Workflow Management System for Stratosphere

by

Suryamita Harindrari

Submitted to the Department of Computer Science and Electrical Engineering
on August 10, 2014, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

In this thesis, we design and partly develop a Workflow Management System (WMS) aimed to work on top of Stratosphere, a Big Data platform developed by TU Berlin. The WMS is defined by means of a Domain Specific Language (DSL) written in Scala. Control Flow and data dependencies are automatically detected by static analysis on the Scala code through the following three stages: (1) create a Control Flow Graph as an intermediate representation from Scala AST, (2) detect data dependencies in the graph, and (3) generate code for the underlying system. We cover the implementation of the first stage and provide the algorithm for the subsequent two stages. In the evaluation, we argue over the advantages of this DSL compared to related WMS work in terms of user-friendliness and independence of underlying platform.

Thesis Advisor: Asterios Katsifodimos
Thesis Supervisor: Volker Markl

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advancement of Big Data Analytics, numerous systems for cluster computing on big data have been developed [37, 6, 29, 10, 36, 26] and data engineers are building more and more complex applications to manage and process large data sets on distributed resources. Such complex application scenarios require means in order to compose and execute complex workflows. Workflows automate procedures that users would otherwise need to carry out manually [12]. It refers to a sequence of steps or computations that a user would like to perform [1]. As an example, within a Hadoop [2] cluster, a user may need to export the production databases and load the data to the Hadoop File System (HDFS) as the first step. The second step would be to run a MapReduce job to clean up the data and step three would be a set of operations that run in parallel to count and filter the data. A workflow is intended to map all of the different operations together. Such a workflow is usually represented as a Directed Acyclic Graph (DAG) where the nodes can be tasks or control flow structures and edges represent the relationships between tasks, namely task or data dependency. A Workflow Management System (WMS) is a system that allows users and developers to create, define, run, and delete a workflow [1].

As an introduction to what a workflow may look like, we will walk through two different use cases that are representative use cases in the Big Data environment.

[1]http://www.crobak.org/2012/07/workflow-engines-for-hadoop
[2]http://hadoop.apache.org/.

The first use case is Analytics/Data Warehousing. A workflow in this first use case consists of the following steps: (1) load the logs into the Fact tables, (2) load the database backups into the Dimension tables, (3) compute the aggregations and perform rollups/cubes inside Hadoop for instance, (4) load the data into a low latency store, and (4) in the end, perform the analytics using a Dashboard and BI tools. The second use case is related to machine learning or collaborative filtering. A workflow in this use case consists of the followings steps: (1) load the logs and database backups into the HDFS, (2) perform the collaborative filtering and machine learning computation, (3) produce the production datasets in Hadoop, for example, (4) perform the sanity check of the production data set, and (5) at the end, load the cleaned data to production data store [1].

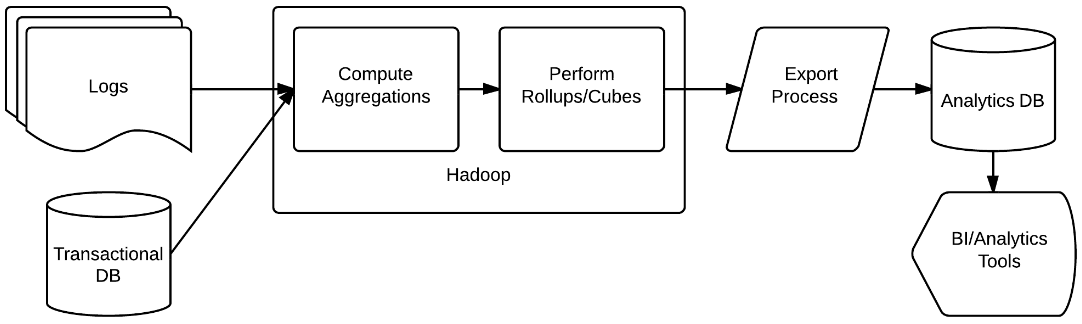

Figure 1-1: Workflow for Analytics/Data Warehousing [1]

1.1 Motivation

The current existing WMS, which will be explored further in Section, mainly act as a "glue" of simple jobs defined by the developer. Data dependencies and control flow in the workflow (e.g. decision making, looping, and branching) are specified manually in the model. This manual job causes a large overhead and confusion to the programmer. Thus, it would be convenient if there exist a design of a WMS that is able to automatically detect the control flow and data dependencies between the tasks based on pure program code that is written in a specific language familiar to the programmer.

The overall goal of this thesis is to design and partly develop a WMS prototype that works on top of Stratosphere [2], an emerging large-scale data processing framework developed TU Berlin, Humboldt Universität zu Berlin, and the Hasso-Plattner-Institut funded by the Deutsche Forschungsgemeinschaft (DFG). The approach that will be taken to define the workflow specification is to develop a Domain Specific Language (DSL) [31] on top of Scala [24] high-level functional programming language. The idea is to build a WMS that will take as an input a program, in which the programmer defines a set of tasks associated with each other in a given sequence in a high-level language that is fairly similar to Scala functional programming language, only with a set of restrictions. The WMS and then execute the tasks. Control flow, and data dependencies will be automatically detected by static analysis on the program code. The WMS will subsequently execute the tasks in a sequence that is according to the control flow and data dependencies. Language integration has been an old goal in the database community. We would like to query, manipulate, store and process data in the same language. That said, the tasks that are triggered by the workflow can be written in any other language e.g. Java.

The first step to building this ideal WMS is to develop the programming model for the workflow DSL. In principle, a workflow language is required to have a model to maintain the tasks and the relationships between the tasks, both control flow and data dependencies. We will define the grammar of DSL to guide the programmer to write a workflow specification in our DSL. This workflow specification will be taken by the WMS as the input program. After the workflow specification is defined in the input program, the process of compiling the program into the target code to be run in the Stratosphere is divided into three stages as follows: (1) conversion from input program to an intermediate representation in a form of a control flow graph whereby the nodes of the graph represent tasks that can be run at once, (2) enrich the control flow graph with data dependencies between the nodes of the graph by performing data flow analysis, and (3) conversion of the intermediate representation which is the control flow enriched dataflow to the script of the jobs [3]. We will define a language grammar for this Scala DSL. This grammar defines the scope of Scala grammar [25] that can be understood, analyzed and later processed by our language to generate the intermediate representation and final job scripts to be run in the WMS. With regards to these stages of development, our contribution in this thesis is summarized as follow:

- Define the DSL grammar and programming model for our workflow,
- Analyze the user program to produce an intermediate representation in the form of a control flow graph, and
- With regards to the third and fourth stage of the WMS development, we present an algorithm to detect data dependencies between each node of the graph as well as an algorithm to generate the job scripts for the target machine.

The most important aim of this process, as mentioned in the beginning, is to avoid the manual job of defining dependencies, both tasks and data, when building the workflow. In a Oozie [4] workflow, an example of workflow systems for Hadoop, nodes in the DAG are forward-chain, that is, the developer needs to specify where a node or a computation in the DAG goes after it is finished. This can be hard to track and it requires the developer to remember every node in the chain when developing the workflow [3]. Thus, the Scala DSL that we aim to develop will attempt to focus around dependencies. The developer needs to look at one node in a workflow at a time, but does not need to define the tasks that that node depends on, the dependencies will be discovered by Scala code analysis. The approach that we will be taking to perform the code analysis and code generation would be to identify the self-contained jobs within branches of the Abstract Syntax Tree generated by the Scala compiler.

In the evaluation section, we argue over the advantages of this DSL compared to related WMS work in terms of user-friendliness and independence of underlying platform by selecting a use case that is representative of use cases running on Stratosphere.

[3]https://github.com/klout/scoozie
[4]http://oozie.apache.org

We show that even though at the moment this DSL can only run on Stratosphere, it can be extended to be used on another system.

Definition 1.1.1. *A domain-specific language (DSL) is a program- ming language or executable specification language that offers, through appropriate notations and ab- stractions, expressive power focused on, and usu- ally restricted to, a particular problem domain [31]*

1.2 Thesis Outline

Chapter 2 contains the background and related work on workflow systems and dataflow systems. When presenting the literature survey, we also introduce the approach of our workflow model.
Chapter 3 presents the implementation part of our DSL and WMS. We define the grammar of our workflow language and talks about the process of transforming the user program to an an intermediate representation which is an control flow enriched dataflow.
Chapter 4 discusses about the advantages of using our workflow language in terms of ease of use and extensible to any platform.
Chapter 5 is the conclusion in which we summarize our contribution and present the limitations and potential future work.

Chapter 2

Background, Approach, and Related Work

The study and development of workflows and workflow management systems have been conducted for years and in various field (i.e. e-Science, Grid computing, and recently Big Data Analytics). In this chapter we present the background on workflow systems, workflow language, workflow model, as well as introducing the design and programming model approach for our workflow language. Furthermore, we also present some of the related work on workflow and dataflow systems especially those developed to run on top of large-scala data processing platform.

2.1 Workflow

Workflow systems [35, 12, 32, 5] provide a way for programmers to arrange the tasks to be executed in a variety of different ways (A workflow is defined formally in Definition 2.1.1). A workflow is especially important for applications in which data dependencies exist between the tasks, and a flexible mechanism of arranging the tasks is necessary so that data produced by some tasks can be consumed by others [20]. In order to execute such a workflow, we require a Workflow Management System (WMS) [32] (refer to Definition 2.1.2 for a formal definition of WMS). Workflow languages refer to a form of high-level programming language in which operations correspond to tasks that are executed by external pieces of software, usually remotely [20]. Commonly used design patterns in workflows have been surveyed by van der Aalst et. al. [30], Bharathi et. al. [7], Pautasso and Alonso [27], and Yildiz, Guabtni, and Ngu [33].

Definition 2.1.1. *"Workflow is the computerised facilitation or automation of a business process, in whole or part [16]."*

Definition 2.1.2. *"Workflow Management System (WMS) is a system that completely defines, manages and executes work- flows through the execution whose order of execution is driven by a computer representation of the workflow logic [16]."*

2.1.1 Workflow Design

[35] classifies the workflow design to at least three taxonomies, namely: (a) workflow structure, (b) workflow model/specification, and (c) workflow composition system. In the view of workflow structure, a workflow is composed by connecting multiple tasks according to their dependencies. In general, the workflow can be represented as a DAG or non-DAG. The workflow that we develop in this thesis belongs to non-DAG-based workflow. In a non-DAG workflow, workflow structure is categorized into sequence, parallelism, choice, and iteration. Sequence is defined as an ordered series of tasks, with one task starting after a previous task has completed. Parallelism represents tasks which are performed concurrently, rather than serially. In choice control pattern, a task is selected to execute at run-time when its associated conditions are true. In Iteration structure, also known as loop, sections of workflow tasks in an iteration block are allowed to be repeated and is often occurred in workflow of complex use cases [35].

Workflow Model, which is also called workflow specification, defines a workflow including its task definition and structure definition. There are two types of workflow models, namely abstract model and concrete model, denoted as abstract workflow and concrete workflow, respectively [11]. In the abstract model, a workflow is described in an abstract form, in which the workflow is specified without referring to specific resources for task execution. The abstract model enables users to define workflows without being concerned about low-level implementation details. In contrast, the concrete model binds workflow tasks to specific resources [35]. In this thesis, we implement the abstract workflow which takes the form of control flow enriched dataflow. Furthermore, we also provide the algorithm to achieve the concrete workflow, namely the job scripts to be executed in the underlying system.

