Workflow Management System for Stratosphere: Design and Preliminary Implementation

by

Suryamita Harindrari

Submitted to the IT4BI Consortium and
Department of Computer Science and Electrical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

TECHNISCHE UNIVERSITÄT BERLIN

August 2014

Author .....
IT4BI Consortium and
Department of Computer Science and Electrical Engineering
August 10, 2014

Certified by .....
Asterios Katsifodimos, PhD
Thesis Advisor

Certified by .....
Prof. Dr. Volker Markl
Thesis Supervisor

Workflow Management System for Stratosphere: Design and Preliminary Implementation

by

Suryamita Harindrari

Submitted to the IT4BI Consortium and
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 develop the preliminary scope of a Workflow Management System (WMS) which is aimed to work on top of Stratosphere, one of the emerging large-scale data processing frameworks. The WMS is developed by means of a Domain Specific Language (DSL) which is deeply embedded in Scala high-level programming language. The aim of this workflow DSL is to enable the progammer to define the workflow of complex use cases without having to manually specify the dependencies between the tasks in the workflow. Control Flow and data dependencies are automatically detected by static analysis on the program code using our compiler framework.

The goal to translate the user program written in our DSL to the target code is achieved through the following three stages: (1) generate a control flow graph as an intermediate representation (IR) from Abstract Syntax Trees (AST) of the program, (2) perform data flow analysis to enrich the graph with data dependencies information, and (3) generate code or job scripts for the underlying system. This research develops the algorithm for each of three stages as well as the implementation of the first stage of the overall process. In the evaluation, we argue over the advantages of this DSL compared to related WMS work in terms of productivity and generality i.e. extensibility to other underlying platforms.

Thesis Advisor: Asterios Katsifodimos, PhD
Thesis Supervisor: Prof. Dr. Volker Markl

Acknowledgments

I would like to offer my sincere gratitude to my Thesis Advisor, Asterios Katsifodimos, for the opportunity, valuable guidance, honest feedbacks, patience, and motivation given to me throughout the making of this thesis. Most importantly, for believing that I could do the work even though I often times have doubts in myself.

A big thank you goes to Alexander Alexandrov for sharing his great knowledge on compiler and the likes, for showing me how to break down the problem into stages which is a very valuable input for the end product of this thesis, for sharing his technical expertise, and for his valuable feedbacks.

I would also like to thank Aljoscha Krettek and Andreas Kunft for introducing me to Scala macros and for being helpful whenever I encounter any technical issues along the road. To Andreas, also for giving important feedbacks on my thesis write-up.

I would like to offer my gratitude to the IT4BI consortium and everyone involved in it as well as the European Commission for the opportunity and for making all this possible especially to Prof. Esteban Zimanyi. Your dream of the IT4BI program opens up ways for many of us and we are forever grateful for that.

Thank you to my fellow IT4BI classmates and friends, especially my colleagues at TU Berlin - Silvia, Janani, Pino, Faisal, and Prateek. Thank you for the company and great team work during the past two semesters and for your support throughout the thesis development. And to my IT4BI friends in Barcelona, thank you for this two-years of incredible journey.

I would also like to thank my wonderful sister, friends, and extended family who share their support and motivation during this process. Especially to my sister for always reminding me to stay calm and that I can accomplish my tasks all right.

I dedicate this thesis to my Mum and my late Dad, the best parents that I could ever hope for, for their everlasting love and support, for showing me the value of hard work, for always reminding me to use give my all to everything that I do, and for always being there even in times when I fail to do so to remind me that in the end, everything is going to be all right.

Contents

List of Figures

List of Tables

List of Algorithms

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 the third step is 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 workflow of the first use case is depicted in Figure 1-1. 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].

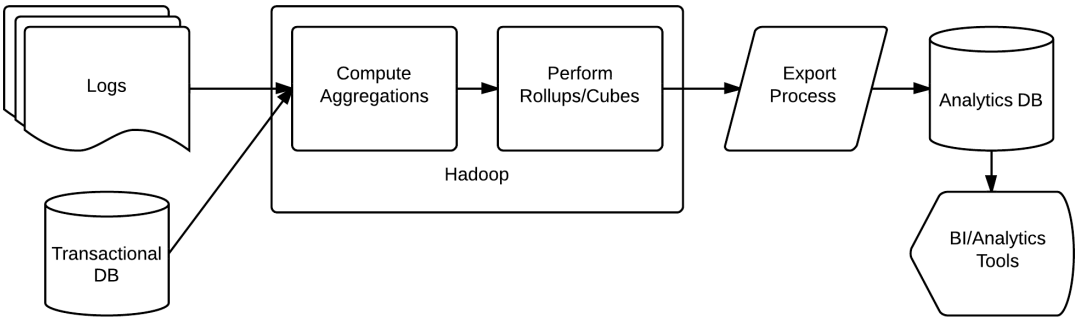

Figure 1-1: Workflow for Analytics/Data Warehousing [1]

1.1 Motivation

The current existing WMS, which will be explored further in section 2.3, 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 specification of dependencies causes a large overhead 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 preliminary develop a WMS prototype that works on top of Stratosphere [2], an emerging large-scale data processing framework developed by 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 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, 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.

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 (IR) 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 IR which is the control-flow-enriched data flow to the script of the jobs. 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 DSL to generate the IR and final job scripts to be run in the WMS. The overall framework of our DSL, compiler, and WMS is depicted in Figure 1-2. 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 IR in the form of a control flow graph, and
- 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 with regards to the second and third stage of the WMS development.

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 an Oozie [3] workflow, an existing workflow system for Hadoop, nodes in the DAG are forward-chain, that is, the programmer is required to specify manually where a node or a computation in the DAG goes after it finishes. This can be hard to track and it forces the developer to remember every node in the chain when developing the workflow [4]. Thus, the Scala DSL that we aim to develop will 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, both control flow and data dependencies, will be discovered by Scala code analysis. In performing the code analysis, we identify the self-contained jobs within branches of the Abstract Syntax Tree generated by the Scala compiler before finally generate the code for the underlying system.

