Computer Science»**

**Направленность (профиль) образовательной программы**
**«Информатика и вычислительная техника»**
**Area of Specialization / Academic Program Title:**
**«Computer Science»**

| | |
|---|---|
| **Тема / Topic** | **Манипуляция параллельным по построению деревом в Haskell / Parallel-by-construction tree manipulation in Haskell** |

| | | |
|---|---|---|
| Работу выполнил / Thesis is executed by | **Дворянов Даниил Сергеевич / Dvorianov Danyil** | подпись / signature |
| Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis | **Маццара Мануэль / Mazzara Manuel** | подпись / signature |
| Консультанты / Consultants | **Кудасов Николай Дмитриевич / Kudasov Nikolai** | подпись / signature |

Иннополис, Innopolis, 2021

# Contents

# List of Tables

# List of Figures

**Abstract**

This research is inspired by the work of A. Hsu on parallel compilers in APL [1]. His findings are interesting, although might be underappreciated due to the unpopularity and complexity of APL[1]. Therefore, I attempt to not only replicate his results, but to provide a purely-functional, Haskell-based implementation for parallel compilation steps, such as flattening expressions or lifting functions.

My implementation, although needs optimisation, provides an easy-to-use functions for construction and manipulation of array-based representation of a tree. Moreover, with little to no effort a variety of functions may be used as standalone, especially general ones, such as innerProduct or selectRows.

My implementation is available in GitHub via link: https://github.com/DanProgrammer1999/parallel-compiler-haskell

[1]Wikipedia: https://en.wikipedia.org/wiki/APL_(programming_language)

# Chapter 1

# Introduction

In this research I took inspiration from the article of A. Hsu [1]. The author describes an efficient and novel approach to an important compilation step using APL. Aaron's main idea is that the resulting program eliminates branching and recursion from parsing, which allows it to efficiently run on CPU or GPU.

In my research, I attempt to not only replicate his approach in Haskell with Accelerate library, but to create a purely-functional and modular interface to an efficient array-based syntax tree manipulation. For this purpose, I provide a toolset of functions and operations on parallel-by-construction tree representation.

To understand the importance of my research goal, let me first describe several concepts, closely related to the topic.

## 1.1 Trees

Trees are one of the most important concepts in programming, second only to, probably, lists or arrays. They are used for a variety of things, such as

compilers, JSON parsing, fast searching, and many others.

In general, a tree is a mathematical structure similar to graph [2]. In fact, it can be described as a simple, connected, acyclic, and undirected graph.

On of the most common definition of a tree in computer science is a collection of nodes, where each node has a value, and a possibly empty list of children. This is a recursive definition, as each child is also a node, and can have another list of child nodes, and so on. Therefore, the most convenient and straightforward way to operate on a tree is using recursive algorithms.

For example, a traditional AST is traversed and transformed multiple times during various compiler stages, such as flattening expressions, lifting functions, folding constants, and others. With traditional approaches, most of the traversals require recursion over the whole tree and conditional branching. However, both recursion and branching prevent compilers from efficient, parallel execution on multi-core CPU or GPU devices.

Historically, Haskell is an excellent choice for dealing with trees, as well as other recursive structures. Haskell's focus on recursion optimization, as well as convenient mechanisms of pattern matching, allow for straightforward and efficient tree manipulation algorithms. Moreover, since ASTs result from parsing a language, we can use an extensive Haskell tool set for implementing a parser, further streamlining the process.

However, even though recursive approach is the most straightforward, it is not the most efficient one. Moreover, it is almost impossible to enable the recursive algorithm to run in parallel.

## 1.2 Different Approach to AST

Recursive structure is not the only way to represent a tree in software. Much like it is possible to describe a graph using adjacency matrix, a 2-dimensional array, it is as well possible to come up with an efficient way to describe a tree using arrays. Depending on the purpose, there are a variety of ways to do so.

In the case of AST, abstract syntax tree, A. Hsu [1] has designed a very efficient structure for storing and operating on a tree. He describes node coordinates, a matrix of values, where each row describes a unique address, almost a path, to a node. This allows to easily find out if one node is a sibling to the other, does it appear higher than the other, is it a direct or an indirect ancestor of the other, etc.

By applying different transformations and operations on the node coordinates matrix, it is possible to perform several compiler stages, such as expression flattening, function lifting, and constants folding, without any recursion and branching.

The algorithm consists of several steps:

- convert a recursive representation to an array representation with 3 columns: node depth in the tree, and its type and value. A. Hsu [1] uses preorder traversal [1] for this.

- convert depth vector to node coordinate matrix. This includes several steps:

  - construct boolean matrix from depth vector. That is, a matrix where

[1] Wikipedia: https://en.wikipedia.org/wiki/Tree_traversal

each row corresponds to a depth, and has 1 at $i = depth$, and 0 everywhere else

- compute cumulative sum along the columns of the matrix obtained in the previous step
- finally, drop values (replace with 0) at each row where $i > depth$

- find the node coordinates of the nodes we are interested in (e.g. in case of expression flattening, nodes with type 'Expr')
- transform the node coordinates matrix into parent coordinates matrix, where each row would specify the coordinates of the node's parent. It is easy to do: just replace with zero the last non-zero element at each row
- using parent coordinates matrix, compute an inner product with the focus nodes, such that: for each row $i$, each element $j$ is 1 if $focus\_nodes_j$ is an ancestor of the $node\_coordinates_i$, otherwise 0. Parent coordinates matrix is used to ensure that a node is not marked as an ancestor to itself
- using the result of the previous step, find for each row the column index of the last non-zero element
- the indices obtained in the previous step signify the closest ancestor of the desired type. Use this vector as indices into node coordinates matrix to construct the matrix of closest focus ancestors
- finally, use the key operator to group the node coordinates by the closest focus ancestors

## 1.3 My Contribution

In this research, I attempt to not only replicate the results of A. Hsu [1] for compiler stages, but to provide a declarative-style, purely-functional and easy to use interface to array-based operations on any tree. This work might be a beginning of creating purely-functional library of functions and operators in Haskell, which allows one to avoid dealing with complexities of array programming, while retaining efficiency and parallelisation capabilities of array programming language.

Therefore, my research question is: is it possible to build purely functional, high-level Haskell interface to highly parallelizable array computational language (such as Accelerate)? And is it possible to do so without sacrificing efficiency?

# Chapter 2

# Literature Review

In this chapter I enlist and review the work already done on parallel-by-construction trees, as well as explain the purpose and novelty of my work.

## 2.1 Compilers in Haskell

One of the reasons for choosing Haskell as a primary language for this research is that the language is often considered to be a great tool for writing a compiler. There exist many efficient easy to use libraries [3] for various stages of compilation. Moreover, the declarative style and expressivity of the Haskell language make it a good choice for writing compilers.