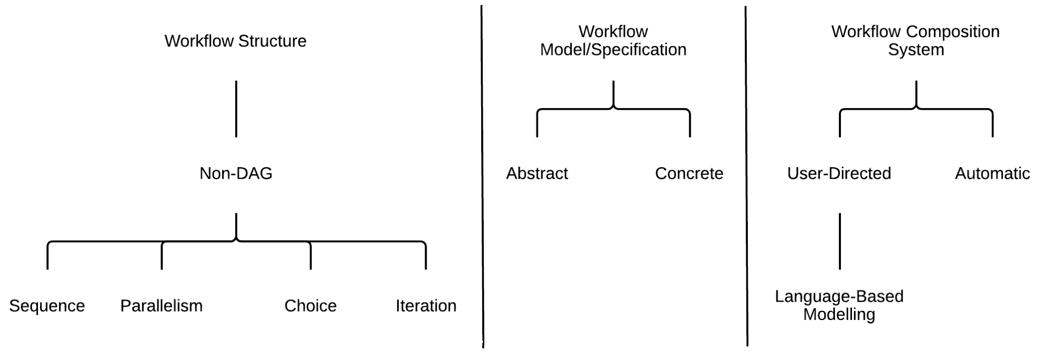

Figure 2-1: Thesis Workflow Design Approach

Workflow composition systems are designed for enabling users to assemble components into workflows [35]. It consists of two classes: (1) User-directed and (2) automatic. User-directed composition systems allow users to edit workflows directly, whereas automatic composition systems generate workflows for users automatically [35]. Both classes are implemented in this thesis. More specifically, in the user-directed, language-based modeling is applied since we require user to write their program in our Scala DSL. However, the control flow and data dependencies will later be identified automatically by performing the static analysis of the Scala program. In this case, user does not have to specify manually the workflow components and dependencies. To summarize, the taxonomy of workflow that we develop in this thesis is depicted in Figure 2-1.

2.1.2 Workflow Language

[20] well summarizes the common concepts of workflow language. Some of those concepts compose the programming model of our workflow language. We define these

concepts and map them in high-level to our programming model as follows:

- **Tasks** refer to units of work that are used within a workflow. Each task takes as input a set of values, and produces another set of values as output when executed. A task can thus be considered equivalent to a function. Each task in our abstract workflow corresponds to a node in our control flow enriched data flow graph that we will explain in detail in Chapter 3. Whereas in our concrete workflow, a task correspond to one job that will be executed in the underlying system.

- **Data dependencies** are directional relationships between tasks, each indicating that the input of one task is an output of another. These data dependencies determine which data to supply as input when executing a task as well as the relative order of the execution of the tasks. In our abstract workflow, these data dependencies correspond to directional data-flow edges in the graph.

- **Control dependencies** indicate a constraint on the relative order of execution of two tasks, without implying data transfer. They only occur in tasks with special effects, where data dependencies alone are not sufficient to determine the ordering.

- **Parallelism** is enabled by establishing the dependencies in the structure of the workflow. Any set of tasks that are not dependent on each other can be run in parallel. The WMS uses the control and data dependencies to automatically determine which set of tasks to run in parallel.

- **Conditional branching and iterations** in a workflow language is similar to branching and iterations in a conventional programming language. Based on a certain condition, a conditional branch selects which part of the workflow will be subsequently executed. The condition itself is performed by executing a task with some input data and returns a boolean value. Iteration enables part of the workflow to be run multiple times, such that in each iteration, tasks withing the loop body are executed with different input value, given that the condition of the iteration is satisfied. Our workflow language support both conditional branching and iterations.

- **Representation** of a workflow normally consists of a graph, whereby nodes corresponds to tasks and edges correspond to dependencies. Parallel execution uses the dependency information to determine which tasks can be run in parallel. In chapter 3, we will see in detail the representation of our workflow language.

2.2 Control Flow vs Data Flow

In addition to the three taxonomies described in the previous section, most, if not all, workflow design belongs to one of two classes: control flow or data flow. The two classes are similar in that they specify the interaction between individual tasks within the group that comprise the workflow. The difference between the two is in their methods of implementing that interaction. In control-driven workflows, or control flows, the connections between the activities in a workflow represent a transfer of control from the preceding task to successor task. This includes control structures such as sequences, conditionals, and iterations. Data-driven workflows, or data flows, on the other hand, are designed to support data-driven applications. The dependencies represent the flow of data between workflow activities from data producer to data consumer [12]. More details explanation on control flow and data flows are presented in the following sections.

2.2.1 Control Flow Model

Most control flow languages provide support not only for simple flows of control between components or services in the workflow but also for more complex control interactions such as iterations and conditionals branching. Users of workflow systems often require more than the simple control constructs that are available to them. The ability to perform branching in the workflow based on conditions and loop over sections of the workflow repeatedly is important for all applications especially for complex use cases [12]. The important issue here is how to represent these conditionals and iterations in the workflow language and to what degree the language should support them. For instance, there is a question to whether a single simple loop construct is enough, or whether the language should support all loop types(i.e. while, do while, for). In the case of conditional behavior, the problem is determining whether the incoming value and the conditional value are equivalent [12].

The term control flow refer to the parts of a program that determine what computation is to be performed, based on various conditions [20]. Examples of control flow constructs include conditional statements, iteration, function calls, and recursion. Control flow does not necessarily dictate the exact order in which execution will occur, but rather determines which expressions will be evaluated in order to compute the final result of a program. Most workflow languages provide limited forms of control flow, such as conditional statements and sequential iteration.

Standard control flow constructs such as conditionals branching and iterations are supported by most large-scale data processing systems such as Stratosphere [3] and Spark [38]. In Stratosphere and Spark API, for example, expressions can include standard arithmetic, relational, and boolean operators, as well as their primitive operators such as map, reduce, filter, etc. The result of an expression may be used in any place that a regular value may be, such as input to a conditional test for branching or iteration. Below is the example of expression in a condition test that can be understood by Stratosphere and Spark API for branching and iteration, respectively.

```
if (A.reduce(_ + _)
while (B.reduce(_ + _) < 10000)
```

2.2.2 Dataflow Model

In a dataflow model, a workflow is represented as a task dependency graph, in which nodes correspond to tasks, and edges indicate data dependencies. The graph specifies a relative order of execution, by constraining each task to begin only after those it depends on have completed assuming that each task is free of side effects, and that its output depends only on its inputs [20]. Most data flow representations are very simple in nature, and unlike their control flow counterparts, contain nothing apart from component or service descriptions and the data dependencies between them; control constructs such as loops and branching are generally not included. The dependencies between tasks are data dependencies, ensuring the data producer has finished before the consumer may start [12]. The key advantage is that, in dataflow, more than one set of tasks can be executed at once. Tasks with no dependencies between them may thus be safely executed in parallel. This simple principle provides the potential for massive parallel execution at the tasks level [19].

A dataflow in Stratosphere is a directed acyclic graph (DAG) that consists of operators, data sources, and sinks or output. The data sources and sinks, as well as the intermediate data sets that flow through operators, are bags of records. A dataflow construct of several operators is therefore a function, whose fixpoint we can find by terminating the loop in the dataflow DAG [14]. Spark has a feature called Resilient Distributed Datasets (RDDs), a distributed memory abstraction that enables programmers perform in-memory computations on large clusters in a fault-tolerant manner[37].

2.3 Related Work

There are some major existing data flow systems and workflow systems that are interesting to mention. This section presents these different dataflow and workflow systems and discuss their characteristics, advantages, and limitations.

2.3.1 Dataflow Systems

Pig is a procedural dataflow system for MapReduce. It offers a SQL-style high-level data manipulation constructs, which can be assembled in any explicit dataflow and combine with custom MapReduce style functions. Pig programs are compiled into sequences of MapReduce jobs and run on the Hadoop MapReduce environment [15]. Pig write jobs for the Hadoop platform using a DSL called Pig Latin [26]. It is a dataflow language that provides extensive support for relational operations. Pig

Latin users need to learn a new language which is not the case with frameworks such as Hadoop. It does not include user defined functions, the user needs to define them externally in another language, which will often prevent optimizations as Pig can not analyze those [1].

A dataflow system developed at Microsoft is Naiad [23] which offers a new computational model called timely dataflow. This timely dataflow model enriches dataflow computation with timestamps that represent logical points in the computation and provide the basis for an efficient coordination mechanism. The model is based on a directed graph in which the nodes send and receive logically timestamped messages along directed edges [23].

Many existing systems for cluster computing have been developed in the past decade namely Hadoop MapReduce, Dryad [17], Pig [26], Hive [29] and Spark [38]. Hadoop MapReduce and Dryad allow programmer to write low level optimized code but can greatly reduce programmers productivity. On the other hand, Spark provides high level operations to increase productivity but hard to gain performance by doing relational op- timization. Other systems like Pig and Hive, have a very limited interface to solve a wide range of problems. Programmers are often required to write their own functions which are not convenient and hard to optimize [1]. Jet is a new domain specific framework which provides high level abstraction similar to Spark and performs relational optimization as well as domain-specific optimizations. It is built to construct the intermediate representation of program and generates optimized code for Spark and Crunch [1] which is comparable with hand optimized implementation [1].

2.3.2 Workflow Systems

In recent years, a number of WMSs have emerged in the Big Data community and they are developed to run on top of Hadoop and/or for more general purpose. Within the Hadoop community, a WMS called Apache Oozie is developed to enable user to combine multiple MapReduce jobs into a logical unit of work to accomplish larger

[1]https://github.com/cloudera/crunch.,

tasks or a workflow [18]. Oozie is a Java Web Application that stores the workflow definitions and the currently running workflow instances, including their status (e.g. running, stalled, failed) and variables (e.g. input files, output files). An Oozie workflow is a sequence of actions (e.g. Hadoop MapReduce jobs, Pig jobs) represented in a control dependency DAG that is specified in the XML Process Definition Language. The workflow consists of Control Nodes and Action Nodes. Control nodes define the flow of execution and include start and end node of a workflow as well as the mechanisms to control the workflow execution path e.g. decision, fork, and join nodes whereas action Nodes are the mechanism to allow a workflow trigger the execution of a processing task [18].

Another example of a WMS that is not built specifically for Hadoop is Luigi and is developed by Spotify. Luigi is a Python package that helps developers build complex pipelines of batch jobs [2]. It facilitates developers to combine many tasks together, where each task may be a Hadoop job, a Hive query, loading a table from a database, etc. Luigi takes care of large portion of the workflow management so that the developer can focus on the tasks themselves and their dependencies. One major difference between Luigi and Oozie is that instead of XML configuration, the DAG in Luigi is specified with Python code constructs. This makes it easy to build complex dependency graphs of tasks. Additionally, the workflow can trigger scripts that are not written in Python e.g., Pig scripts [2]. The Luigi WMS consists of a centralized scheduler and a number of workers. The workers communicate with the scheduler over a JSON REST API. In Luigi, the developer defines a job as a Python class. Luigi workers communicate with HDFS and walk through the work-flow DAG and check, for each task, whether a tasks output exists in order to determine the next tasks to be run. After it walks through the DAG, it then runs the tasks e.g. MapReduce jobs [1].