In the evaluation (Chapter 4), we argue over the advantages of this DSL compared to related WMS work in terms of productivity and independence of underlying platform by selecting a use case that is representative of use cases running on large-scale

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

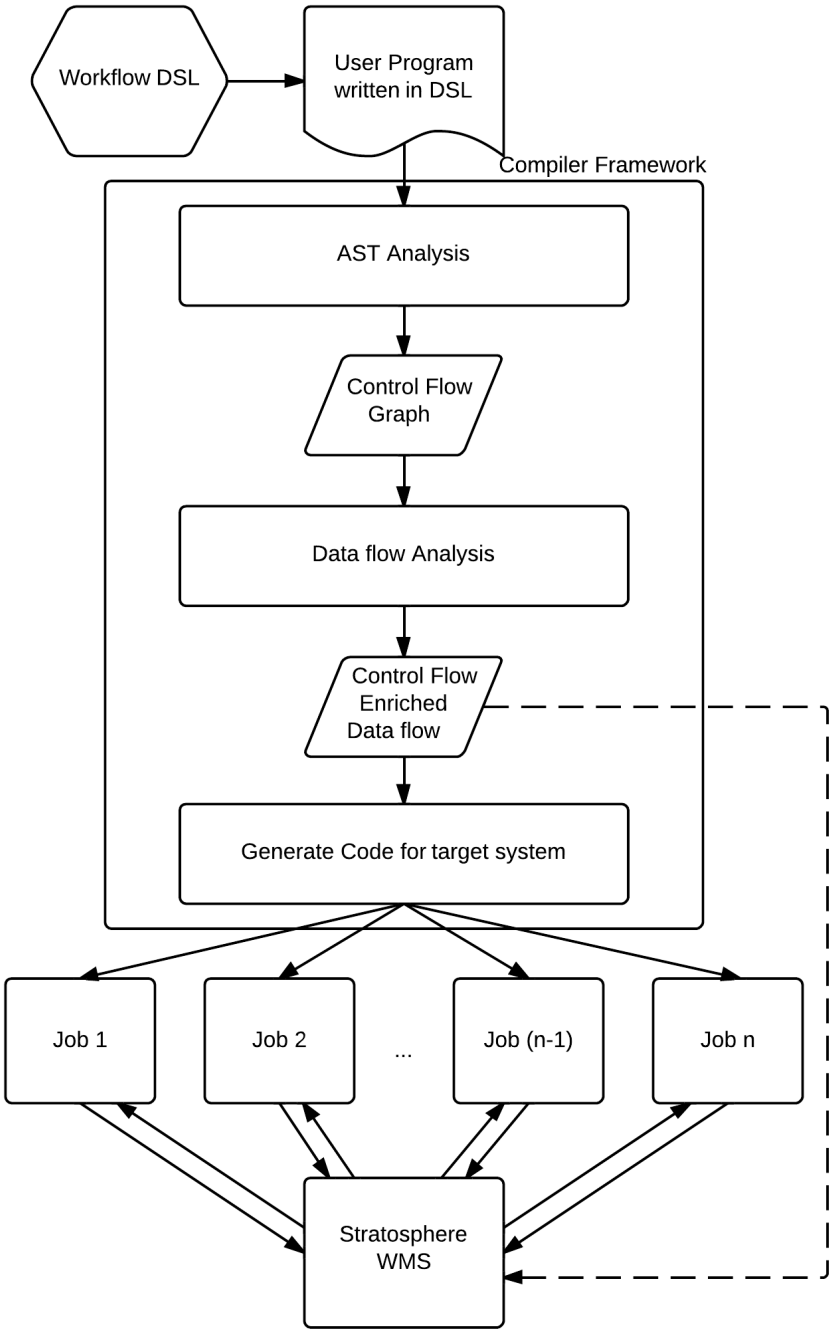

Figure 1-2: Overall Framework for DSL, Compiler, and WMS

data processing platform. We show that even though currently our workflow DSL can only run on Stratosphere, it can easily be extended to be used on another system.

Definition 1.1.1. *A domain specific language (DSL) is "a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain". [31]*

1.2 Thesis Outline

Chapter 2 contains the background and related work on workflow systems and data flow systems. When presenting the literature survey, we also introduce the approach of our workflow model and workflow language.

Chapter 3 presents the implementation part of our DSL, compiler stages, and WMS. We define the grammar of our workflow language and talks about the process of transforming the user program to the target code for the underlying system through the three stages.

Chapter 4 discusses about the advantages of our workflow DSL in terms of ease of use and extensible to other platform by means of implementing a sample use case.

Chapter 5 is the conclusion in which we summarize our contribution throughout this research and present the limitations and potential future works.

Chapter 2

Background, Approach, and Related Work

The study and development of workflows and WMS 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 which includes workflow design and workflow language, 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 data flow systems especially those developed to run on top of large-scala data processing platforms.

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. 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 WMS [32] (refer to Definition 2.1.2 for a formal definition of WMS). Workflow languages are some kind of meta 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 computerized facilitation or automation of a business process, in whole or part" [16].*

Definition 2.1.2. *Workflow Management System is "a system that completely defines, manages and executes workflows 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 design of a workflow 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 may be represented as a DAG or non-DAG. The workflow that we develop in this thesis classifies into 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 runtime 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. Iteration is often occurred in workflow of complex use cases [35].

Workflow Model, or workflow specification, defines a workflow in terms of 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 in a form of control flow enriched data flow. Furthermore, we also provide the algorithm to achieve the concrete workflow, namely the job scripts to be executed in the underlying system. The representation of abstract workflow and concrete workflow of our design will be presented in details in Chapter 3.

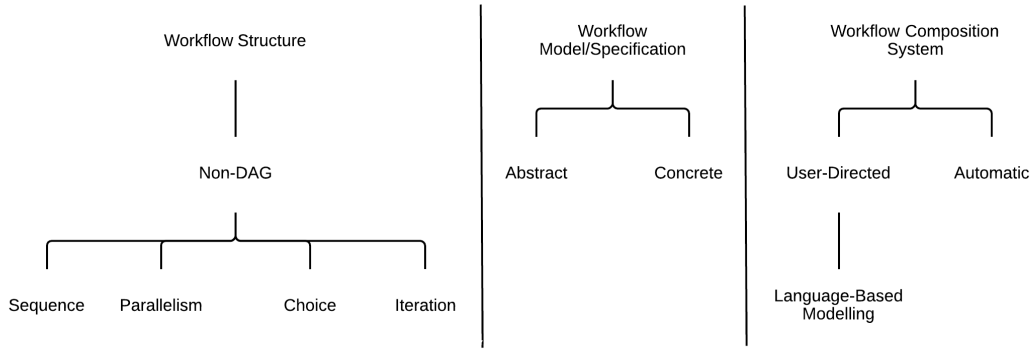

Figure 2-1: Thesis Workflow Design Approach

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 adopted in this thesis. More specifically, in the user-directed, language-based modeling is applied since we require the users to write their program in our workflow DSL. However, the control flow and data dependencies will later be identified automatically by performing the static analysis of the user program. In this case, users are not required to specify manually the workflow components and dependencies. In summary, 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 describes these concepts and map them in a high-level manner to our programming model in the following points:

- **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. Whereas, in our concrete workflow, a task corresponds 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, whereby 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 executed 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 within the loop body are executed with different input value, given that the condition of the iteration is satisfied. Our workflow language supports both conditional branching and iterations.

- **Representation** of a workflow normally consists of a graph, whereby nodes correspond to tasks and edges to dependencies. Parallel execution uses the dependency information to determine which tasks can be executed in parallel. The representation of our workflow language will be presented in detail in Chapter 3.