### 2.1.1 Functional quantum programming language

One example of a compiler written in Haskell is Quantum programming language (QML) [4]. J. Grattage uses Haskell to implement a functional language for quantum computations. This language can be used to write functional-style programs and compile them to an annotated quantum circuit.

## 2.2 The Key to a Data Parallel Compiler

Research that is of particular importance to this topic is described in article [1]. A. Hsu presents an approach to an inherently data-parallel compiler and provides the details of its implementation in an array programming language (APL). His idea is simple but powerful. Let me describe it briefly here with an example (fig. 2.1) from the article [1].

### 2.2.1 Algorithm outline

- Given an AST, construct a depth vector, as in fig. I
- Construct node coordinate matrix, where each row is a unique coordinate for a node. This can be done in 3 steps (as shown by example in fig. II:
  - For a given depth vector, construct a corresponding boolean matrix with dimensions ($node\_count\ X\ max\_depth$). We can describe each row by the function of corresponding depth d: $f(d) = [if\ d = i\ then\ 1\ else\ 0 \mid i \in [0, max\_depth)]$. You can see this matrix for our example in part 1 of II
  - Accumulate the values along the column. In other words, in each new column $j$, $X'_{0,j} = X_{0,j},\ and\ X'_{i,j} = X_{i-1,j} + X'_{i,j}$. You can see the result in part 2 of II
  - Finally, we need to fix some inaccuracies introduced by the previous step. Since each row is a unique address, and can be understood as a path from root to the node, each row needs not to be longer than the depth of the corresponding node. In other words, $\forall\, i \in$

$[0, node\_count),\ j \in (depth\_vec[i], max\_depth) : X_{i,j} = 0$. You can see the result in part 3 of II

- Find the coordinates of focus nodes (e.g. either expression or function nodes). See fig. III for given example.

- Transform node coordinates so that each coordinate represents the coordinate of its parent. It's easy to do: just set to 0 the last non-zero element in the row. See fig. IV.

- Next important part is finding the closest ancestor from focus nodes. See fig. V for the example of described steps.

  – First, we construct a new matrix, where each row $i$ is a boolean vector, each element $j$ denotes whether the $focus\_nodes_j$ is an ancestor of the node $i$. In other words, $X_{i,j} = and\ [for\ k \in [0, max\_depth) : parent\_nodes_{i,k} == focus\_nodes_{i,k}\ ||\ parent\_nodes_{i,k} == 0]$, where *and* simply folds an array with logical 'and'.

  – In the resulting matrix, the closest ancestor's index (in focus nodes) is the index of the rightmost non-zero element. To get it, we can replace each non-zero element with its index, and then simply retrieve the maximum for each row. We get an array of indices into the *focus_nodes* matrix.

  – Finally, construct the matrix of closest ancestors by indexing the focus nodex matrix with the array from previous step.

- Using the ancestor matrix, group nodes with the same ancestor. (see example in fig. 2.2)

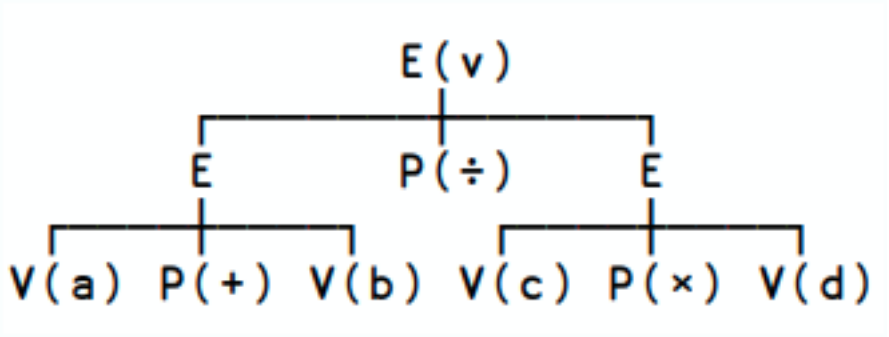


**Figure 2.1:** Example AST

| Depth | Type | Value |
|---|---|---|
| 0 | E | v |
| 1 | E | 0 |
| 2 | V | a |
| 2 | P | + |
| 2 | V | b |
| 1 | P | / |
| 1 | E | 0 |
| 2 | V | c |
| 2 | P | * |
| 2 | V | d |

**Table I:** Depth Vector example

- Now that we have grouped the whole tree into several independent chunks, we can give names to these subgroups and reference them in other subgroups. This helped to reduce the nesting and avoid recursion or branching (which perform poorly on GPU).

### 2.2.2 Key operator

In his work, A. Hsu emphasizes the Key operator [5] as central to his strategy. This operator was first introduced in J language [6], and subsequently implemented in some other programming languages, such as APL. The idea is straightforward: given any 2 arrays a, b of the same outer dimension, and a

| | | |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |
| 0 | 0 | 1 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |
| 0 | 0 | 1 |
| 0 | 0 | 1 |

| | | |
|---|---|---|
| 1 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |
| 1 | 1 | 2 |
| 1 | 1 | 3 |
| 1 | 2 | 3 |
| 1 | 3 | 3 |
| 1 | 3 | 4 |
| 1 | 3 | 5 |
| 1 | 3 | 6 |

| | | |
|---|---|---|
| 1 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |
| 1 | 1 | 2 |
| 1 | 1 | 3 |
| 1 | 2 | 0 |
| 1 | 3 | 0 |
| 1 | 3 | 4 |
| 1 | 3 | 5 |
| 1 | 3 | 6 |

**Table II:** Construction of node coordinates

| | | |
|---|---|---|
| 1 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 3 | 0 |

**Table III:** Node coordinates of nodes of type E (Expression)

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 0 | 0 |
| 1 | 3 | 0 |
| 1 | 3 | 0 |
| 1 | 3 | 0 |

**Table IV:** Node coordinates of nodes of type E (Expression)

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 0 | 1 |
| 1 | 0 | 1 |

| |
|---|
| 0 |
| 0 |
| 1 |
| 1 |
| 1 |
| 0 |
| 0 |
| 2 |
| 2 |
| 2 |

| | | |
|---|---|---|
| 1 | 0 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 0 | 0 |
| 1 | 3 | 0 |
| 1 | 3 | 0 |
| 1 | 3 | 0 |