[2]https://github.com/spotify/luigi

Chapter 3

Generating Control-Flow-Enriched Data Flows from User Programs

This chapter revolves around intermediate code generation part of the compiler in which translation of the source program into target code takes place. In the process of translating a program written in a given language into code for a given target machine, a compiler typically constructs a sequence of intermediate representation which can have a variety of forms [21]. High-level representations are close to the source language and low-level representations are close to the target machine. Syntax tree is one of the most commonly used forms of high-level intermediate representation during syntax and semantic analysis. During intermediate code generation phase of the compiler, another set of intermediate representation i.e. control flow graph is produced. In the preliminaries section of this chapter, we present overview of the theory around syntax trees and graph in order to equip the reader with necessary knowledge to understand how to generate the control-flow enriched data flows from given user programs which we talk about later in this chapter.

3.1 Preliminaries

3.1.1 Syntax Directed Translation

Before going through the intermediate code generation phase of the compiler, we first visit its preliminary phase which is the Syntax-Directed Translation phase in which the translation of languages guided by context-free grammars is developed (for formal explanation of context-free grammars, refer to Definition 3.1.1). The parser uses the components produced by the lexical analyzer to create a tree-like intermediate representation depicting the grammatical structure of the source program [21]. This translation technique will be applied in intermediate code generation. The most general approach to syntax-directed translation is to construct a syntax tree and to compute the values of attributes at the node of the tree by visiting all the nodes [21]. Information is associated with the syntax tree by attaching attributes to the grammar symbols representing the programming construct.

Syntax-Directed Definition (SDD) is context-free grammar completed with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. An attribute is any quantity associated with a programming construct. Example of attributes are data types of expressions, the number of instructions in the generated code, or the location of the first instructions in the generated code for a construct. Since we are using grammar symbols (nonterminals and terminals), the notion of attributes is extended from constructs to the symbols that represent them.

A context-free grammar in itself specifies a set of terminal symbols (inputs), another set of nonterminals (e.g. symbols representing syntactic context), and a set of productions. Each of these gives way in which strings represented by one nonterminal can be constructed from terminal symbols and strings represented by other nonterminals. Production consists of a head which is the nonterminals to be replaced and a body which is the replacing string of grammar symbols. There are two kinds of attributes for nonterminals, which are Inherited and Synthesized Attributes. The differences between the two are as follows [21]:

- Synthesized Attributes. A synthesized attribute for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N.

- Inherited Attributes. An inherited attribute for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body.

Definition 3.1.1 (Context-Free Grammars)**.** *A context-free grammar has four components as follow [21]:*

1. *A set of terminal symbols. The terminals are the elementary symbols of the language defined by the grammar.*

2. *A set of nonterminals. Each nonterminal represents a set of strings of terminals.*

3. *A set of productions. Each production consists of a nonterminal, called head or left side of the production, a arrow, and a sequence of terminals and/or nonterminals, called the body or right side of the production.*

4. *One of the nonterminals is designed as the start symbol.*

3.1.2 Abstract Syntax Trees

During syntax-analysis or parsing, the compiler creates syntax-tree nodes to represent significant programming constructs (e.g. operators, classes, control flow etc). This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure that reflects the structure of the program. The purpose of the syntax analysis or parsing phase is to recombine the tokens produced by the lexical analysis as a result of input splitting into a form that reflects the structure of the source program. This form is typically a data structure called the syntax tree or parse tree of the program. As the name indicates, syntax tree is a tree structure [22].

As the analysis continues, information is added to the node in the form of attributes associated to the node depending on the translation to be performed [21].

During syntax-directed translation phase in the compiler, a syntax-directed translator is constructed to translate arithmetic expressions into postfix form (see Definition 3.1.2). Abstract Syntax Tree (AST) is a data structure that is most useful for designing this syntax-directed translator. In an AST for an expression, each interior node represents an operator whereas the children of the node represent the operand of the operator. The difference between AST and the parse tree is that the AST keeps the essence of the structure of the program but omits the irrelevant details [22]. It corresponds to one or more nodes in the parse tree. In general, any programming construct can be handled by creating an operator for the construct and treating the semantically meaningful components of that construct as operands [21]. The AST represents an expression formed by applying the operator **op** to the subexpressions represented by E_1 and E_2. It can be created for any construct, not limited to expressions. Each construct is represented by a node, with children for the semantically meaningful components of the construct. An operator in the abstract syntax is defined for every statement construct. For constructs that begin with a keyword, the keyword is used for the operator (e.g. operator **while** for while-statements).

We define the scope of the language grammar for our thesis as depicted in Figure 3-1 and explain the syntax tree for a language based on this grammar. Syntax-tree *Expr* is used to represent all kinds of expressions, and *stmt* to represent all kinds of statements. Since a block is a grouping of a program, a syntax tree for a program is of type block. When a statement is a block, it has the same syntax tree as the block (hence, $stmt \rightarrow block$). The syntax tree for nonterminal block is simply the syntax tree for the sequence of statements in the block. A sequence of statements is represented by using a leaf **null**. We ignore declarations in our grammar definition since they are not used in the syntax tree. Blocks, with or without declarations, appear to be just another statement construct in intermediate code. Conditionals can be handled by defining two operators **ifelse** and **if** for if-statements with and without an else part, respectively. The syntax tree node for a while-statement and a do-while statement has an operator, which we call **while** and **dowhile**, respectively and two children - the syntax trees for the *expr* and the *stmt*.

$program \rightarrow block$
$block \rightarrow' \{'stmts'\}'$
$stmts \rightarrow stmts_1 stmt$
$stmts \rightarrow \epsilon$
$stmt \rightarrow expr$
$stmt \rightarrow \textbf{if}(expr) stmt_1$
$stmt \rightarrow \textbf{if}(expr) stmt_1 else stmt_2$
$stmt \rightarrow \textbf{while}(expr) stmt_1$
$stmt \rightarrow \textbf{do}(stmt_1) expr$
$stmt \rightarrow block$

Figure 3-1: Context-Free Grammar

Definition 3.1.2 (Postfix Form). *Postfix form for an expression E can be defined as follows [21]:*

1. *If E is a variable or constant, then the postfix form for E is E itself.*
2. *If E is an expression of the form E_1 **op** E_2, where **op** is any binary operator, then the postfix form for E is E_1' E_2' **op** where E_1' and E_2' are the postfix forms of E_1 and E_2 respectively.*
3. *If E is a parenthesized expression of the form (E_1), then the postfix form for E is the same as the postfix form for E_1*

3.1.3 Control-Flow Graphs

The Control-Flow Graph (CFG) is another intermediate representation that is produced during syntax-analysis . Frances E. Allen [4] defines a CFG as "a directed graph in which the nodes represent basic blocks and the edges represent control flow paths". The CFG serves as framework for static analysis of program control flow. Many code generators partition intermediate representation instructions into basic blocks, which consist of sequences of instructions or statements that are always executed together

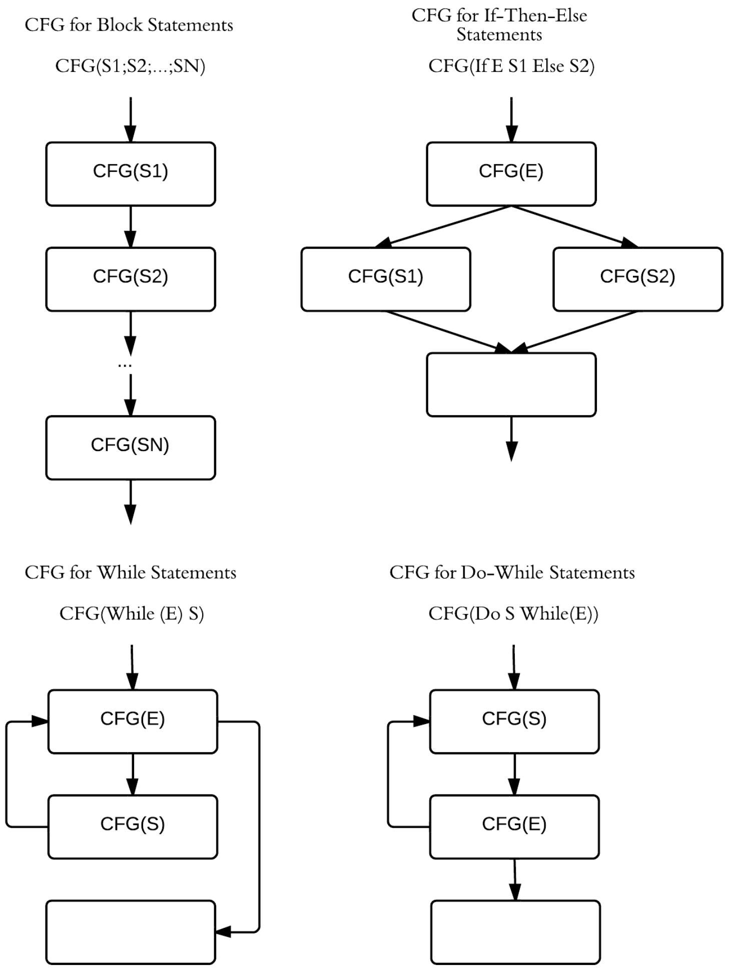

Figure 3-2: CFG of Various Statements

[21]. Basic blocks are a straight line, single-entry code with no branching except at the end of the sequence.

Edges represent possible flow of control from the end of one block to the beginning of the other. There may be multiple incoming or outgoing edges for each block [4]. After the intermediate code has been partitioned into basic blocks, the flow of control between them can be represented by a CFG. There is an edge from block A to block B if and only if it is possible for the first statement in block B to immediately follow the last statement in block A. Given a Block statement, If-Then-Else statement, and While-statement, we formulate the expected CFG result from the first stage of the algorithm as shown in Figure 3-2. CFG(S) is a CFG of high-level statement S. CFG(S) is a single entry and single-exit graph with one entry node and one exit node both in the form of a basic block. In the first stage of our algorithm, construction of CFG(S) is recursively defined and will be presented in detailed in the next section.

3.1.4 Data-Flow Analysis

The CFG introduced in the previous section capture one aspect of dependencies among parts of program [34]. Although CFG depicts the control flow of the program, the transmission of information through program variables is missing from the graph. Data flow models resulted from Data-Flow analysis provide a complimentary view, showing relations involving transmission of information [34].

In the second stage of our algorithm, we analyze the graph to detect data dependencies and subsequently add another type of edges which contain information of the data dependencies between the nodes of the CFG. Data dependencies describe the normal situation that the data some statements or basic blocks use depends on the data created by other statements or other basic blocks. Data-flow analysis aims to derive information about the flow of data along with program execution paths. In analyzing the behavior of a program, all the possible sequences of program points ("paths") through a CFG that the program execution can take must be considered. From the possible program states at each point, we extract the information we need for the particular data-flow analysis problem to be solved. The possible execution paths in a CFG can be defined as follows [21].

- Within one basic block, the program point after a statement is the same as the program point before the next statement.
- If there is an edge from block B_1 to block B_2, then the program point after the last statement of B_1 may be followed immediately by the program point before the first statement of B_2.

In [21], *Execution path* or *path* from point p_1 to point p_n is defined as a sequence of points $p_1, p_2, ..., p_n$ such that for each $i = 1, 2, ..., n-1$, either

1. p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or

2. p_i is the end of some block and p_{i+1} is the beginning of a successor block.

Data-Flow Analysis Schema on Basic Blocks

A **data-flow value** at a program point represents the set of all possible program states that can be observed for that point, for example, all definitions in the program that can reach that point [21]. We denote the data-flows values immediately before and immediately after each basic block B in a CFG of a program by IN[B] and OUT[B], respectively. A transfer function f_B relates the data-flow values before and after a block B. Transfer functions can be either the information that propagate forward along the execution paths, or the information that flow backwards up the execution paths. The **data-flow problem** for a CFG is to compute the values of IN[B] and OUT[B] for all blocks B in the CFG [21].