2.2 Control Flow vs Data Flow

In addition to the three taxonomies described in section 2.1.1, most, if not all, workflow designs belong to one of the two classes: control flow or data flow. The two classes are similar in that they specify the interaction between individual tasks in the workflow. The difference between the two classes lies in their method of implementing that interaction. In control-driven workflows, or control flows, the connections between the tasks in a workflow represent a transfer of control from the preceding task to successor task. This includes control structures such as sequences, conditionals branching, 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 consumer [12]. More details explanation on control flow and data flows are presented in the following subsections.

2.2.1 Control Flow Model

Most control flow languages provide support not only for simple flows of control between components 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 or inequivalent (i.e. less than, greater than, etc) [12].

The term control flow refers to 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 in a User Defined Function (UDF) are supported by most large-scale data processing systems such as Stratosphere [3] and Spark [38]. However, Stratosphere does not support conditionals branching on the outer level or outside UDFs which is the main motivation to introduce workflow on top of it. Spark, on the other hand, also allows the user to define workflows with conditionals branching and iterations [37]. In 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 a branching or an iteration. Below is the example of expression in a condition test that can be understood by Spark API for branching and iteration, respectively.

```
if (A.reduce(_ + _) < 30)
while (B.reduce(_ + _) < 10000)
```

2.2.2 Data Flow Model

In a data flow model, a workflow is represented by a task dependency graph, in which the nodes of the graph correspond to tasks and edges to 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 descriptions and the data dependencies between them. Control constructs such as loops and branching are generally not included in a data flow. The dependencies between tasks are data dependencies that ensures the data producer has finished before the consumer may start [12]. The key advantage is that, in data flow, 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 data flow in Stratosphere is a DAG which consists of operators, data sources, and sinks or output of the program. The data sources and sinks, as well as the intermediate data sets that flow through operators, are bags of records. For workflow with iterations, a data flow construct of several operators is therefore a function, whose fixpoint we can find by terminating the loop in the data flow DAG [14].

2.3 Related Work

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

2.3.1 Data Flow Systems

Many existing systems for cluster computing have been developed in the past decade such as 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 optimization [1].

Pig is a procedural data flow system for MapReduce. It offers a SQL-style high-level data manipulation constructs, which can be assembled in any explicit data flow 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 data flow 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 is thus required to define them externally in another language, which will often prevent optimizations as Pig can not analyze those [1].

Hive, is a solution which is similar with Pig. It supports queries expressed in a declarative language with the same feel as SQL - called HiveQL. These queries are compiled into MapReduce jobs and subsequently executed using Hadoop [29]. Both Hive and Pig, 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. Its main aim is to construct the IR of program and generates optimized code for Spark and Crunch [1] which is comparable with hand optimized implementation [1]. Jet is built upon the Lightweight Modular Staging (LMS) library [**?**]. LMS uses facilities provided by Virtualized Scala, an experimental version of Scala

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

which provides facilities for DSLs [?], to build a modular compiler infrastructure for developing staged DSLs.

Another existing data flow system which is developed at Microsoft is Naiad [23]. It offers a new computational model called timely data flow whose aim is to enrich data flow 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].

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 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. Luigi is a Python package, built in Spotify, 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. Similar to Oozie, the developer need to define the tasks and their dependencies themselves. One

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

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 easier to build complex dependency graphs of tasks. Additionally, the workflow can trigger scripts that are not written in Python e.g., Pig scripts [2]. In Luigi, the developer defines a job as a Python class. Luigi workers communicate with HDFS and walk through the workflow DAG and perform data flow analysis. During the data flow analysis, Luigi checks, for each task, whether a tasks output exists in order to determine the next tasks to be run. After it finishes, it then runs the tasks e.g. MapReduce jobs [1].

Spark [38], as mentioned in section 2.2.1, supports generalized workflows. In contrast to MapReduce, where all jobs are expressed in rigid map and reduce semantics, Spark jobs can describe arbitrary workflows with one or multiple stages of processing [3]. This is enabled by advanced DAG engine feature that Spark has which supports cyclic data flow and in-memory computing. Each Spark job creates a DAG of task stages to be executed on the cluster [37].

[3] http://www.mapr.com/products/product-overview/apache-spark

Chapter 3

Generating Control-Flow-Enriched Data Flow and Target Code from User Program

As mentioned in Chapter 1, we would like to take a user program written in our workflow DSL as an input, perform code analysis on it in order to detect control flow and data dependencies, and output the target code for the underlying system. This chapter presents in detail the stages of this translation process. In the process of translating a program written in a given language into code for a given target system, a compiler typically constructs a sequence of IR which can have a variety of forms [21]. Syntax tree is one of the most commonly used forms of high-level IR during syntax and semantic analysis. In this chapter, we will look in details into how we generate, analyze and transform the syntax tree of the user program into our own IR (the control-flow-enriched data flow) and subsequently how this IR is used to produce code for Stratosphere. In the preliminaries (section 3.1), we present overview of the theory around syntax trees and graph in order to equip the reader with necessary knowledge to understand the whole translation process.

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 syntax analyzer or parser uses the components produced by the lexical analyzer to create a tree-like IR 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. Examples 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 - refer to Definition 3.1.1), 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.

Definition 3.1.1 (Context-Free Grammars)**.** *A context-free grammar consists of 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

As mentioned previously, 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].

An 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). A whole user program, for example, is one AST. Each operator inside the program is also an AST. However, in a global view, these ASTs are just subtrees of the complete program AST.

We define the scope of the language grammar for our thesis and it is summarized in Table 3.1. Syntax-tree *Expr* is used to represent all kinds of expressions, and *stmt* to represent all kinds of statements. We describe the AST representation of each grammar as follows:

- **program**. Since a block is a grouping of a program, an AST for a program is of type block.

- **block**. The AST for a block is simply the AST for the sequence of statements in the block. A sequence of statements is represented as a block (or sequence of statements) in the grammar. Blocks, with or without declarations, appear to be just another statement construct in intermediate code.

- **declaration(s)**. Variable declaration typically refers to type and id assignment to a variable. Our language also supports control flow and reassignment of the value to the variable in a variable declaration. The last returned value from a conditionals branching or an iteration declared in a variable declaration will be assigned back to the variable. Hence, $decl \rightarrow stmt$. We will show an example of this case in section 3.3.2.

program	$\rightarrow$	*block*
block	$\rightarrow$	$'\{'$ *decls stmts* $'\}'$
decls	$\rightarrow$	$decls_1$ *decl*
	$\rightarrow$	ϵ
decl	$\rightarrow$	**type id**
	$\rightarrow$	*stmt*
stmts	$\rightarrow$	$stmts_1$ *stmt*
	$\rightarrow$	ϵ
stmt	$\rightarrow$	*block*
	$\rightarrow$	**if** (*expr*) $stmt_1$
	$\rightarrow$	**if** (*expr*) $stmt_1$ *else* $stmt_2$
	$\rightarrow$	**while** (*expr*) $stmt_1$
	$\rightarrow$	**do** ($stmt_1$) *expr*
expr	$\rightarrow$	**if** (*expr*) **bool** *else* **bool**
	$\rightarrow$	*rel*

Table 3.1: Grammar Definition

- **statement(s)**. When a statement is a block, it has the same syntax tree as the block (hence, *stmt* $\rightarrow$ *block*). A statement can also refer to control flow AST. Conditionals branching 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 **do-while**, respectively and two children - the AST for the *expr* and the *stmt*.