**Table V:** Finding closest ancestor

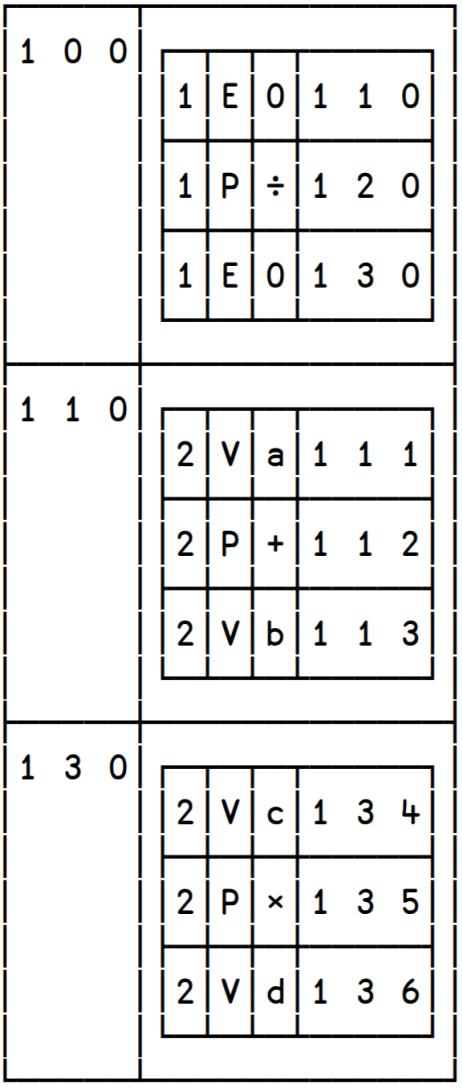


**Figure 2.2:** Node coordinates grouped by expression nodes

function f, find all equivalent rows in a, and However, Haskell Accelerate library does not provide a built-in key operator, so I will attempt to implement it in my project.

## 2.3 Haskell Array Computations

There are many Haskell libraries which implement efficient array data structures. The most popular choices are Accelerate [7] and REPA [8] libraries. Their API is similar, and they both present an embedded language (e.g. a language with identical or similar to Haskell syntax, but with custom compilers). REPA provides a compiler which targets CPU execution, while Accelerate provides compilers which target both CPU[1] and GPU[2] among others.

Since Accelerate offers more flexibility and more targets for compiling, I decided to choose it as the main Haskell library for the research.

## 2.4 Stack Package Tool

To simplify development and managing of multiple packages, I decided to use Stack [9] package tool. Among other features, it allows to create new project with boilerplate files included, automatically install GHC in an isolated location and build the project from specification file.

The main configuration file is package.yaml, which allows developers to specify project meta information, such as author's name or license type, as well as specify various parameters for different parts of the project, such as dependencies, GHC options, and others.

[1] http://hackage.haskell.org/package/accelerate-llvm-native
[2] accelerate-llvm-ptx

By default, Stack divides the codebase into three logic parts: library, executables, and tests. Library files are located by default in src folder and contain the main logic of the program, while executable files are found under App folder and serve as an entry point for the application. The executables could also contain simple command line or graphical interface logic. Test files are located under test folder; however, I did not implement any test suite for this project.

# Chapter 3

# Methodology

In this chapter I will describe the main architectural decisions, challenges, and technologies used for implementation of the system.

## 3.1 Technologies used

In the course of this study, I used several technologies to implement and operate on various data structures.

The main technologies behind the implementation are Haskell and Accelerate library. Their usage will be further described in details.

I also used Haskell project management tool Stack [9] to simplify package management and reuse best practices and structures to work with a Haskell project.

## 3.2 Haskell

Since the goal in this research is to replicate and benchmark the results achieved by A. Hsu [1] in Haskell, its usage is the main focus of implementation.

Similarly to some other programming languages like C++ or Python, Haskell is not a concrete implementation or system by itself. Instead, it is a language standard [10] that must be followed for any implementation. However, even though various implementations of Haskell [11] exist, the GHC compiler [12] is considered a de-facto standard state-of-the-art compiler traditionally associated with the Haskell language.

For this project, I used the latest release to date, ghc-8.10.3.

## 3.3 Stack

The main file of the implementation is src/AccTree.hs, which contains the main logic of the program. Since Accelerate redefines most standard functions and operators, which are included in Prelude and are implicitly imported in all files, Accelerate documentation [1] suggests to explicitly import Prelude as P with qualified keyword. This means that all Prelude functions, operators, and structures will not be added to the namespace, but will be accessible via P qualifier (e.g. P.map or P.==).

Another file, sec/Tree.hs, contains basic definition, as well as Functorinstance implementation and several utility functions for Treetype.

Additionally, a few utility functions are contained in src/Utils.hs file. I did not define them in src/AccTree.hs simply because qualified Prelude namespace makes the definition of the utility functions much more cumbersome and harder to read and understand.

Finally, files Demo.hs and Benchmarks.hs contain various functions for testing and benchmarking, which will be described in detail in the next chapter.

[1] https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html

## 3.4 Accelerate

Accelerate is an embedded array language for high performance in Haskell [7]. "Embedded" means that the language has separate standalone compilers. Indeed, the package provides several backends for compiling to target several platforms, in particular, CPU and GPU. The idea behind the language is simple. All data are represented as arrays of subtypes of Elt element and include signed and unsigned Int, Bool, Char, as well as tuples and shapes. The arrays have fixed size which is required to specify upon their creation. The size of an array, its shape, and an index to it are all specified using the so-called *snoc-list* that is just a sequence of elements starting with the zeroth element Z, and separated with :. operator (e.g. Z :. 1 :. 2 :. 3 ...). More interestingly, this structure serves both as a data structure to define array sizes and construct indexes, and as a type constructor to specify an array type. As such, array dimensionality is always specified by a type signatures, and is checked at compile time.

To perform any operation on an array, one must first make the data accessible to accelerate via function use. Depending upon which backend is used to eventually execute array computations, use may entail data transfer. This function also "wraps" an array in Acc type, which is required for all array manipulation functions of Accelerate.

As an *Embedded* language, an Accelerate program is *not* compiled by GHC. Instead, each Accelerate backend is a runtime compiler which generates and executes parallel code of the target language at application runtime using function run. So, each operation on an array is combined, optimized, and then executed at runtime.

## 3.5 Node coordinates

Node coordinates matrix is a central data structure in parallel compiler implementation. The matrix consists of multiple rows, where each row, node coordinate, is a vector of natural numbers with length equal to the depth of the tree. Every node in the tree is uniquely represented by a node coordinate vector, and each such vector is zero-padded on the right, with the number of non-zero elements equal to the depth of the tree.

Node coordinate matrix is lexicographically ordered and has dimensions $NxD$, where $N$ is the total number of nodes in the tree, and $D$ is the depth of the tree. In other words, each subsequent row in matrix differs from its predecessor by only one non-zero element.