Suppose block B contains of statements $s_1, ..., s_n$. If s_1 is the first statement of the basic block B, then IN[B] = IN[s_1]. Similarly, if s_n is the last statement of basic block B, then OUT[B] = OUT[s_n]. The transfer function of a basic block B, denoted by f_B is derived by composing the transfer functions of the statements in the block. That is, let f_{s_i} be the transfer function of statement s_i. Then $f_B = f_{s_n} \circ ... \circ f_{s_2} \circ f_{s_1}$. [21] defines the relationship between the beginning and the end of the block as follow:

$$OUT[B] = f_B(IN[B])$$

Given a CFG, in a forward data-flow problem the IN set of a basic block B is computed from the OUT sets of B's predecessors.

$$IN[B] = \cup_{P\ a\ predecessor\ of\ B}\, OUT[P]$$

Vice versa, a backward data-flow problem occurs when the IN set of a basic block B is computed from the OUT set of B's successor. When the data-flow is backwards, which we will see in detail in live-variable analysis subsection, the equations are similar, but the roles of IN and OUT are reversed as follow:

$$IN[B] = f_B(OUT[B])$$

$$OUT[B] = \cup_{S\ a\ successor\ of\ B}\ IN[S]$$

Live Variable Analysis

The purpose of Live variable analysis is to know for variable x and point p whether the value of x at p could be used along some path in CFG starting at p. If so, then x is said to be live at p; otherwise, x is dead at p. The basic motivation of live variable analysis is to manage register allocation. A program contains an unbounded number of variables and must be executed on a machine with bounded number of registers. Two variables can use the same register if they are never in use at the same time (i.e. never simultaneously live). The result of this live-variable analysis is also used to enrich our CFG from the first stage of the algorithm with the data dependencies information. Live-variable analysis is an example of a backward data-flow problem [21].

The point in a program where a value is produced (called a "definition") is associated with the points at which the value may be accessed (called a "use") by the most fundamental class of data flow model [34]. Associations of definitions and uses fundamentally capture the flow of information through a program, from input to output. Definitions occur where variables are declared or initialized, assigned values, or received as parameters, and in general at all statements that change the value of one or more variables. Uses occur in expressions, conditional statements, parameter passing, return statements, and in general in all statements whose execution extracts a value from a variable [34].

[21] define the data-flow equations in terms of IN[B] and OUT[B], which represent the set of variables live at the points immediately before and after block B, respectively as follows:

- define def_B as the set of variables defined (i.e. assigned values) in basic block B prior to any use of that variable in B, and
- define use_B as the set of variables whose values may be used in B prior to any definition of the variable.

These definitions lead to any variable in use_B considered live when entering Block B. Whereas definitions of variables in def_B are dead at the beginning of B. We associate the use and def to the unknowns IN and OUT problem. No variables are live on exit on the program [21]. Hence,

$$IN[EXIT] = \emptyset.$$

A variable is defined live when entering a block if either it is used before redefinition in the block or it is live coming out of the block and is not redefined inside the block. A variable is coming out of the block if and only if it is live when entering one of its successors. These two definitions can be summarized into the following equations.

$$IN[B] = use_B \cup (OUT[B] - def_B)$$

$$OUT[B] = \cup_{S\ a\ successor\ of\ B}\ IN[S]$$

Information flow for liveness is directed backward, opposite to the direction of control flow since in this problem, we want to make sure that the use of variable x at point p is transmitted to all points prior to p in an execution path. Thus, we may know at the prior point that x will have its value used in the later points. To solve a backward problem, IN[EXIT] is initialized instead of OUT[ENTRY]. The solution to liveness equation is not necessarily unique and the aim is to find a solution with the smallest sets of live variables.

3.2 Translating ASTs to CFGs

We divide the problem of translating the source program to target code into three stages. The first stage is to traverse the given AST and transform it to a more low-level intermediate representation which is Control Flow Graph (CFG); the steps are depicted in Figure 3-3. This thesis covers the design of the algorithm and implementation to transform the AST into CFG. A sample Scala program with control flow and iteration is presented in this chapter to show the process and result (refer to Listing 3.3. The second stage is to analyze the CFG and identify the data dependencies between each block of the graph. In the end, we generate the executable code for the underlying system. The driver program of the WMS will then execute this code. The design of the algorithm and expected result of the last two stages are also delivered in this thesis.

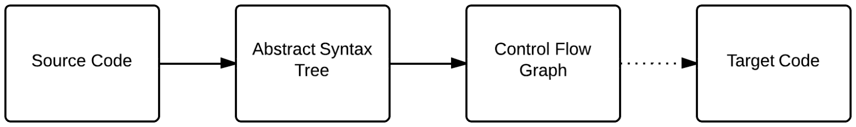

Figure 3-3: Intermediate Representations

3.3 Scala AST

In this thesis, we do not create our own syntax trees representation but reuse the Scala Abstract Syntax Trees (AST) given freely by the Scala compiler's parser and type checker [28]. This section gives necessary introduction to Scala AST to allow the reader to understand the transformation process from the source language to the target language. As described in the previous section, AST is one of the most important intermediate representations. The Scala compiler's parser and type checker provide the Scala AST as intermediate representation that we can directly work on. Additionally, the Scala compiler also provides a tool to traverse and transform an AST [28].

The Scala ASTs are the basis of abstract syntax which is used to represent programs. The Scala compiler uses ASTs as an intermediate representation before generating bytecode [13]. In Scala reflection, Trees can be produced or used by the following APIs[1]:

- Scala annotations. This API uses the AST to represent their arguments and is exposed in `Annotation.scalaArgs`.
- `reify`. This special method takes an expression and returns an AST that represent this expression.

[1]http://lang.org/overviews/reflection/symbols-trees-types.html

- Compile time reflection with macros [8] and runtime compilation with toolboxes use trees as their program representation medium. Macros expand trees at compile time allowing programmers to hack and manipulate AST within the compilation scope [8].

3.3.1 Scala Macros

Compile time metaprogramming is the algorithmic construction of programs at compile-time. Scala macros is an experimental facility to allow user to perform compile-time metaprogramming [9]. With this feature, compiler is enabled to recognize certain methods in Scala programs as metaprograms, or macros, which can be invoked at certain points. When invoked, macros expose a compiler context which includes the compiler's representation of the program being compiled along with an API that provides certain compiler functionality such as parsing and typechecking. Using these API, macros is able to affect compilation e.g. by changing the code being compiled [8].

Defmacros, plain methods whose invocations are expanded during compilation, is the most basic form of compile-time metaprogramming and the one that we work with for this thesis. In the eye of the programmer, def macros appear to look like regular Scala methods with a special property-when a method in a Scala method satisfies this property, that macros definition is expanded by invoking a corresponding metaprogram, called *macroimplementation*. The only fundamental difference with regular method is that macros are resolved at compile time [8]. An example of def macro implementation for `printf` function is depicted in Listing 3.1 below.

```
def printf(format: String, params: Any*): Unit = macro impl
def impl(c: Context)(format: c.Expr[String], params: c.Expr[Any]*): c.Expr[Unit] =
    ...
printf("Hello %s", "world")
```

Listing 3.1: Macros Printf Function [8]

3.3.2 AST Classes

This section introduces some of the concrete trees classes that are used in traversing the trees in our implementation. All concrete classes are case classes, thus their parameters are listed following the class name as follows [28].

- `Block(stats: List[Tree], expr: Tree)`. A Block consists of a list of statements and returns the value of expr. As the name indicates, this class represents Block in the language grammar.

- `ValDef(mods: Modifiers, name: Name, tpt: Tree, rhs: Tree)`. Value definitions are all definitions of vals, vars (identified by the MUTABLE flag) and parameters (identified by the param flag). In Scala, aside from value definitions, ValDef can also contain If statement.

- `LabelDef(name: Name, params: List[Ident], rhs: Tree)`. The LabelDef class is used to represent While and Do-While statement. The Scala language specification [25] defines that the while loop statement `while(e1) e2` is typed and evaluated as if it is an application of `whileLoop(e1)(e2)` where the hypothetical function `whileLoop` is defined in Listing 3.2.

```
def whileLoop(cond: => Boolean)(body: => Unit): Unit = if (cond) { body ;
    whileLoop(cond)(body) } else {}
\label{def:program}
```

  Listing 3.2: WhileLoop Function

- `Assign(lhs: Tree, rhs: Tree)`. Assign trees are used for non-initial assignments to variables. The lhs typically consists of an Ident(name) and is assigned the value of the rhs which normally contains an application (Apply) of a function.

- `If(cond: Tree, thenp: Tree, elsep: Tree)`. An If statement consists of three parts: the condition, the then part and the else part. If the else part is omitted, the literal () of type Unit is generated and the type of the conditional is set to an upper bound of Unit and the type of the then expression, usually Any.

As the name indicates, this class represents the If statement in the language grammar.

3.3.3 Generating Scala AST

Scala macros is used in this thesis to lift the root Block of a Scala program into a monatic comprehension of intermediate representation. We present a sample Scala program with iteration and an If statement inside the iteration (refer to Listing 1.2) and show the generated Scala AST of the program.

```
val e1 = DataSource("/tmp/input1.txt", CsvInputFormat[(String, Int, Int)]())
          .filter(x => x._1 == "Joshua")
val e2 = DataSource("/tmp/input2.txt", CsvInputFormat[(String, Int, Int)]())
          .filter(x => x._1 == "Marten")
var e3: DataSet[(String, Int, Int)] = null
var i = 0

while(i < 0) {
if (e1.map(x => x._2).reduce((x, y) => Math.max(x, y)).fetch().head > 50)
        e3 = e1.map { x => (x._1, x._2 + 1000, x._3)}
    else
        e3 = e2.map { x => (x._1, x._2 + 1500, x._3)}
i = i + 1
}

val e4 = e3.write("/tmp/output.txt", CsvOutputFormat[(String, Int, Int)]())

e4
```

Listing 3.3: Workflow with Conditional

As shown in Scala AST in Figure 1.2, the program is represented by a Block which consists of list of statements and an expression which holds the final return value. Each of the variable definition is presented by a ValDef. The LabelDef in the AST represent the While statement in the program and consists of a name and a rhs of type If. The If statement consists of the three parts: condition, then part, and else part. In the while or LabelDef case, the else part which is of type Literal only contains an empty constant value. The then part is expanded to another list of statements and expression. Given that in the sample Scala program, there is a control flow inside the body of the loop, the statement then consists of another If statement. The then and else part of this If statement are of type Assign since in the program we assign a map function in the rhs to a variable name in the lhs.

3.4 Create CFG from AST Algorithm

Creating CFG from AST is the first stage of the intermediate code and code generation process. This algorithm takes as an input a Scala AST and produces the output of a lower-level intermediate representation CFG $G = (V, E)$ with set V of vertices and a set E of directed edges. Each vertex V is a sequence of one or multiple nodes n in the AST.

The idea is to traverse the tree from top to bottom starting from the *root* and to visit each node n of the children recursively. We check the type of each node n and perform a set of actions accordingly. The procedure $createCFG(nCurr, G, vCurr, X)$ takes as input the following parameters.

- $nCurr$ refers to the node that is currently being visited in the AST. In the beginning, $nCurr$ is initialized to root of the full AST of the program. Since we traverse the tree from top to bottom, the $nCurr$ becomes the root of the subtree of the initial root node. Depending on the type of the Tree, the $nCurr$ is either pushed to the current Vertex $vCurr$ or the subtree of the $nCurr$ is visited recursively.