- **expression**. Expression can be an relational comparison operators (denoted as *rel* here) such as less than or greater than. In our language, expression can also contain branching in a form of if-else statement which returns a boolean value.

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 Graph

Control Flow Graph (CFG) is a form of IR which is produced during syntax-analysis phase of the compiler. 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 IR instructions into *basic blocks*, which consist of sequences of instructions or statements that are always executed together [21]. Basic blocks are a straight line, single-entry code with no branching except at the end of the sequence.

Edges in a CFG represent possible flow of control from the end of one basic block to the beginning of another. We will we refer to basic block as *block* for the rest of this thesis. There may be multiple incoming or outgoing edges for each block [4]. After the intermediate code, in our case the AST, has been partitioned into blocks, the flow of control between them can be represented by the edges. 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. We consider various input (e.g. block statement, if-else statement, and while statement) and formulate the expected CFG result of the partitioning the statements into blocks process as shown in Figure 3-1. We define CFG(S) as 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 block. It is important to highlight that in the case of CFG for Block Statements, if statements S1, S2, ..., SN appears to be sequential (no control flow), then the CGF will consist of only one block comprising all the statements. In the first stage of our algorithm, construction of CFG(S) is recursively defined and will be presented in details in section 3.4.

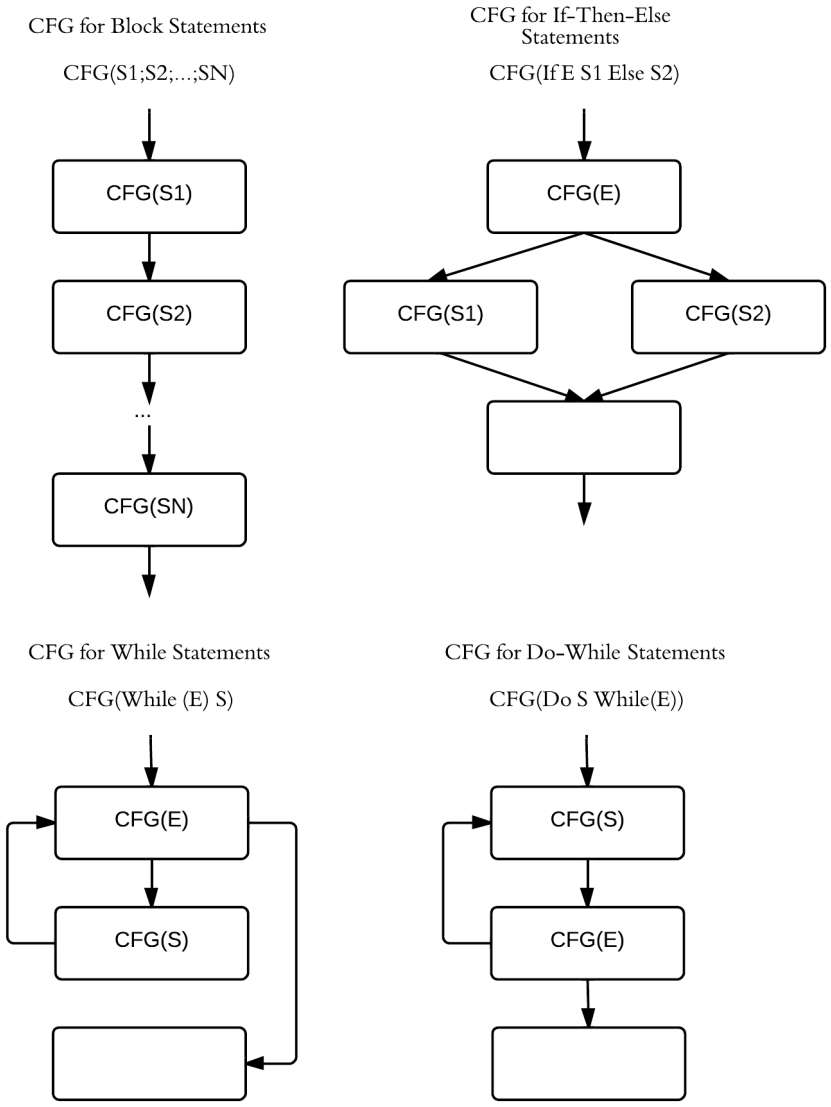

Figure 3-1: CFG of Various Statements

3.1.4 Data Flow Analysis

The CFG described in section 3.1.3 captures 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 which are resulted from data flow analysis provide a complimentary view to the graph, showing relations involving transmission of information [34].

In the second stage of our algorithm, we analyze the CFG to detect data depen-

dencies and subsequently add another type of edges which presents information on the data dependencies between the blocks of the CFG. Data dependencies describe the normal situation that the data which some blocks use depends on the data created by other blocks. Data flow analysis aims to derive the 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 program points in 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 flow 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 a block B contains a sequence of statements $s_1, ..., s_n$. If s_1 is the first statement of the block B, then IN[B] = IN[s_1]. Similarly, if s_n is the last statement of block B, then OUT[B] = OUT[s_n]. The transfer function of a 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 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 block B is computed from the OUT set of B's successors. When the data flow is backwards, which we will see in detail in section 3.1.4, the equations are similar, but the roles of IN and OUT are reversed such that:

$$IN[B] = f_B(OUT[B])$$
$$OUT[B] = \cup_{S \, a \, successor \, of \, B} IN[S]$$

Live Variable Analysis

The purpose of live variable analysis is to know for a 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 used to enrich our CFG with the data dependencies information. In our case, this can lead to a

optimization potential such as parallelization of some tasks if they do not share a live variable between them. Live-variable analysis is an example of a backward data flow problem [21].

In a data flow model, 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") [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:

- Let def_B be the set of variables defined (i.e. assigned values) in block B prior to any use of that variable in B, and
- Let use_B be 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 [21].

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

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

Information flow for liveness is directed backward, opposite to the direction of control flow since, in this problem, we aim 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 [21].

3.2 Translating ASTs to CFGs

We start describing the application of the theory that are introduced in the previous sections to our thesis. As described in Chapter 1, 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 another type of IR which is CFG; the steps are depicted in Figure 3-2. This thesis covers the design of the algorithm and implementation to transform the AST into CFG. A sample Scala program with control flow (refer to Listing 3.3 and iteration is presented in this chapter to show the process and result of our algorithm and implementation. 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 target code. The design of the algorithm and expected result of the last two stages are also delivered in this thesis and are presented in section 3.5 and 3.6, respectively.