One important property, which is not observed in recursive implementation, is that every node coordinate is a prefix of each of its descendant, ignoring zeros; and conversely, each child node contains a number of sub-sequences, each of which points to its ancestor. This means that it is possible to identify such node relations as siblings, ancestors, and descendants in constant time.

## 3.6 Design and Architecture

### 3.6.1 Node coordinates construction

As described above, several steps are needed to group the nodes using node coordinates.

To better illustrate those steps, consider a sample expression (fig. 3.1) with the AST (fig. 3.1), and observe how it is transformed step by step. For the purposes of this research, I simply treat AST as the input, already built

somewhere else. This treatment allows for flexibility to not assume a specific syntax, specific AST, or even specific purpose: in fact, this approach can be used for something other then compiler construction.

$$(x * 2 + 1)/(3 + y) \tag{3.1}$$

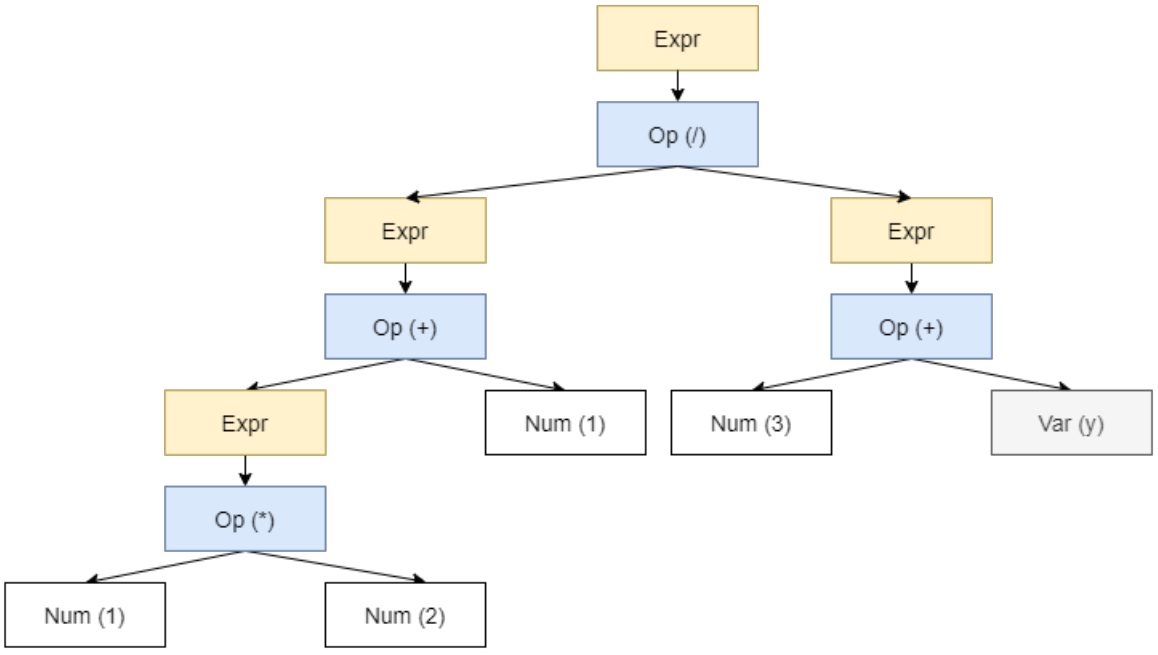


**Figure 3.1:** Sample AST

1. *Step 0:* the tree is converted to vector form, so that each node is represented by its depth, type, and value (fig. VI). The resulting type and value arrays are padded to have the same length, and the data is converted to accelerate arrays

2. *Step 1:* creating boolean vector matrix, where each for each depth matrix coordinate $i$ $element_{i,depth\_vector[i]} = 1$, and all other values are 0.

3. *Step 2:* accumulating values in columns of matrix from the previous step,

4. *Step 3:* removing extra numbers, so that count of non-zero elements in each node coordinate corresponds to the depth of that node. Fig. VII shows these steps.

### 3.6.2 Finding closest focus ancestors

Another important algorithm is used to find closest ancestors of certain type, focus nodes. Similarly to node construction, this algorithm also consists of several parts. However, these parts are independent and can be used separately for other purposes.

1. Find focus nodes given their type ('Expr' in the example). Result for the sample tree is illustrated in VIII.

2. Construct parent matrix. This matrix is used to ensure that a node is not marked as an ancestor to itself. As mentioned in 3.5, each node coordinate contains parent coordinates as prefixes. Therefore, to accomplish this step, simply set the last non-zero element in each row to zero. Parent matrix for sample AST is illustrated in IX

3. Construct matrix of closest focus ancestors, where each $row_i$ is a node coordinate of closest focus ancestor of $node_i$. This involves can be done in several steps, described below and illustrated in X:

    - Produce and inner product between parent coordinate matrix and focus node matrix with boolean function that tests whether a vector is a prefix to the other vector, ignoring zeros. In other word, each

| | | |
|---|---|---|
| 0 | Expr | |
| 1 | Op | / |
| 2 | Expr | |
| 3 | Op | + |
| 4 | Expr | |
| 5 | Op | * |
| 6 | Var | x |
| 6 | Num | 2 |
| 4 | Num | 1 |
| 2 | Expr | |
| 3 | Op | + |
| 4 | Num | 3 |
| 4 | Var | y |

**Table VI:** Depth vector representation

$element_{i,j}$ of such matrix will specify whether or not jth focus node is an ancestor of ith node.

- Construct an vector of indexes to focus nodes, which specifies closest ancestor of each node from parent matrix. To do that, one can simply replace non-zero elements in the matrix from previous step by their column index, and take maximum in each row.
- Finally, using the vector of indexes obtained in the previous step, construct a matrix where where each row specifies the closest ancestor from focus nodes for the corresponding node

### 3.6.3 Key operator

As mentioned in 2, key operator is crucial as the last step of Hsu's algorithm. Ideally, after applying key operator to closest focus ancestors and node coordinates matrix, one gets result similar to the one in XI. However, this data

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |

**(a)** Step 1

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 1 | 1 | 1 | 1 | 2 | 1 | 2 |
| 1 | 1 | 2 | 1 | 2 | 1 | 2 |
| 1 | 1 | 2 | 2 | 2 | 1 | 2 |
| 1 | 1 | 2 | 2 | 3 | 1 | 2 |
| 1 | 1 | 2 | 2 | 4 | 1 | 2 |

**(b)** Step 2

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 1 | 1 | 1 | 1 | 2 | 0 | 0 |
| 1 | 1 | 2 | 0 | 0 | 0 | 0 |
| 1 | 1 | 2 | 2 | 0 | 0 | 0 |
| 1 | 1 | 2 | 2 | 3 | 0 | 0 |
| 1 | 1 | 2 | 2 | 4 | 0 | 0 |