- $G = (V, E)$ refers to the CFG produced by the procedure and is continuously being updated whenever recursion takes place. The CFG consists of a set of vertices and a set of directed edges. Each vertex of the resulted CFG is a sequence of statements or nodes in the AST. The vertices set V of the graph has initial member of $vCurr$ whereas the edges set E is initialized to an empty set.

- $vCurr$ refers to the vertex of the CFG that is currently being built. If the $nCurr$

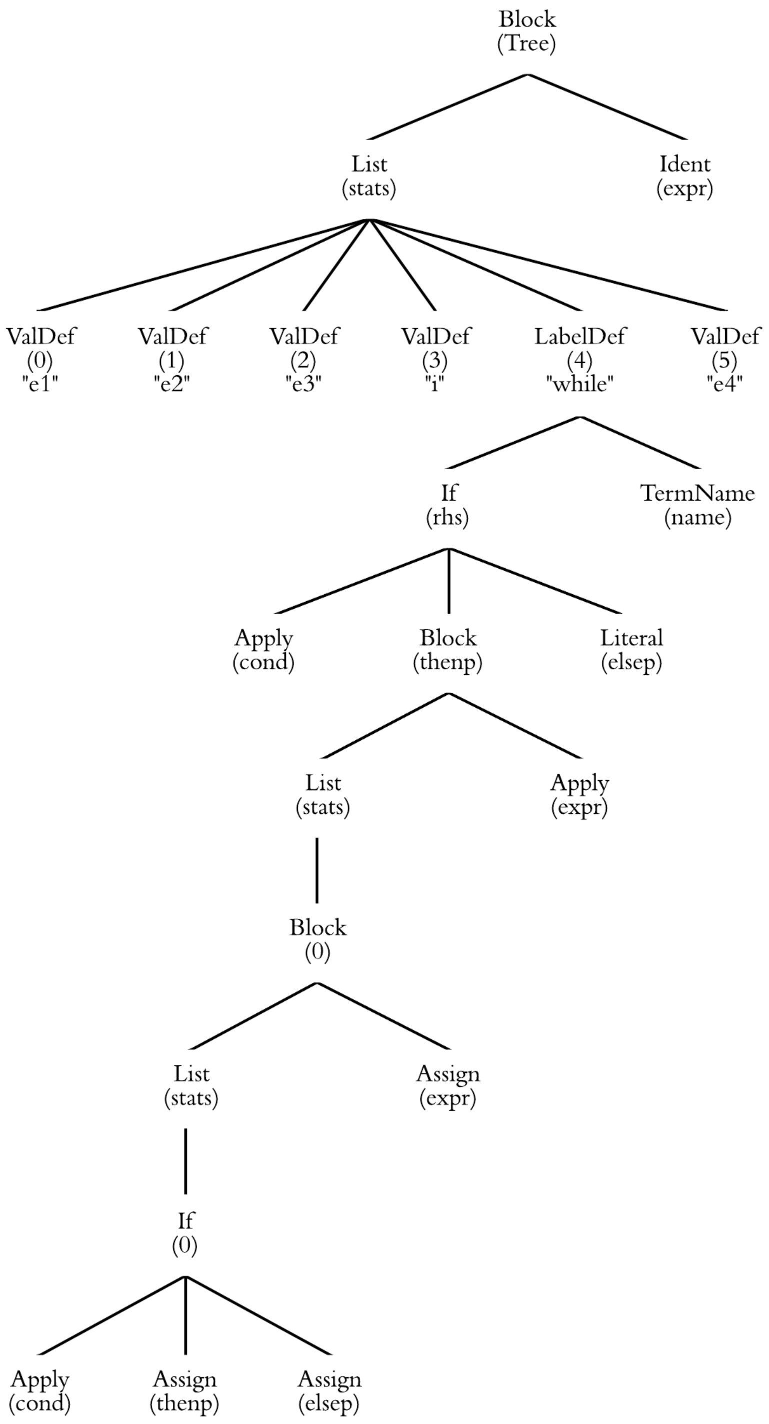

Figure 3-4: Scala AST of Program in Listing 3.3

or a subtree does not contain control flow or branches, the $nCurr$ will be pushed to the $vCurr$. Subsequently, if the subtree contains branches or control flow, new vertex will be added to the CFG as well as directed edges from the $vCurr$ to the new vertex. Furthermore, the new vertex will become the new $vCurr$ and the whole procedure of checking $nCurr$ for control flow is repeated. In the initialization, $vCurr$ is set to empty sequence since we just begin to traverse the tree.

- X is a stack to store the variable which contains branches and needs to be updated with the return value of either the then part or the else part of the control flow. In the algorithm, this variable will be re-assigned to the return value and will be deleted from the stack allowing the stack empty at the end of the procedure.

Using Scala pattern matching, we are able to traverse the tree and match the current node of the tree $nCurr$ to a Scala AST type. Based on its type, the following procedure will be taken.

- $Block(stats, expr)$. A Block typically consists of a list of statements and an expression. For every statement inside the Block, the procedure is called recursively and the Graph $G = (V, E)$, current vertex $vCurr$, and stack of variable X are subsequently updated. After each statement is visited, the procedure is also called recursively for the single expression.

- $ValDef(name, rhs)$ As mentioned in the Scala AST introduction section, ValDef refers to definition of a variable both immutable (val) and mutable (var). In this case, we check whether the ValDef contains branches or control flow. If there is a control flow inside the ValDef, the same ValDef with a mutable identifier is pushed to the current vertex $vCurr$ and the name of the variable of type TermName is stored in the stack variable X. The name of the variable needs to be stored because it must be assigned later on with the value in the Then part or the Else part of the control flow. The rhs then becomes the current node $nCurr$ and the procedure is called recursively. Sample program that will encounter this case is provided in the Listing 1.3 below. In this case, the variable $e3$ is stored in X and later will be assigned to the function "e1.map .." or function "e2.map ..".

```
val e3 = if (e1.map(x => x._2).reduce((x, y) => Math.max(x, y)).fetch().head >
    50)
        e1.map { x => (x._1, x._2 + 1000, x._3)}
else
        e2.map { x => (x._1, x._2 + 1500, x._3)}
```

Listing 3.4: ValDef with Branches

If the ValDef does not contain branches, then the current node $nCurr$ is simply pushed to the current vertex $vCurr$.

- $Assign(name, rhs)$ The procedure for the case Assign is similar with ValDef. We check whether the tree or node contains branches. If it does, the name of the variable is stored in the stack variable X to be assigned later on with the value in the Then part or the Else part of the control flow. The procedure is then called recursively with the rhs as the new current node $nCurr$. Similar to ValDef, if the Tree does not contain branches, the current node $nCurr$ is simply pushed to the current vertex $vCurr$.

- $While(cond, body)$ As depicted in Figure 2-3, for While loop, a new vertex $vCondStart$ is created to store the condition of the iteration. A directed edge is defined from the current vertex $vCurr$ to the new vertex $vCondStart$. A new vertex $vBodyStart$ is also created to store the statements in the *body* of the iteration. Both the *cond* and *body* may or may not contain branches. Hence, recursive call both for *cond* and *body* take place which results in the new vertex $vCondEnd$ and $vBodyEnd$ respectively. A directed edge is defined from the end of the *body* which is the vertex $vBodyEnd$ to the beginning of the *cond* which is the vertex $vCondStart$. At the end, a new vertex is created as the new current vertex $vCurr$ and a directed edge is defined from the vertex $vBodyEnd$ to the new current vertex $vCurr$.

- $DoWhile$(*cond*, *body*) The procedure for DoWhile case is similar with While case with the differences in the order the vertices are created as well as the directed edges connecting the vertices. The *vBodyStart* is created first and is connected to the $vCurr$. The vertex result of the recursive call to the body of the iteration, vertex *vBodyEnd* is connected to the new vertex $vCondStart$ which stores the condition of the iteration. The end of the condition $vCondEnd$ is connected to the start of the *body* $vBodyStart$. Similar to the While loop, at the end, a new vertex is created as the new current vertex $vCurr$ and a directed edge is defined from the vertex $vCondEnd$ to the new current vertex $vCurr$.

- If(*cond*, *thenp*, *elsep*) At first, we create a new vertex $vCond$ to store the condition of the If statement. A directed edge is connected from the current vertex $vCurr$ to the *vCond*. A recursive call is performed to the procedure since the *cond* may or may not contain branches. We then create two new vertices *vThenp* and *vElsep* to represent the Then part and the Else part of the If statement consecutively. A directed edge is defined from *vCond* to both new vertices *vThenp* and *vElsep* to represent all possible flows. Recursive call is performed both for the Then part *thenp* and Else part *elsep* since they may or may not contain branches. At the end, a new vertex is created as the new current vertex $vCurr$ and a directed edge is defined both from the vertex *vThenp* and vertex *vElsep* to the new current vertex $vCurr$.

- $DefaultCase$) We first check whether the stack of variable X contains any member. If yes, then we have get the last member inserted to X which is a name, and assign the current node *nCurr* to the name. This assignment is then pushed to the current vertex $vCurr$. If the stack variable X does not contain any member, which means no assignment is needed, the current node *nCurr* is simply pushed to the current vertex $vCurr$.

We perform this algorithm with the input of the Scala AST of the program in Listing 2-1. The CFG result is shown in Figure 1.4.

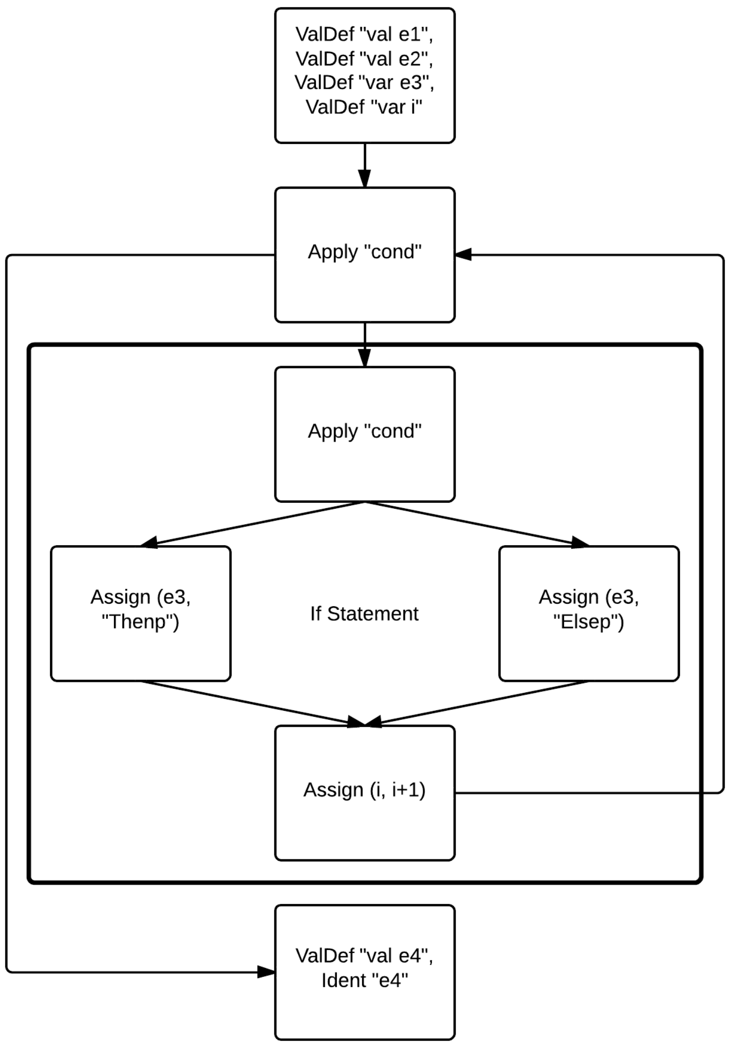
ValDef "val e1", ValDef "val e2", ValDef "var e3", ValDef "var i"
Apply "cond"
Apply "cond"
Assign (e3, "Thenp")
If Statement
Assign (e3, "Elsep")
Assign (i, i+1)
ValDef "val e4", Ident "e4"

Figure 3-5: CFG of Scala AST of sample Program

3.5 Generate Control Flow Enriched Data-Flow

As described in section 3.1.4, the CFG produced from the first stage of algorithm not yet depicts the transmission of information through program variables. Hence, in this stage of algorithm, we perform data-flow analysis on the CFG with the aim to compliment the existing CFG with the information on the flow of data between each vertex of the CFG. The flow of data is designed as another type of directed edge in the CFG which contains information of the variable that is shared from the one vertex to the vertex.