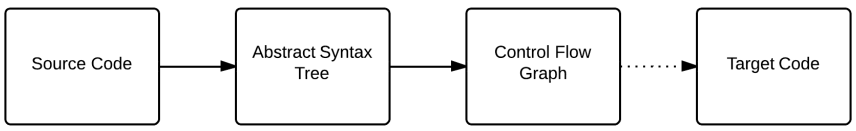

Figure 3-2: Intermediate Representations

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 section 3.1.2, AST is one of the most important forms of IR. The Scala compiler's parser and type checker provide the Scala AST as IR 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 IR before generating bytecode [13]. In Scala reflection, AST 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 as a parameter and returns an AST that represent this expression.
- 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]. One of the reasons to perform this at compile time and not on runtime is to avoid runtime overhead.

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

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

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 the *macro implementation*. The only fundamental difference with regular method is that macros are resolved at compile time [8]. Def macros offers the programmer the ability to work on the AST representation of the parameters given to the method. The programmer then can inspect, manipulate or completely rewrite the AST, which is compiled by the Scala compiler after the method finishes.

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, and similar to other classes, 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 our language grammar.

- `ValDef(mods: Modifiers, name: Name, tpt: Tree, rhs: Tree)`.
  Value definitions are all definitions of immutable variables val, mutable variable var (identified by the MUTABLE flag), and parameters (identified by the param flag). In Scala, apart from value definitions, ValDef can also contain If statement as also defined in our language grammar.

- `Assign(lhs: Tree, rhs: Tree)`.
  Assign trees are used for non-initial assignments to variables. The lhs typically consists of an id (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.