**(c)** Step 3

**Table VII:** Constructing node coordinates

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 2 | 0 | 0 | 0 | 0 |

**Table VIII:** Coordinates of nodes of type 'Expr'

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 2 | 0 | 0 | 0 | 0 |
| 1 | 1 | 2 | 2 | 0 | 0 | 0 |
| 1 | 1 | 2 | 2 | 0 | 0 | 0 |

**Table IX:** Parent coordinates matrix

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 |

**(a)** Inner product of parent matrix with focus nodes

| |
|---|
| 0 |
| 0 |
| 0 |
| 1 |
| 1 |
| 2 |
| 2 |
| 2 |
| 1 |
| 0 |
| 3 |
| 3 |
| 3 |

**(b)** Step 2: Indexes to focus node matrix

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 2 | 0 | 0 | 0 | 0 |
| 1 | 1 | 2 | 0 | 0 | 0 | 0 |
| 1 | 1 | 2 | 0 | 0 | 0 | 0 |

**(c)** Step 3: Matrix of closest focus ancestors

**Table X:** Constructing closest focus ancestors matrix

structure is impossible in accelerate: since each grouped segment has different length, and accelerate does not allow nested arrays. One solution to this problem is used throughout accelerate itself: simply return a tuple with segment descriptor[2] as a second element. Segment descriptor is simply a vector which describes length of each specific segment in the return array. So the result looks similar to XII.

[2]https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#t:Segments

| | | | | |
|---|---|---|---|---|
| 1 0 0 0 0 0 0 | 0 | Expr | | 1 0 0 0 0 0 0 |
| | 1 | Op | / | 1 1 0 0 0 0 0 |
| | 2 | Expr | | 1 1 1 0 0 0 0 |
| | 2 | Expr | | 1 1 2 0 0 0 0 |
| 1 1 1 0 0 0 0 | 3 | Op | + | 1 1 1 1 0 0 0 |
| | 4 | Expr | | 1 1 1 1 1 0 0 |
| | 4 | Num | 1 | 1 1 1 1 2 0 0 |
| 1 1 1 1 1 0 0 | 5 | Op | * | 1 1 1 1 1 1 0 |
| | 6 | Var | x | 1 1 1 1 1 1 1 |
| | 6 | Num | 2 | 1 1 1 1 1 1 2 |
| 1 2 0 0 0 0 0 | 3 | Op | + | 1 1 2 2 0 0 0 |
| | 4 | Num | 3 | 1 1 2 2 3 0 0 |
| | 4 | Var | y | 1 1 2 2 4 0 0 |

**Table XI:** Result of key operator - using nested arrays

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 2 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 2 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 1 | 1 | 2 | 2 | 0 | 0 | 0 |
| 1 | 1 | 2 | 2 | 3 | 0 | 0 |
| 1 | 1 | 2 | 2 | 4 | 0 | 0 |

| |
|---|
| 3 |
| 2 |
| 5 |
| 3 |

**Table XII:** Result of key operator - using segment descriptor

# Chapter 4

# Implementation

This chapter will further describe and dive into details of implementation.

## 4.1 Project configuration

As mentioned above, this project uses Stack [9] package manager. It allows to specify all project-related configuration details in package.yaml, and specify extra dependencies in stack.yaml.

Dependencies for this project include accelerate, along with accelerate-native for CPU targeting, and criterion [1] for benchmarking.

## 4.2 Types

In the previous chapter, it was established that the input to our system is an AST. Therefore, I implemented several simple data structures to represent them, along with Functor, Foldable, and Show instances to simplify various operations on trees. You can see the type specifications in fig. 4.1. Instance im-

[1] https://hackage.haskell.org/package/criterion

```
data Tree a = Tree
{ treeRoot     :: a
, treeChildren :: [Tree a]
}

data ASTNode = ASTNode
{ nodeType  :: String
, nodeValue :: String
}

type AST = Tree ASTNode
```

**Figure 4.1:** AST Data Structure

plementation is intentionally omitted, since it is mostly trivial and not essential to the solution.

## 4.3 General functions

Throughout the implementation several patterns and operations emerge. These operations are common enough to be abstracted to separate general functions. Moreover, implementing separate functions for common operations allows for easier optimization. That is, instead of optimizing the same operation throughout the codebase, which could possibly be implemented in a variety of ways, one simply needs to focus on one function to improve overall efficiency of a library.

This section lists and describes such functions in my codebase.

### 4.3.1 Inner product

First common function is inner product. It is similar to matrix product[2], but instead of multiplying rows by columns, it multiplies rows by rows. Also,

[2]https://en.wikipedia.org/wiki/Matrix_multiplication

```
innerProduct :: (Elt a, Elt b, Elt c)
    => (Exp a -> Exp b -> Exp c)
    -> (Exp c -> Exp c -> Exp c)
    -> Acc (Matrix a)
    -> Acc (Matrix b)
    -> Acc (Matrix c)
innerProduct prodF sumF a b = fold1 sumF $ zipWith prodF aExt bExt
    where
        (I2 r1 _) = shape a
        (I2 r2 _) = shape b

        aExt = replicate (lift (Z :. All :. r2 :. All)) a
        bExt = replicate (lift (Z :. r1 :. All :. All)) b
```

**Figure 4.2:** Inner product function

instead of multiplication for elements and summation for rows, the two functions can be given as parameters, first is a product function (how to combine 2 elements from 2 matrices), and second is sum function (how to combine the row of results into a single element). That is, given two matrices $A_{m,n} :: Matrixa$ and $B_{k,m} :: Matrixb$, and two combination functions $f :: a- > b- > c$, and $g :: c- > c- > c$, an element in resulting matrix is obtained by combining rows $A_i$ and $B_j$ with f, and folding the resulting vector with g. In practice, since Accelerate does not facilitate easy operations on segments, one can implement inner product in Accelerate by transforming matrices to the same dimensions by replication. See the implementation in fig. 4.2 for more details.

### 4.3.2 Select rows

Another useful general function is selectRows. The logic is very simple: given a matrix, and a vector of indices, return a new matrix with rows selected by the index vector. But since the index vector is constrained only by maximum number it can contain, number of matrix rows - 1, selectRows has multiple

```
selectRows :: (Elt a) => Acc (Vector Int) -> Acc (Matrix a) -> Acc (Matrix a)
selectRows rowIndex arr = zipWith (\i j -> arr ! I2 i j) rowIndexMat colIndexMat
  where
    nResultRows = size rowIndex
    (I2 _ nCols) = shape arr
    rowIndexMat = replicate (lift (Z :. All :. nCols)) rowIndex
    colIndexMat = replicate (lift (Z :. nResultRows :. All))
      $ enumFromN (I1 nCols) (0 :: Exp Int)
```

**Figure 4.3:** Select rows function

applications, such as selecting a subset of matrix rows, replicating rows into a bigger matrix with a given order, and simply rearranging the existing rows.