3.5.1 Generate Def-Use Pair

In order to compliment the CFG with the data flow edges, the first step that we need to perform is to compute the set of variables defined and the set of variables used in each vertex of the CFG, which is denoted by def_B and use_B, respectively (refer to theory in section 3.1.4). Each vertex in the CFG is the same as basic block that we see in the theory - we will use the term vertex instead of basic block in this section.

Furthermore, we want to associate the vertex of the graph to the variable in the program that it defines or uses. Let the vertex be B and each variable in the program be v. Then the association between the vertex and variable of the program can be defined as follows:

$def_{(B,v)}\, holds,\ for\, a\, variable\, v\, and\, a\, vextex\, B,\ if\ B\, defines\, v$

$use_{(B,v)}\, holds,\ for\, a\, variable\, v\, and\, a\, vextex\, B,\ if\ B\, uses\, the\, value\, of\, v$

Using these definitions, we generate the Def-Use pair information for each of the vertex in G(V,E) resulted from the first stage of the algorithm using the sample program in Listing 3.3. The complimented CFG is depicted in Figure 3-6.

TODO: Explain figure Def-Use Graph mentioning the Def-Use pair of each vertex.

3.5.2 Adding Data-Flow to the CFG

This subsection is the true essence of our second stage of the algorithm which is to produce a Control Flow enriched Data-Flow by adding another type of directed edges that depicts the information flow on variable of the program. After we compute the Def-Use pair of each vertex in the CFG, we are going to traverse the graph once again to enrich the CFG with another type of edge which is the data-flow edge. The data flow edge is a directed edge depicting the flow of information on program variables between vertices.

The flow of information is derived from the Def-Use pair that we already computed for each vertex. The technique we deploy is by comparing two different vertices and checking whether one vertex $B1$ defines a variable v that the other vertex $B2$ uses. If

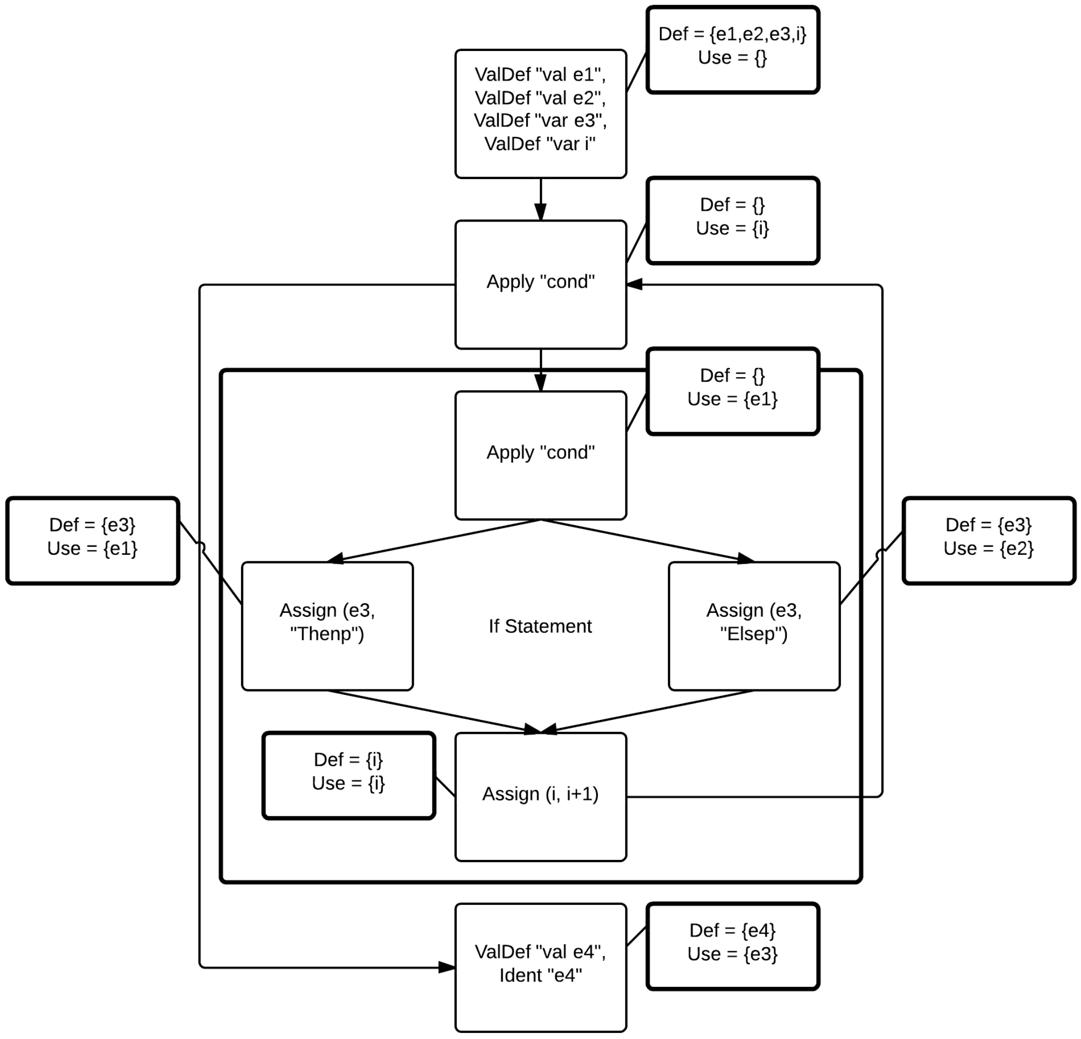

Figure 3-6: CFG with Def-Use Pair Information of Program in Listing 3.3

so, then we draw a directed edge from vertex $B1$ to vertex $B2$ that depicts the data flow of variable v given that $def_{(B1,v)}$ reaches $use_{(B2,v)}$. This last property is related to the definition clear path problem and can be described as follow:

$$def_{(B1,v)}\ reaches\ use_{(B2,v)}\ when\ there\ is\ a\ definition\ clear\ path\ from\ B1\ to\ B2$$

A definition-use pair is formed only if there is a program path on which the value assigned in the definition can reach the point of use without being overwritten by another value. If there is another assignment to the same variable on the path, we say that the first definition is killed by the second. A definition-clear path is a path from definition to use on which the definition is not killed by another definition of the same variable [34]. Hence, the steps to enrich the CFG resulted from previous stage is summarized in the following Algorithm.

Algorithm 2 Adding Data Flow to CFG

```
Input: $G(V, E)$ with $Def-Use$ $Pair$ $for$ $each$ $vertex$
Output: $G(V, E, DFE)$
1: Initialize:
       $DFE \leftarrow \emptyset$
2: procedure createCFDFG($G$)
3:     for $each$ $vertex$ $B1$ $in$ $G$
4:         for $each$ $vertex$ $B2$ $in$ $G$ $other$ $than$ $B1$
5:             for $each$ $variable$ $v$ $in$ $def_{B1}$
6:                 if $There$ $exists$ $use(B2, v)$ $and$ $def(B1, v)$ $reaches$ $use(B2, v)$
7:                     $DFE \leftarrow DFE \cup \{(B1, B2, v)\}$
8:     return $G$
9: end procedure
```

The input of our algorithm is CFG G(V,E) with set of vertices V and a set of directed edges E as well as computed Def-Use pair for each vertex. The expected output of our algorithm is Control Flow Enriched Data Flow G(V,E,DFE) with DFE referring to a set of typed-directed edges that depicts the information flow of a variable from one vertex to another vertex. The result of data-flow analysis performed on CFG produced with sample program in Listing 3.3 is depicted in Figure 3-7 below.

3.6 Generate Code for Underlying System

After the previous stage, we have now an intermediate representation in the form of a control-flow-enriched Data-Flows information. The final phase in our compiler model is the code generator. It takes as input the intermediate representation produced by the previous stage of the algorithm, and produces as output a semantically equivalent

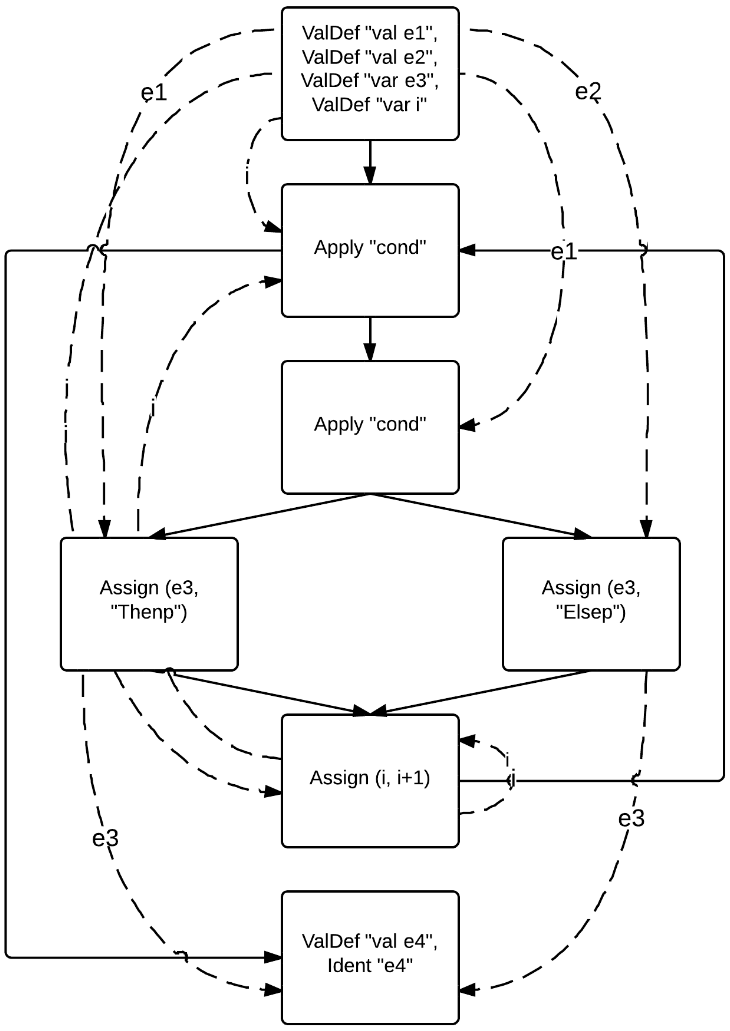

Figure 3-7: Control Flow enriched Data Flow of Program in Listing 3.3

target program for the underlying system [21].