- `LabelDef(name: Name, params: List[Ident], rhs: Tree)`.
  The LabelDef class in Scala represents the while and do-while statement in our language grammar. 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 {}
```

Listing 3.2: WhileLoop Function

3.3.3 Generating Scala AST

Scala macros is used in this thesis to provide us the AST representation of the method arguments the macro expansion is applied to, which in this case is our workflow pro-

gram. In this section, we present a sample program with iteration and an If statement inside the iteration written in our language (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 3-3, 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 represents 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: the condition, the then part, and the else part. In the LabelDef case, the else part, which is of type Literal only contains an empty constant value, whereas the then part is expanded to another list of statements and expression. Given that in the sample 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.

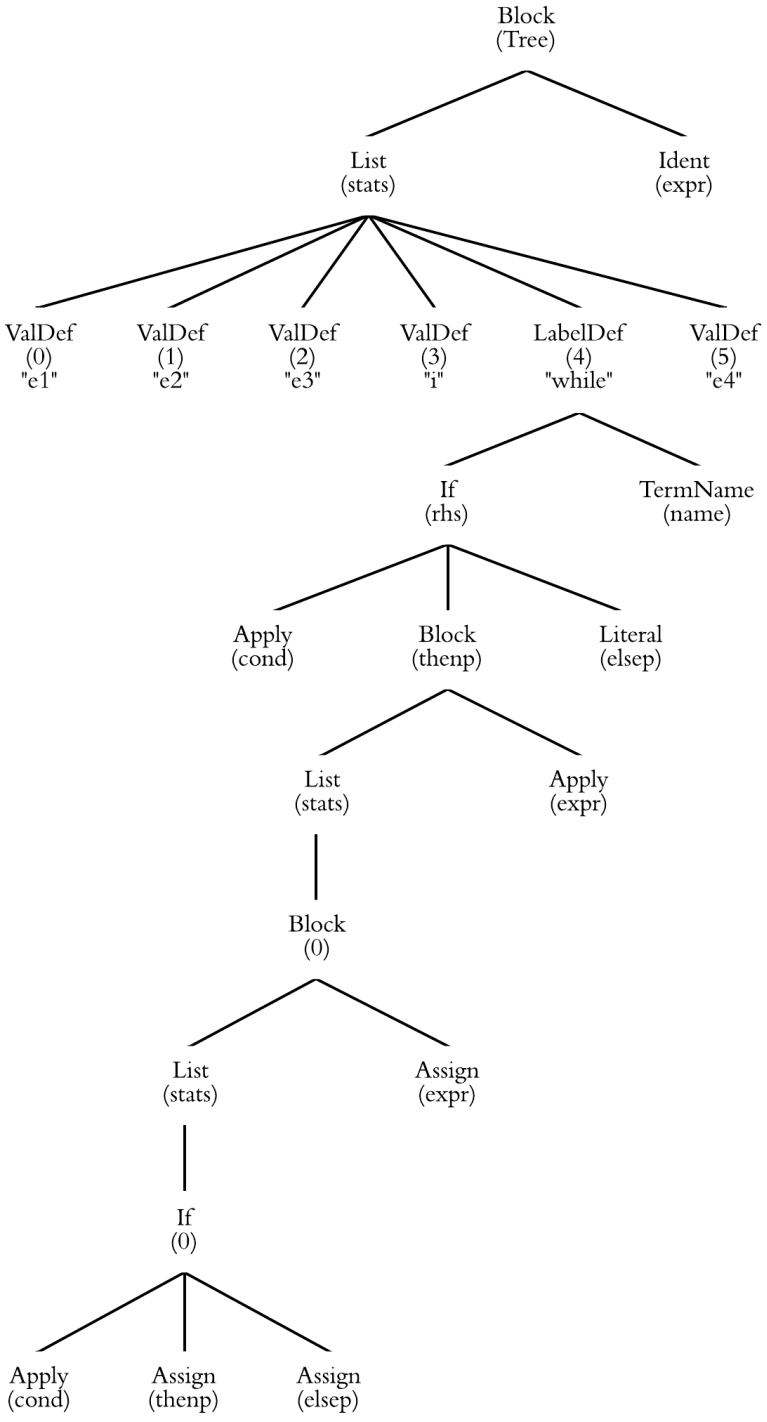

Figure 3-3: Scala AST of Program in Listing 3.3

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 IR in a form of CFG. Here, we formally define CFG as $G = (V, E)$ 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 of the algorithm 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 AST 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 complete 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 AST, 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. Each vertex of the resulted CFG contains a sequence of statements in the AST that can be executed in a sequence (no control flow). In the CFG theory in section 3.1.3, each vertex represents a basic block. 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 $nCurr$, which may be a subtree, does not contain control flow or branches, the $nCurr$ will be pushed to the $vCurr$. Subsequently, if the $nCurr$ 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 variables which contain branches and needs to be updated with the return value of either the then part or the else part of the control flow. In our language grammar, this kind of variables refer to a declaration containing an if statement. In the algorithm, this variable will be reassigned to the return value and will be deleted from the stack and the stack will be empty when the procedure finishes.

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** CREATECFG$(n_{curr}, G, v_{curr}, X)$
3: **match** n_{curr}
4: **case** $Block(stats, expr)$
5: **for** each s in $stats$
6: $(G, v_{curr}, X) \leftarrow$ CREATECFG$(s, G, v_{curr}, \emptyset)$
7: $(G, v_{curr}, X) \leftarrow$ CREATECFG$(expr, G, v_{curr}, X)$
8: **return** (G, v_{curr}, X)
9: **case** $ValDef(name, rhs)$
10: **if** $containsBranches(rhs)$
11: PUSH$(ValDef(name, \emptyset), v_{curr})$
12: PUSH$(name, X)$
13: $(G, v_{curr}, X) \leftarrow$ CREATECFG(rhs, G, v_{curr}, X)
14: **else**
15: PUSH(n_{curr}, v_{curr})
16: **return** (G, v_{curr}, X)

Algorithm 1 Creating Control Flow Diagram from AST Part 2

17: **case** $Assign(name, rhs)$

18: **if** $containsBranches(rhs)$

19: PUSH$(name, X)$

20: $(G, v_{curr}, X) \leftarrow$ CREATECFG(rhs, G, v_{curr}, X)

21: **else**

22: PUSH(n_{curr}, v_{curr})

23: **return** (G, v_{curr}, X)

24: **case** $While(cond, body)$

25: $v_{condStart} \leftarrow$ NEWV()

26: $V \leftarrow V \cup \{v_{condStart}\}; E \leftarrow E \cup \{(v_{curr}, v_{condStart})\}$

27: $(G, v_{condEnd}, X) \leftarrow$ CREATECFG$(cond, G, v_{condStart}, \emptyset)$

28: $v_{bodyStart} \leftarrow$ NEWV()

29: $V \leftarrow V \cup \{v_{bodyStart}\}; E \leftarrow E \cup \{(v_{condEnd}, v_{bodyStart})\}$

30: $(G, v_{bodyEnd}, X) \leftarrow$ CREATECFG$(body, G, v_{bodyStart}, \emptyset)$

31: $v_{curr} \leftarrow$ NEWV()

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)

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)

Algorithm 1 Creating Control Flow Diagram from AST Part 3

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**

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 section 3.3, ValDef refers to definition of a variable both immutable (val) and mutable (var). In this case, we check whether the ValDef contains conditionals branching 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 with the value returned by 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. A sample program that will encounter this case is depicted in the Listing 3.4 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 branching, then the current node $nCurr$ is directly pushed to the current vertex $vCurr$.