The implementation is quite simple: replicate index vector as a column, replicate column indices as a row, and zip the two matrices with function to index from the original array. See the implementation in fig. 4.3.

## 4.4 Node coordinates construction

### 4.4.1 Tree to array representation

To be able to work with this data in Accelerate, we first need to convert it to arrays. There are several steps involved. First, one has to transform tree to depth vector representation. This involves several steps. First, the tree needs to be transformed into depth vector representation. It can be accomplished via a simple recursive function. See Second step pertains to transforming types and values data. Since Accelerate does not support String type, the only solution is to use Char matrix for storing both types and values of nodes. But this requires all strings in a matrix to be of the same length, which is not guaranteed in case of types of values. Therefore, in order to enable this conversion, we first need to make all types and values to be of equal length by padding them with some

```
padRight :: Int -> a -> [a] -> [a]
padRight n e arr
  | length arr == n = arr
  | otherwise       = arr ++ replicate (n - length arr) e

padToEqualLength :: a -> [[a]] -> [[a]]
padToEqualLength e arr = map (padRight maxLength e) arr
  where
    maxLength = maximum $ map length arr
```

**Figure 4.4:** Padding to equal length code

character. I chose ‘
0’, since it is guaranteed to not be used inside the strings themselves. For this, I implemented simple functions in file Util.hs to avoid namespace conflicts. See fig. 4.4 for its implementation.

Function padRight pads an array with the given symbol to the specified length, while padToEqualLength takes an array of arrays (e.g. array of Strings, since String is equivalent to [Char]) and uses the first function to pad all arrays to equal length.

After the padding is done, convert depth, types, and values lists to arrays. It can be done using fromList function of Accelerate library. See the complete implementation of the described steps in fig. 4.5.

### 4.4.2 Depth vector to node coordinates

The next transformation is construction of node coordinates, which was explored in depths in section 3.6.1. Implementation simply aims to replicate the steps in accelerate language. See fig. 4.6 for the code. Several points need clarifications, though. Scan function in Accelerate[3] is similar to scan in

[3]https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html#g:35

```
treeToVectorTree :: AST -> VectorTree
treeToVectorTree tree = (depthAcc, typesAcc, valuesAcc)
  where
    treeToList currLevel (Tree root children)
      = (currLevel, nodeType root, nodeValue root)
      : P.concatMap (treeToList (currLevel + 1)) children

    listTree = treeToList 0 tree

    depthVector = P.map (\(d, _, _) -> d) listTree
    types  = padToEqualLength '\0' $ P.map (\(_, t, _) -> t) listTree
    values = padToEqualLength '\0' $ P.map (\(_, _, v) -> v) listTree

    maxLength arr = P.maximum (P.map P.length arr)

    depthAcc = fromList (Z :. P.length depthVector) depthVector
    typesAcc = fromList (Z :. P.length types :. maxLength types) (P.concat types)
    valuesAcc = fromList (Z :. P.length values :. maxLength values) (P.concat values)
```

**Figure 4.5:** Tree to vector conversion code

pure Haskell[4], but it runs along the innermost dimension of accelerate arrays. That is, if array is 2-dimensional, each row will be scanned separately. Function scanl1 is a variant of scan without using an initial value. Function the 'unpacks' a value of single-element array, returned by maximum function in this case. Function generate is important as one of the ways to create an accelerate array. The function allows to construct an array by specifying the shape and index-to-value mapping for the new array.

The final transformation step is to simply transfer node coordinates, types, and values into an Accelerate value. To convert array to accelerate, one can simply apply function use to the array. Depending on chosen backend, this may entail transfer to a CPU or GPU.

[4]https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-List.html#g:6

```
constructNodeCoordinates :: Acc (Vector Int) -> Acc (Matrix Int)
constructNodeCoordinates depthVec = nodeCoordinates
  where
    maxDepth = the $ maximum depthVec
    nodeCount = size depthVec

    depthMatrix
      = generate
      (I2 nodeCount (maxDepth + 1))
      (\(I2 i j) -> boolToInt (depthVec !! i == j))

    cumulativeMatrix = transpose $ scanl1 (+) $ transpose depthMatrix

    dropExtraNumbers (I2 i j) e = boolToInt (j <= depthVec !! i) * e
    nodeCoordinates = imap dropExtraNumbers cumulativeMatrix
```

**Figure 4.6:** Node coordinates construction code

## 4.5 Towards closest focus ancestors

Finally, with tree represented as Accelerate data, one could apply various transformations to it. If the goal is grouping expressions, the next steps are the following:

- Find focus nodes, that is, nodes of some particular type useful for compiler optimization, such as Expr
- Construct parent matrix - a matrix where each row is a parent of the corresponding node
- Compute inner product using matrices from previous steps; find largest index of non-zero element in each row, and use the resulting vector to select rows from focus nodes matrix

```
findNodesOfType :: [Char] -> NCTree -> Acc (Matrix Int)
findNodesOfType query (T3 nc types _)
  = selectRows selector nc
  where
    queryAcc = use (fromList (Z :. P.length query) query)

    -- either index is out of bounds, and the element is zero, or elements match
    isValidCharacter (I2 _ j) c =
      (j >= size queryAcc && c == constant '\0') || c == queryAcc ! I1 j
    selector = afst
          $ filter (>= 0)
          $ imap (\(I1 i) v -> boolToInt v * (i + 1) - 1)
          $ and (imap isValidCharacter types)
```

**Figure 4.7:** Find nodes of specified type

### 4.5.1 Find nodes of type

This operation is quite easy: simply compare each row element-by-element with the query, summarise each row with and operation, obtaining boolean vector that specifies which rows match the query, and which do not. From there, replace each non-zero element with column index, filter for non-zero elements, and finally, use the resulting vector to select node coordinates. See the implementation in fig. 4.7.