3.6.1 Assumptions

We assume that all syntactic and static semantic errors have been detected and that the necessary type-checking has taken place, and that type-conversion operators have been inserted wherever necessary [21]. The code generator can therefore proceed on the assumption that its input is free of these sorts of errors.

It is also assumed that the code generated by this algorithm will run only for systems with a specified set of primitives. The following are variant expressions that could be lifted into the context:

- DSL defined operations
- Basic Scala types (primitives, Arrays, Lists, Tuples, etc.)
- Functions
- Control structures (If, For, While, ...)
- Equality
- Variable declaration and assignment

3.6.2 Code Generation Algorithm

Our intermediate representation takes the form of a graph in which each vertex in the graph represents the basic blocks which consists of sequences of statements that are always executed together. The objective of this stage is to transform each vertex or basic block to a Stratosphere job. For each job, we will add the necessary environment declarations to enable a driver program or a workflow manager to execute each job in the underlying infrastructure. The workflow program will automatically select the job to be run since flow of the execution is determined by the control flow edge of the intermediate representation. The following Figure 3-8 depicts the role of workflow manager inside the underlying system.

With regards to the data dependencies, we will check data flow edges in the graph. As mentioned in the previous section, each edge is a directed edge which has the information of the source vertex $B1$, destination vertex $B2$ and the data or variable that they are moving v. Each incoming edge to a vertex B shows that the Stratosphere job that represents the vertex B requires the input of the data or variable contained in the data-flow edge. Simultaneously, each outgoing edge from a vertex B shows that the Stratosphere job that represents the vertex B need to output or sink the data or variable in contained in the data-flow edge so that other jobs in

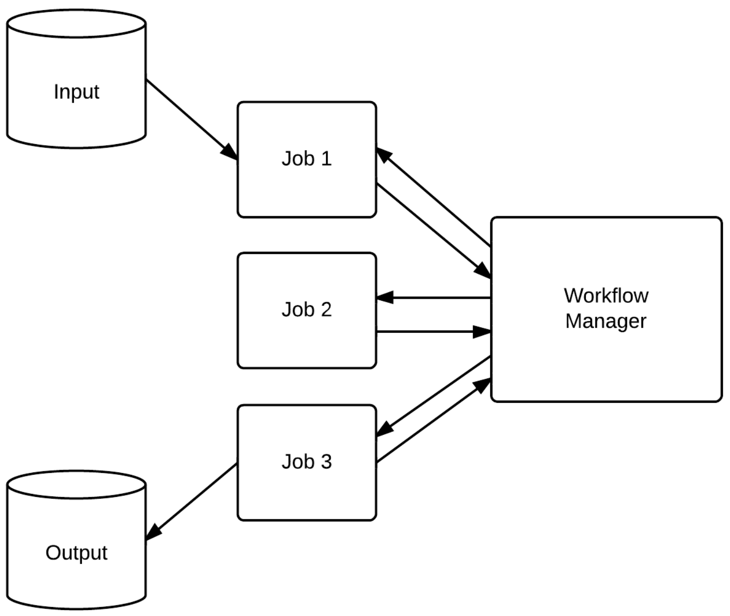

Figure 3-8: Workflow Manager in Underlying System

the workflow can get the data if they need it. At the end, we will need to add a suffix to the job to execute. As a result of the previous stage, the input of this final stage is notated as $G(V, E, DFE)$. The output would be a sequence of Stratosphere jobs notated by J. The algorithm to create the job scripts for the underlying system is summarized in the Algorithm 3 as follows.

Algorithm 3 Generate Code for Underlying System

```
Input: $Intermediate\ Representation\ G(V, E, DFE)$
Output: $Sequence\ of\ Stratosphere\ Jobs\ J$
 1: Initialize:
        $J \leftarrow \emptyset$
 2: procedure createJob($G$)
 3:     for $each\ vertex\ B\ in\ G$
 4:         for $each\ variable\ v\ in\ DFE(any, B, v)$
 5:             $Add\ v\ to\ DataSource\ to\ the\ Job\ j$
 6:         for $each\ variable\ v\ in\ DFE(B, any, v)$
 7:             $Add\ v\ to\ DataSink\ of\ the\ Job\ j$
 8:         $Addexecutionsuffixtothe Jobj$
 9:         $J \leftarrow J \cup \{j\}$
10:     return $J$
11: end procedure
```

Algorithm 1 Creating Control Flow Diagram from AST Part 1

```
1: **Initialize:**
       $v_{curr} \leftarrow [], V \leftarrow \{v_{curr}\}, E \leftarrow \emptyset, X \leftarrow [], n_{curr} \leftarrow root$
2: **procedure** $\textsc{create}\text{CFG}(n_{curr}, G, v_{curr}, X)$
3:     **match** $n_{curr}$
4:         **case** $Block(stats, expr)$
5:             **for** each $s$ in $stats$
6:                 $(G, v_{curr}, X) \leftarrow \textsc{create}\text{CFG}(s, G, v_{curr}, \emptyset)$
7:             $(G, v_{curr}, X) \leftarrow \textsc{create}\text{CFG}(expr, G, v_{curr}, X)$
8:             **return** $(G, v_{curr}, X)$
9:         **case** $ValDef(name, rhs)$
10:            **if** $containsBranches(rhs)$
11:                $\textsc{push}(ValDef(name, \emptyset), v_{curr})$
12:                $\textsc{push}(name, X)$
13:                $(G, v_{curr}, X) \leftarrow \textsc{create}\text{CFG}(rhs, G, v_{curr}, X)$
14:            **else**
15:                $\textsc{push}(n_{curr}, v_{curr})$
16:            **return** $(G, v_{curr}, X)$
17:        **case** $Assign(name, rhs)$
18:            **if** $containsBranches(rhs)$
19:                $\textsc{push}(name, X)$
20:                $(G, v_{curr}, X) \leftarrow \textsc{create}\text{CFG}(rhs, G, v_{curr}, X)$
21:            **else**
22:                $\textsc{push}(n_{curr}, v_{curr})$
23:            **return** $(G, v_{curr}, X)$
24:        **case** $While(cond, body)$
25:            $v_{condStart} \leftarrow \textsc{new}\text{V}()$
26:            $V \leftarrow V \cup \{v_{condStart}\}; E \leftarrow E \cup \{(v_{curr}, v_{condStart})\}$
27:            $(G, v_{condEnd}, X) \leftarrow \textsc{create}\text{CFG}(cond, G, v_{condStart}, \emptyset)$
28:            $v_{bodyStart} \leftarrow \textsc{new}\text{V}()$
29:            $V \leftarrow V \cup \{v_{bodyStart}\}; E \leftarrow E \cup \{(v_{condEnd}, v_{bodyStart})\}$
30:            $(G, v_{bodyEnd}, X) \leftarrow \textsc{create}\text{CFG}(body, G, v_{bodyStart}, \emptyset)$
31:            $v_{curr} \leftarrow \textsc{new}\text{V}()$
32:            $V \leftarrow V \cup \{v_{curr}\}; E \leftarrow E \cup \{(v_{condEnd}, v_{curr}), (v_{bodyEnd}, v_{condStart})\}$
33:            **return** $(G, v_{curr}, X)$
```

Algorithm 1 Creating Control Flow Diagram from AST Part 2

```
34:         case $DoWhile(cond, body)$
35:             $v_{bodyStart} \leftarrow$ NEWV()
36:             $V \leftarrow V \cup \{v_{bodyStart}\}; E \leftarrow E \cup \{(v_{curr}, v_{bodyStart})\}$
37:             $(G, v_{bodyEnd}, X) \leftarrow$ CREATECFG$(body, G, v_{body}, \emptyset)$
38:             $v_{condStart} \leftarrow$ NEWV()
39:             $V \leftarrow V \cup \{v_{bodyEnd}, v_{condStart}\}; E \leftarrow E \cup \{(v_{bodyEnd}, v_{condStart})\}$
40:             $(G, v_{condEnd}, X) \leftarrow$ CREATECFG$(cond, G, v_{cond}, \emptyset)$
41:             $v_{curr} \leftarrow$ NEWV()
42:             $V \leftarrow V \cup \{v_{curr}, v_{condEnd}\}$
43:             $E \leftarrow E \cup \{(v_{condEnd}, v_{curr}), (v_{condEnd}, v_{bodyStart})\}$
44:             return $(G, v_{curr}, X)$
45:         case $If(cond, thenp, elsep)$
46:             $(G, v_{cond}, X) \leftarrow$ CREATECFG$(cond, G, v_{cond}, \emptyset)$
47:             $v_{thenp} \leftarrow$ NEWV()
48:             $V \leftarrow V \cup \{v_{thenp}; E \leftarrow E \cup \{(v_{curr}, v_{thenp})\}$
49:             $(G, v_{thenp}, X) \leftarrow$ CREATECFG$(thenp, G, v_{thenp}, X)$
50:             $v_{elsep} \leftarrow$ NEWV()
51:             $V \leftarrow V \cup \{v_{elsep}\}; E \leftarrow E \cup \{(v_{curr}, v_{elsep})$
52:             $(G, v_{elsep}, X) \leftarrow$ CREATECFG$(elsep, G, v_{elsep}, X)$
53:             $v_{curr} \leftarrow$ NEWV()
54:             $V \leftarrow V \cup \{v_{curr}\}; E \leftarrow E \cup \{(v_{thenp}, v_{curr}), (v_{elsep}, v_{curr})\}$
55:             return $(G, v_{curr}, X)$
56:         case _
57:             if $X \neq \emptyset$
58:                 $name \leftarrow X.head$
59:                 PUSH$(Assign(name, n_{curr}), v_{curr})$
60:             else
61:                 PUSH$(n_{curr}, v_{curr})$
62:             return $(G, v_{curr}, X)$
63: end procedure
64: procedure CONTAINSBRANCHES$(n_{curr})$
65:     match $n_{curr}$
66:         case $Block(, expr)$
67:             CONTAINSBRANCHES$(expr)$
68:         case $If(\_)$
69:             return $true$
70:         case _
71:             return $false$
72: end procedure
```

Chapter 4

Evaluation

This chapter addresses the competitive advantages of our workflow language. In the first section of the chapter, we select a sample use case for large-scale data processing. To justify the programmer productivity advantage and user-friendliness of our language, we show how to define the workflow in our language versus how it is defined in Oozie, the workflow system that we explained briefly in Chapter 2. The final section of this chapter discusses the advantages of our workflow language in terms of generality. We argue that our workflow DSL is extensible to various underlying systems.

4.1 Productivity

We argue that our high-level functional DSL increases productivity. We show this by selecting a good reference of use case, a vehicle GPS probe data ingestion, which has been implemented in Oozie workflow by B. Lublinsky and M. Segel [1].

Use Case: Ingestion Process

The Probes data is delivered to a specific HDFS directory every hour in a form of a file which contains all probes for that particular hour. In this example, the name of the directory is the date for which the data is collected. Probes ingestion is done

[1]http://www.infoq.com/articles/oozieexample

daily for all 24 files for that day. The ingestion process will only start if the amount of files is 24. Otherwise, the following actions will be taken.

- If it is the current day, no action will be taken.
- If it is up to seven days prior to current day, then send a reminder to the probes data provide.
- If the age of directory is seven days or more, ingest all available probes files even though the amount is less than 24.

The overall workflow of the use case is depicted in Figure 4-1.