- $Assign(name, rhs)$. The procedure for the case Assign is similar with ValDef. We check whether the current node $nCurr$ or subtree contains conditionals branching. If so, the name of the variable is stored in the stack variable X to be assigned later with the value retuned by 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 subtree does not contain branching, the current node $nCurr$ is directly pushed to the current vertex $vCurr$.

- $While(cond, body)$. In the case of while statement, as depicted in CFG expected result (Figure 3-1), a new vertex $vCondStart$ is created to store the condition of the while loop or 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 branching. Hence, recursive call both for *cond* and *body* need to 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 do while case is similar with while case with the differences in the order the vertices are created as well as the placement of 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, which is vertex $vBodyEnd$ is connected to the new vertex $vCondStart$ which stores the condition of the iteration. The end of the condition $vCondEnd$ is then connected to the start of the body $vBodyStart$. Similar to the case of 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 branching. We then create two new vertices $vThenp$ and $vElsep$ to represent the then part and the else part of the If statement, respectively. 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 branching. 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 so, we retrieve the last member inserted to X which is a name of variable (of type TermName in Scala AST), 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 value reassignment is required, the current node $nCurr$ is directly pushed to the current vertex $vCurr$.

We perform this algorithm with the input of the Scala AST of the program in Listing 3.3. The CFG result is shown in Figure 1.4.

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

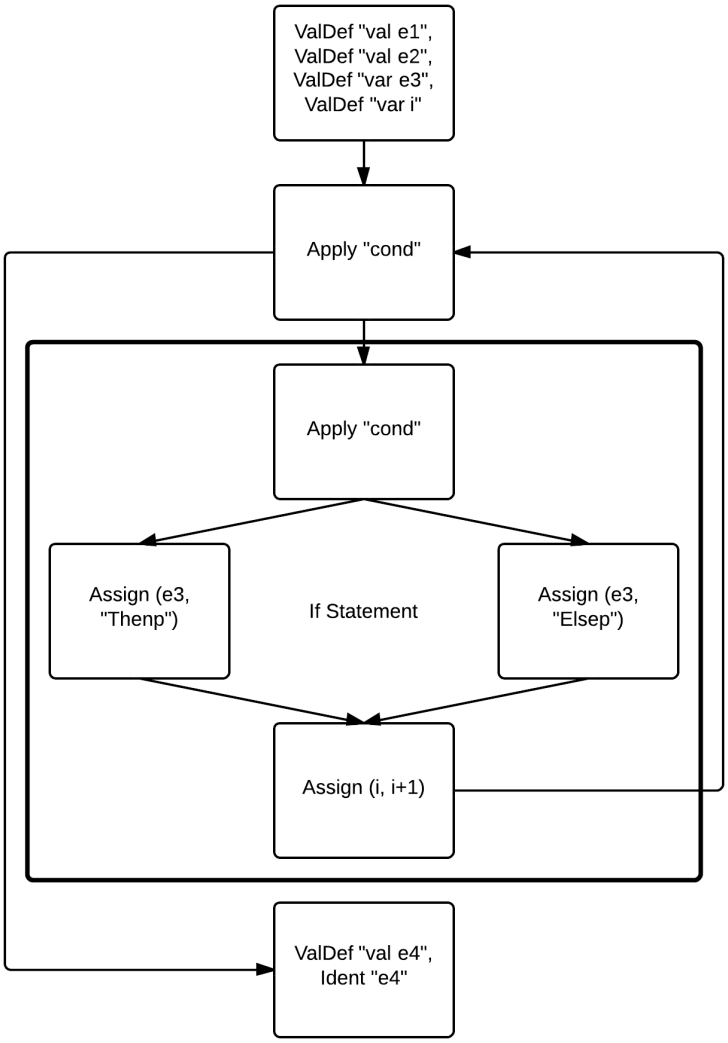

Figure 3-4: CFG of Scala AST of sample Program

vertices 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 one vertex to another.

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

the theory in section 3.1.4). Each vertex in the CFG represents the block that we witness in the theory - we use the term vertex instead of block in this section.

Furthermore, we would like 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) of the sample program in Listing 3.3 resulted from the first stage of the algorithm. The complimented CFG with the Def-Use pair for each vertex is depicted in Figure 3-5.

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 of variables in the program. After we compute the Def-Use pair of each vertex in the CFG, we need 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 graph 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 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 of clear path problem which can be described as follow:

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

A Def-Use pair is formed only if there is a program path on which the value

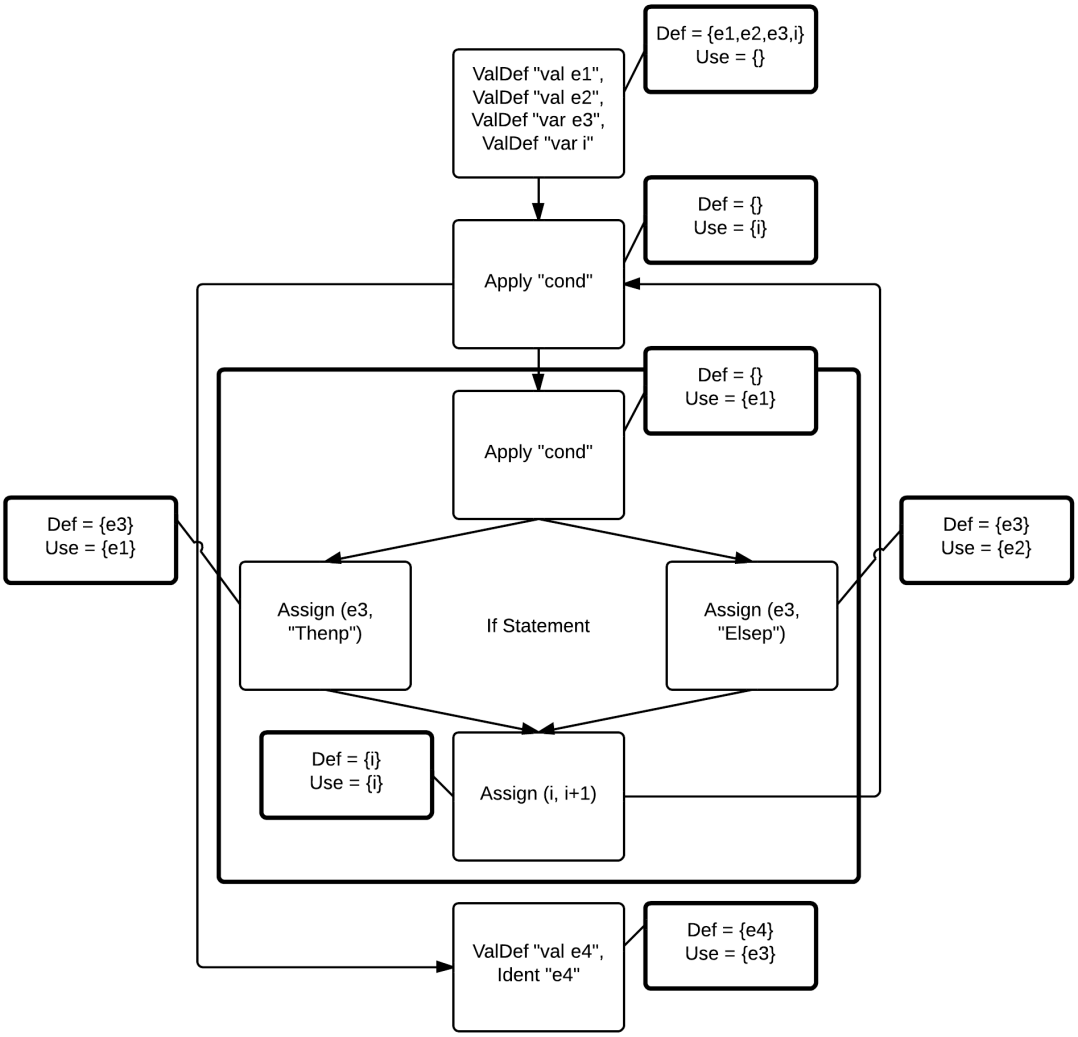

Figure 3-5: CFG with Def-Use Pair Information of Program in Listing 3.3

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 along 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 Control Flow Diagram

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. The result of data flow analysis performed on CFG produced with sample program in Listing 3.3 is depicted in Figure 3-6 below.

3.6 Generate Code for Underlying System

As a result of the previous two algorithms, we have now an IR of the program in the form of a control-flow-enriched data flow graph. The final phase in our compiler model is the code generator. Code generator takes as input the IR produced by the previous stage of the algorithm, and produces as output a semantically equivalent target program for the underlying system.

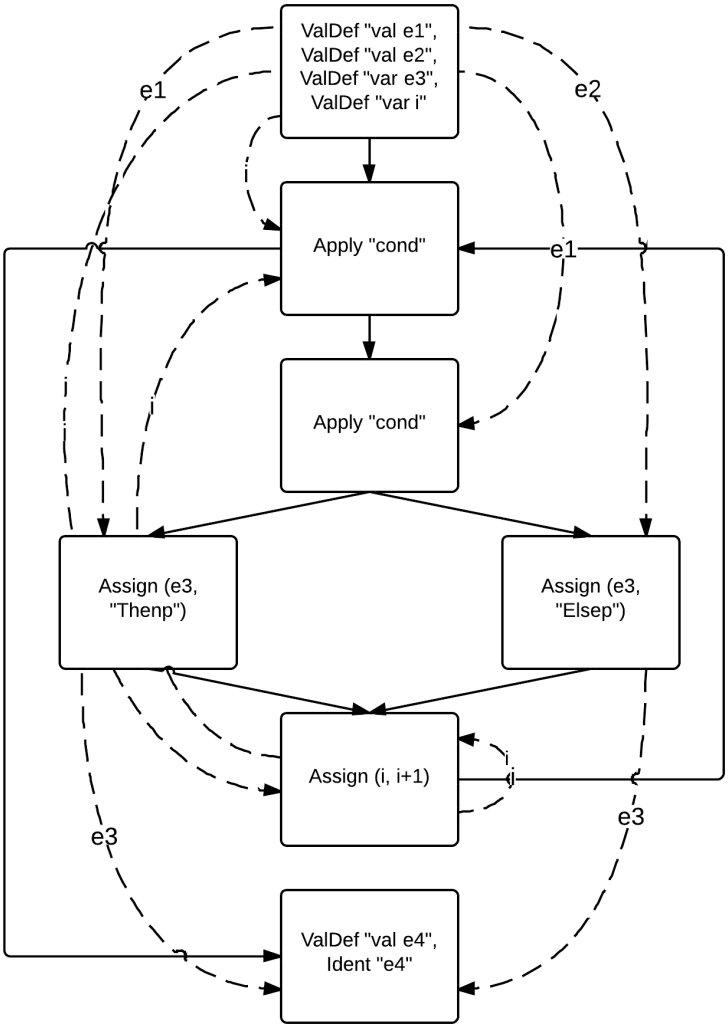

Figure 3-6: Control-Flow-Enriched Data Flow of Program in Listing 3.3

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.