### 4.5.2 Parent matrix

The second input to generating closest focus ancestor matrix is parent matrix: matrix of parents of each corresponding node coordinate. Due to the properties of node coordinates, one can simply replace with zero the last non-zero element to obtain the parent matrix. And this is exactly what getParenCoords function accomplishes. See the implementation in fig. 4.8.

```
getParentCoordinates :: Acc (Matrix Int) -> Acc (Matrix Int)
getParentCoordinates nc = imap (\i e -> boolToInt (not (isReplacedWithZero i)) * e) nc
  where
    (I2 _ nCols) = shape nc
    isReplacedWithZero (I2 i j) = j + 1 == nCols || nc ! I2 i j == 0
```

**Figure 4.8:** Get parent coordinates function

```
findAncestorsOfType :: [Char] -> NCTree -> Acc (Array DIM2 Int)
findAncestorsOfType query tree@(T3 nc _ _)
  = selectRows closestAncestorIndexVec focusNodes
  where
    focusNodes = findNodesOfType query tree
    parentCoords = getParentCoordinates nc

    closestAncestorIndexVec = fold1 max $ imap
      (\(I2 i j) e -> boolToInt e * j)
      (innerProduct (\a b -> a == b || b == 0) (&&) parentCoords focusNodes)
```

**Figure 4.9:** Function for finding closest ancestors of specified type

### 4.5.3 Closest focus ancestors

Finally, after finding focus nodes and obtaining parent matrix, it is possible to compute matrix of closest focus ancestors. To accomplish this, it is required to compute inner product of parent coordinates matrix with focus nodes matrix, using (==) operator as product function for combining two elements of different matrices, and (&&) as sum function to fold a row of results into a single value. As a result, a boolean vector matrix is produced, where each $element_{i,j}$ specifies whether or not row $i$ of parent matrix is a prefix of node coordinate of focus node $j$. Replacing each *true* element with column index, each *false* with 0, and selecting maximum in each row, one obtains a vector of indexes of focus nodes, which are closest focus ancestors of the corresponding nodes. Using selectRows, one finally obtains a matrix of closest focus nodes for each node. See the implementation in fig. 4.9.

## 4.6 Key

Key is an important part of the algorithm of A. Hsu. He relies on a native APL implementation of the key operator of an easy and efficient grouping. Unfortunately, Accelerate library does not provide such an operator, so it was my task to implement it. However, due to the complexity of Accelerate and limited time, I was only able to only partially implement a 2-dimensional version of the key operator.

While the key operator by definition groups the values matrix by the equal rows in key matrix, applying the given function each row of keys and values to get the resulting row, my implementation currently only functions as a grouping function. In other words, while the operator accepts a row transformation function, it currently remains unused.

It was the hardest hard to implement, since accelerate does not provide any tools for operations on rows, only either on the values themselves, or on the array in general. Of course, there are functions like scan and fold, but those operate on elements rather than on rows.

On the other hand, grouping part works and produces correct results. Once again, it involves both innerProduct and selectRows functions.

To implement grouping in key, one might begin by finding equal rows in keys. This can be done simply by computing inner product of key with itself, using (==)) as product function, and (&&) as sum function. As result, a matrix of groups is generated where each $element_{i,j}$ is true if the row $i$ is equal to the current row. In other words, the rows in the matrix of groups show which other rows are equal to the current one.

Next, one can replace the true elements with column number, and fold the matrix by selecting minimum non-zero number. Finally, a vector is generated

which contains possibly repeating indexes of unique rows. But it can easily be fixed to contain unique numbers. Since each repeating row in the matrix of groups is equal, and the first element in the first encounter of a unique row is the index of the row itself, the resulting vector, where we select minimum non-zero index of row, has an interesting property: the first encounter of a number n is necessarily at index n. That is, to obtain a vector of indices of unique rows in matrix of groups, one simply needs to fold each row to the minimum non-zero value, replace in the resulting vector each value that is different from its index by -1, and filter the result with $(>= 0)$ function.

Using the obtained vector, one can select unique rows from groups matrix, and filter non-zero elements. As a result, a vector is generated which contains the arrangement of rows in vectors array. Moreover, as a second argument of tuple, filter returns segment descriptor, which in this case describes a number of value rows in each group. Finally, one can simply select rows using the earlier obtained vector.

See the implementation in fig. 4.10.

Even though I was not able to implement n-dimensional key operator, I have designed the type signature for it. See the type signature in fig. 4.11.

```
key2 :: (Elt k, Elt v, Elt r, Eq k)
    => (Acc (Vector k) -> Acc (Vector v) -> Acc (Vector r))
    -> Acc (Matrix k)
    -> Acc (Matrix v)
    -> Acc (Matrix v, Segments Int)
key2 _ keys vals = T2 (selectRows selectors' vals) descriptor
    where
        (I2 nKeysRows nKeysCols) = shape keys

        -- each row contains indexes of equal rows
        -- groupsMatrix: nKeysRows x nKeysRows
        groupsMatrix = imap (\(I2 _ j) v -> boolToInt v * (j + 1) - 1)
            $ innerProduct (==) (&&) keys keys

        -- if a or b is zero, min a b is zero, and if not, (a == 0 || b == 0) is zero
        chooseMinId a b = if a < 0 || b < 0 then max a b else min a b
        uniqueRowsIdxVec = afst
                    $ filter (>= 0)
                    $ imap (\(I1 i) n -> boolToInt (i == n) * (n + 1) - 1)
                    $ fold1 chooseMinId groupsMatrix

        (T2 selectors' descriptor) = filter (> 0) $ selectRows uniqueRowsIdxVec groupsMatrix
```

**Figure 4.10:** 2-dimensional key operator

```
key :: (Shape sh, Shape sh', Elt k, Elt v, Elt r)
    => (Acc (Array sh k) -> Acc (Array sh' v) -> Acc (Array sh' r))
    -> Acc (Array (sh :. Int) k)
    -> Acc (Array (sh' :. Int) v)
    -> Acc (Array (sh' :. Int) r, Vector Int)
key f keys vals = undefined
```

**Figure 4.11:** N-dimensional key operator signature

# Chapter 5

# Evaluation and Discussion

## 5.1 Measuring Performance

To measure the performance of the system, I used a popular Haskell library for benchmarks Criterion[1]. It allows a simple but powerful way to measure efficiency of code.

To compare an array-based implementation, I implemented Functorand Foldableinstances of a tree using recursion, which allowed to easily manipulate a tree and create recursive analogs of accelerate-based operations. The implementation is mentioned in chapter 4.

Using instances, I created a simple recursive algorithm for finding closest focus ancestor. See the implementation in fig. 5.1

Finally, I needed a large enough data sample to avoid statistical errors. Therefore, I implemented a simple function which, given a number of levels, creates a simple tree of sum expressions. See the implementation in fig. 5.2.