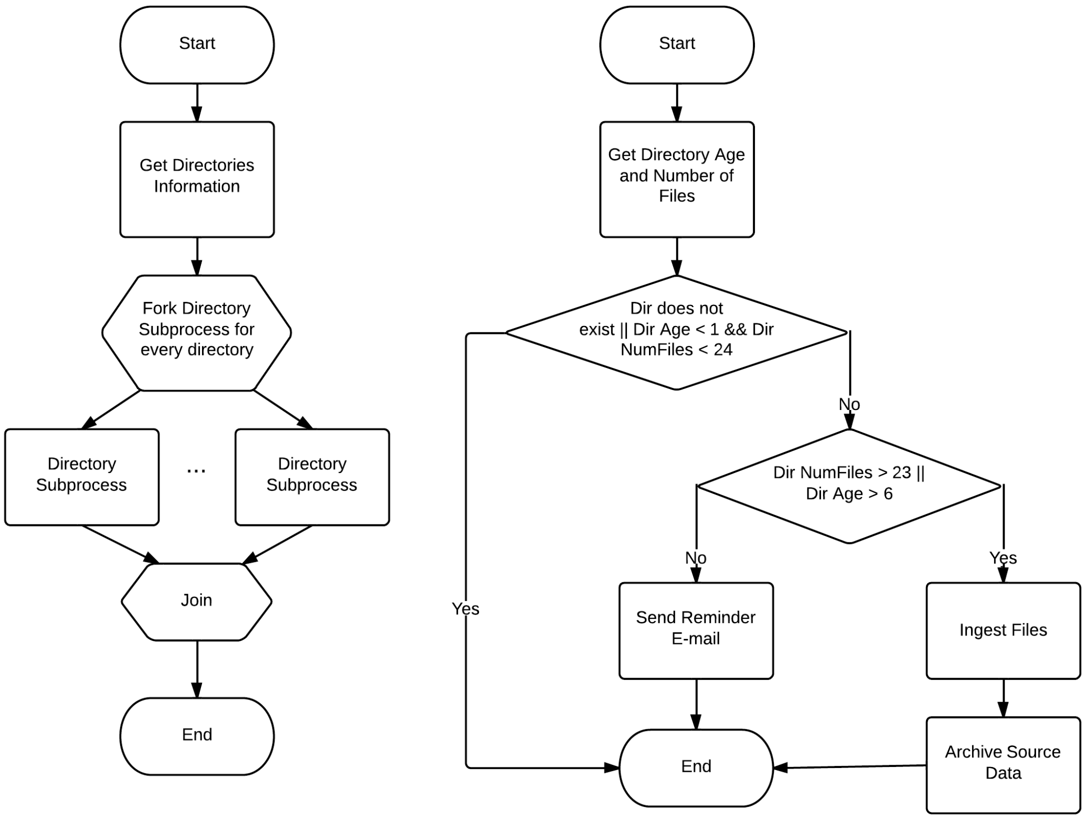
Start
Get Directories Information
Fork Directory Subprocess for every directory
Directory Subprocess
...
Directory Subprocess
Join
End
Start
Get Directory Age and Number of Files
Dir does not exist || Dir Age < 1 && Dir NumFiles < 24
No
Dir NumFiles > 23 || Dir Age > 6
No
Yes
Yes
Send Reminder E-mail
Ingest Files
Archive Source Data
End

Figure 4-1: Ingestion Process and Directory Subprocess [1]

The ingestion process, which is the main process, first calculates the directories names for current day and the six previous days. Then, the seven directory sub-processes are started. The subprocess starts by retrieving information about the directory - its age and the amount of files inside it. It then takes one of the three actions described previously (ingest and archive data, send reminder e-mail, or do nothing) based on the age and amount of files information. In the following subsections, we define and describe the implementation of this process in Oozie workflow and our workflow DSL. Based on this implementation, we later compare and show that our workflow specification is more intuitive and increase the productivity of the programming with way less lines of code to write for the same process.

4.1.1 Oozie Implementation

In Oozie workflow, this implementation is divided into two, the main ingestion process and the subprocess of directory information collection. The programmer need to specify two XML definitions, for the main process and the subprocess, each containing the action nodes and decision nodes based on the overall workflow. Oozie implementation of main ingestion process can be found in Appendix B while the implementation of the subprocess is listed in Appendix C. The main process is pretty straightforward - in the beginning, there is an action node that invokes a java node to calculate a list of directories to process and later starts seven action nodes, each containing a subprocess for each directory in the list. This implementation forces the programmer to write approximately 93 lines of XML code. The input and output directory of each subprocess is also defined manually in the XML definition.

The subprocess in each directory is more complex than the main one. The first action node gets the directory information and also invokes a java routine that we ignore in this evaluation. The next step in the subprocess is a decision node, which decides the action to be performed for the directory. If the directory does not exist or if it is the directory of current day and number of files is less than 24, subprocess will end. If all the files are in the subdirectory or directory is at least 7 days old or more, the subprocess will go to ingest action node which contains a Map/Reduce program to ingest the data and continue to go to action node archive data. This implementation of the subprocess requires the programmer to write approximately 98 lines of XML code giving the total 191 lines of code to implement the overall workflow.

4.1.2 Workflow DSL Implementation

We implement and test the same use case using our workflow DSL. Using our workflow DSL increases the productivity of the programmer since the programmer only need to specify one workflow definition that contains both the main process and subprocess. The implementation of our DSL for this use case can be found in Listing 4.1. Functional programming provides an intuitive way for the programmer to code the process. For example, the fork node in the main process which starts the subprocess for each directory in the list can be replaced by a general while style iteration over the list of directories. The body of the iteration would be the subprocess itself which contains the conditionals branching based on the directory information (age and number of files). This workflow consists of approximately 24 lines of codes, far less codes than its Oozie implementation.

```
var temp = new Directories()
 var dirList = temp.get
 var i: Int = 0

 while (i < temp.getSize)
 {
   var dir = new DirInfo(dirList(i))
   var dirAge = dir.getAge
   var dirSize = dir.getSize

   if(if(dirAge < 1) dirSize > 23 else dirSize > 0)
   {
     if(dirAge > 6 || dirSize > 23)
     {
       var ingest = ingestFile(dir.getName)
       var archive = archiveFile(dir.getName)
     }
     else
     {
       var reminder = sendReminder(dir.getName)
     }
   }

   i = i+1
 }
```

Listing 4.1: Use Case Implementation in Workflow DSL

4.2 Generality

Our workflow DSL provides a high-level declarative interface which adheres only for Stratosphere at the moment. It is deeply embedded with Scala functional programming language - it has the same syntax and semantics as the stardard Scala programming language with some restrictions. We define the restrictions in our workflow DSL grammar in the previous chapter. Other domain specific approach for dataflow such as Pig [26] and Hive [29] allows productivity but their programming model is often too specific and place a high overload for the programmer of learning new language.

Independence of underlying system is also one of the key advantages of our workflow DSL. Even though our workflow DSL is designed to be run on top of Stratosphere, it is also possible to compile a program written in our DSL to other underlying platforms such as Spark and Hadoop MapReduce. The program written in our DSL is analyzed to produce the intermediate representation in a form of control-flow enriched dataflow. This intermediate representation is subsequently transformed into job scripts with properties (e.g. environment variables, execution plan) required by the underlying system that the workflow is intended to run on.

Spark, for example, can understand the general-style if statement and while statement that our DSL grammar support. Spark normally uses For-comprehension for iteration in its programming syntax but we can easily accomodate this to while-statement in our grammar. We show one example of a machine learning algorithm Logistic Regression implemented in Spark [2] in Listing 4.2.

```
val data = spark.hdfsTextFile(...).map(readPoint).cache()
var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
 var gradient = spark.accumulator(Vector.zeros(D))

 data.foreach(p => {
 val scale = (1/(1+exp(-p.y*(w dot p.x))) - 1) * p.y
 gradient += scale * p.x
 })

```

[2] http://laser.inf.ethz.ch/2013/material/joseph/LASER-Joseph-6.pdf

```
 w -= gradient.value
}

println("Final w: " + w)
```

Listing 4.2: Logistic Regression in Spark

We show that we can accomodate this into our workflow DSL with limited changes. The logistic regression algorithm implemented in our workflow DSL is shown in Listing 1. Since For-Comprehension is not yet supported, we modify the iteration in a general while-statement.

```
val data = spark.hdfsTextFile(...).map(readPoint).cache()
var w = Vector.random(D)

while(i < ITERATIONS) {
        w -= data.map(p => {
        val scale = (1/(1+exp(-p.y*(w dot p.x))) - 1) * p.y
        scale * p.x
        }).reduce(_+_)
    i = i + 1
}

println("Final w: " + w)
```

Listing 4.3: Logistic Regression in Our DSL

Chapter 5

Conclusion

- Running on multiple systems - Parallelization Optimization

Bibliography

[1] Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. Jet: An embedded dsl for high performance big data processing. In *International Workshop on End-to-end Management of Big Data (BigData 2012)*, number EPFL-CONF-181673, 2012.

[2] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, pages 1–26, 2014.

[3] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Mapreduce and pact-comparing data parallel programming models. In *BTW*, pages 25–44, 2011.

[4] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[5] Adam Barker and Jano Van Hemert. Scientific workflow: a survey and research directions. In *Parallel Processing and Applied Mathematics*, pages 746–753. Springer, 2008.

[6] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130. ACM, 2010.

[7] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10. IEEE, 2008.

[8] Eugene Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, page 3. 2013.

[9] Eugene Burmako and Martin Odersky. Scala macros, a technical report. In *Third International Valentin Turchin Workshop on Metacomputation*, number EPFL-CONF-183862. Citeseer, 2012.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Ewa Deelman, James Blythe, Yolanda Gil, and Carl Kesselman. Workflow management in griphyn. In *Grid Resource Management*, pages 99–116. Springer, 2004.

[12] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, 2009.

[13] Mathieu Demarne, Adrien Ghosn, and Eugene Burmako. Scala ast persistence. Technical report, 2014.

[14] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment*, 5(11):1268–1279, 2012.

[15] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[16] David Hollingsworth and UK Hampshire. Workflow management coalition the workflow reference model. *Workflow Management Coalition*, 68, 1993.

[17] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[18] Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. Oozie: towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 4. ACM, 2012.

[19] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

[20] Peter M Kelly. Applying functional programming theory to the design of workflow engines. 2011.

[21] Monica Lam, Ravi Sethi, JD Ullman, and Alfred Aho. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[22] Torben Ægidius Mogensen. *Basics of Compiler Design*. Torben Ægidius Mogensen, 2009.

[23] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[24] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[25] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[26] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[27] Cesare Pautasso and Gustavo Alonso. Parallel computing patterns for grid workflows. In *Workflows in Support of Large-Scale Science, 2006. WORKS'06. Workshop on*, pages 1–10. IEEE, 2006.

[28] Mirko Stocker. *Scala Refactoring.* PhD thesis, HSR Hochschule für Technik Rapperswil, 2010.

[29] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[30] Wil MP van Der Aalst, Arthur HM Ter Hofstede, Bartek Kiepuszewski, and Alistair P Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.

[31] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[32] Gregor von Laszewski, Mihael Hategan, and Deepti Kodeboyina. Work coordination for grid computing. *Grid Technologies: Emerging from Distributed Architectures to Virtual Organizations*, 5:309–329, 2007.

[33] Ustun Yildiz, Adnene Guabtni, and Anne HH Ngu. Towards scientific workflow patterns. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, page 13. ACM, 2009.

[34] Michal Young. *Software testing and analysis: process, principles, and techniques.* John Wiley & Sons, 2008.

[35] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, 2005.

[36] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[38] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.