[1] https://hackage.haskell.org/package/criterion

```haskell
findAncestorsOfTypeRec :: AST -> [Char] -> [(ASTNode, ASTNode)]
findAncestorsOfTypeRec tree targetType =
    let root = treeRoot tree
    in findClosestAncestors tree root [(root, root)]
    where
        findClosestAncestors (Tree root []) _ res = res
        findClosestAncestors node@(Tree root children) currClosest res
            = mconcat [
                newRes,
                mconcat $ map (\node -> findClosestAncestors node newClosest []) children
            ]
            where
                newClosest = if nodeType root == targetType then root else currClosest
                newRes = res ++ zip (map treeRoot children) (repeat newClosest)
```

**Figure 5.1:** Recursive algorithm for finding closest ancestors

```haskell
buildNLevelAST :: Int -> Int -> AST
buildNLevelAST width n
    | n <= 0 = astLeafNode "Num" "42"
    | otherwise =
        Tree (ASTNode "Expr" "")
        [
            Tree (ASTNode "Op" "+")
            (replicate width (buildNLevelAST width (n - 2)))
        ]
```

**Figure 5.2:** Simple tree generation

## 5.2 Comparison

### 5.2.1 Interface

One of the goals of this research was to create a purely-functional Haskell interface to array-based computations. The implementation proposed by A. Hsu [1] is efficient and important to the development of compilers, but it has a problem: the programming language that the author uses, APL[2], while efficient and concise, is not general purpose, and hard to understand for those who have not worked with it. The language uses a large variety of characters to represent operations. This might impose a negative effect on the true appreciation of his findings.

Therefore, in my code I tried to amend this problem by providing a solution which uses a more readable and general purpose programming language, Haskell, as well as providing a purely-functional interface to all operations. As a result, one needs to simply define or provide an AST to use any of the defined functions as easily as writing a pure Haskell code.

### 5.2.2 Performance

As mentioned above, I have created benchmarks for testing accelerate implementation against a recursive one. Unfortunately, the accelerate implementation is much less efficient than the recursive one. Benchmarks show the recursive implementation to be a factor of 1000 times more efficient (16 ms. against 14 ns.). But this does not mean that the approach itself is wrong. There are multiple reasons for the lack of efficiency, such as

[2]https://www.dyalog.com/what-is-dyalog.htm

- Lack of time and knowledge on my part led to inefficient implementation of various functions
- Accelerate documentation does not mention efficiency of the functions, so there may be some functions which are more efficient, and some functions which should not be used, or used as less as possible
- I benchmark finding closest focus ancestors, and not the key operator, which might prove to be more efficient in accelerate than in recursive implementation

## 5.3 Future work

As mentioned above, the current results do not necessarily signify the unfitness of this approach for Accelerate or Haskell. On the contrary, it opens up a big room for optimizations and improvements to the current code.

For example, one might want to measure each accelerate function, and make a conclusion about their efficiency. Another option is to implement most crucial functions, such as selectRowor innerProductnatively in C++ or even in LLVM instruction code.

Another possible research direction is to generalize current functions and implement additional ones to provide, ideally, a library of parallel computations in pure Haskell.

Overall, if current toolset is both optimised and expanded with new functions, it could become an important step towards library for parallel compiler in pure Haskell.

# Chapter 6

# Conclusion

In the course of my research I have successfully implemented the novel approach to parallel compilation in Haskell. This approach, originally suggested by A. Hsu [1], focuses on eliminating recursion and branching in the compilation process to allow for concurrent execution of various compilation steps on multi-core CPU or GPU.

In his research, Hsu proposes an array-based approach for an AST and implements it in APL, Array Programming Language. He describes a series of steps which allow group the nodes in the tree by the closest ancestor of a chosen type, e.g. Expr. Selecting different types allows to accomplish several crucial compilation steps, namely expression flattening, function lifting, and constant folding.

Unfortunately, Hsu's findings might not be appreciated enough, since he uses an unpopular and hard to understand language, APL. Therefore, in my implementation I focused not only on replicating his result, but also on providing a purely functional, declarative-style interface for various general and specific operations. For example, I implemented such functions as inner_product,

select_rows, and grouping part of 2-dimensional key operator. Therefore, while the implementation is oriented towards parallel compilers, it could be used for other applications.

Unfortunately, the implementation turned out to be inefficient compared to recursive approach. But this does not mean the general ineffectiveness of this approach, and should not discourage further research on this topic.

There are multiple areas of improvement for future researches. Firstly, one could optimize the existing functions by using Accelerate more efficiently, or even implementing some crucial functions as native ones. Another research goal is to add other general functions to the system, with the end goal of creating a library of with various array operators and functions, found in APL or similar array programming languages, with purely functional, easy-to-use Haskell interface.

# Bibliography cited


[1] A. W. Hsu, "The key to a data parallel compiler," ser. ARRAY 2016, Association for Computing Machinery, 2016, pp. 32–40. DOI: 10.1145/2935323.2935331. [Online]. Available: https://doi.org/10.1145/2935323.2935331.

[2] E. W. Weisstein. (2021). "Tree," [Online]. Available: https://mathworld.wolfram.com/Tree.html (visited on 04/23/2021).

[3] HaskellWiki. (2016). "Applications and libraries/compiler tools," [Online]. Available: https://wiki.haskell.org/index.php?title=Applications_and_libraries/Compiler_tools&oldid=60537 (visited on 06/19/2021).

[4] J. Grattage. (2008). "An overview of qml with a concrete implementation in haskell." eprint: 0806.2735, [Online]. Available: https://arxiv.org/abs/0806.2735 (visited on 06/19/2021).

[5] R. Hui. (Apr. 2018). "Essays/key," [Online]. Available: https://code.jsoftware.com/wiki/Essays/Key (visited on 03/13/2021).

[6] E. Iverson. (2001). "J primer," [Online]. Available: https://jsoftware.com/help/primer/contents.htm (visited on 03/13/2021).

[7] T. McDonell, M. Chakravarty, V. Grover, and R. Newton, "Type-safe runtime code generation: Accelerate to llvm," *ACM SIGPLAN Notices*, vol. 50, Aug. 2015. DOI: 10.1145/2887747.2804313.

[8] HaskellWiki. (2016). "Numeric haskell: A repa tutorial," [Online]. Available: https://wiki.haskell.org/index.php?title=Numeric_Haskell:_A_Repa_Tutorial&oldid=60893 (visited on 03/13/2021).

[9] (2021). "The haskell tool stack," [Online]. Available: https://docs.haskellstack.org/en/stable/README/#the-haskell-tool-stack (visited on 03/13/2021).

[10] (2010). "Haskell 2010 language report," [Online]. Available: https://www.haskell.org/onlinereport/haskell2010/ (visited on 03/13/2021).

[11] (2020). "Implementations — haskell wiki," [Online]. Available: https://wiki.haskell.org/Implementations (visited on 03/13/2021).

[12] (2020). "Ghc — haskell wiki," [Online]. Available: https://wiki.haskell.org/index.php?title=GHC&oldid=63344 (visited on 03/13/2021).