Since we only implement the translation to Stratosphere for now, it is also assumed that the code generated by this algorithm will run only for systems with a specified set of primitives that are currently supported by Stratosphere e.g. map, reduce, join.

That said, it is generally possible to generate the code for any execution framework which is based on working with collections or data sets by providing the functional for that.

3.6.2 Code Generation Algorithm

Each vertex in the graph of our IR represents a basic block which consists of sequences of statements that are always executed together. The objective of this stage is to transform this vertex or block to a Stratosphere job so that in the end of this stage, we have a number of Stratosphere jobs to be run in the WMS with an order according to the dependencies defined in the IR. For each of this Stratosphere job, we will add the necessary environment declarations to enable a driver program in the WMS to execute each job in the underlying infrastructure. The WMS will automatically select which job to be run out of the many Stratosphere jobs since the flow of the execution is already determined by the control flow edge and data flow edge of the IR. The following Figure 3-7 depicts the role of workflow manager inside the underlying system.

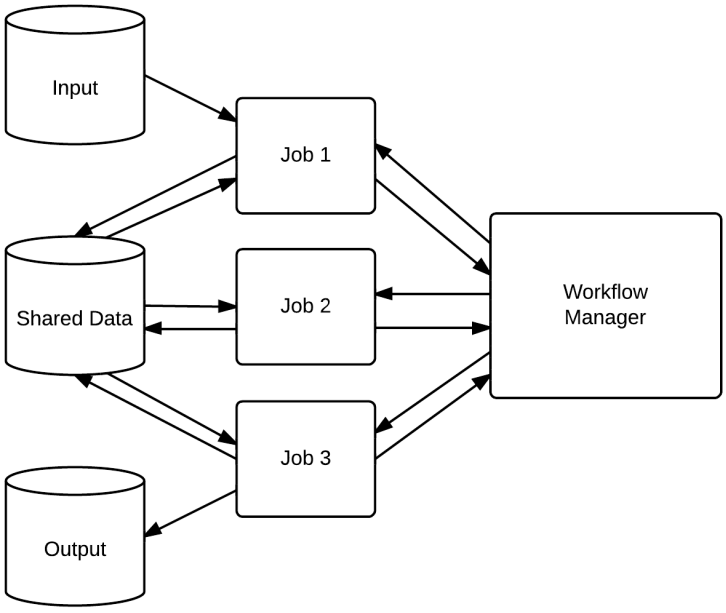

Figure 3-7: Workflow Manager in Underlying System

With regards to the data dependencies, as mentioned in section 3.5, each edge of type data flow is a directed edge which contains 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 contained in the data flow edge to the disk so that other jobs in the workflow can get the data if they use it.

With respect to the output of this algorithm, we denote the sequence of Stratosphere job as J. J is a sequence of Stratosphere jobs denoted by $j(I, O)$ with data source set I consisting of all input variables to the job and data sink set O consisting of all output variables from the job. The input of this final stage is denoted by $G(V, E, DFE)$ as a result of the previous stage of the algorithm. The idea of the algorithm is to traverse the graph and visit each vertex B of the graph. For each vertex B, we will add the required input and output variables v to the data source set I and data sink set O of the job, respectively. At the end, we will need to add a suffix to the job to execute it.

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: $I \leftarrow \emptyset$

5: $O \leftarrow \emptyset$

6: **for** *each variable* v *in* $DFE(\_, B, v)$

7: $I \leftarrow I \cup \{v\}$

8: **for** *each variable* v *in* $DFE(B, \_, v)$

9: $O \leftarrow O \cup \{v\}$

10: $J \leftarrow J \cup \{j(I, O)\}$

11: **return** J

12: **end procedure**

Chapter 4

Evaluation

This chapter addresses the competitive advantages of our workflow DSL. In section 4.1, we select a sample use case for a large-scale data processing. In order to justify that our language increases programmer productivity, we show how to define the workflow in our DSL versus how it is defined in Oozie, the existing workflow system that we explained briefly in section 2.3.2. Section 4.2 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

Data ingestion is the process of obtaining, importing, and processing data for later use or storage in a database. This process often involves altering and validating individual files [2]. The Probes data is delivered to a specific HDFS directory every

[1] http://www.infoq.com/articles/oozieexample
[2] http://whatis.techtarget.com/definition/data-ingestion

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 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.

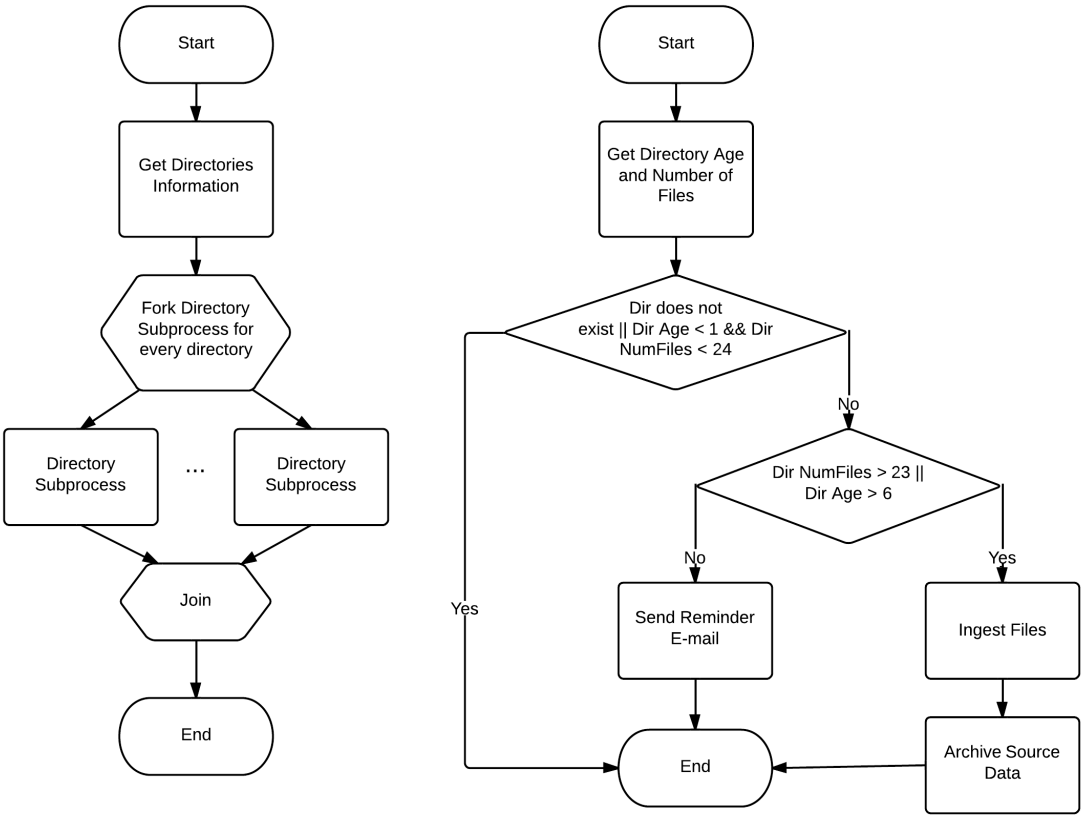

Figure 4-1: Ingestion Process and Directory Subprocess [1]

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 subprocesses 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

The comparison of productivity in terms of lines of code between the implementation of this workflow in Oozie and in our DSL is summarized in Table 4.1 below.

Oozie	Our DSL
191	24

Table 4.1: Productivity Comparison between Oozie and Our 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 data flow 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 IR in a form of control-flow enriched data flow. This IR 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 state-

ment that our DSL grammar support. Spark normally uses For-comprehension for iteration in its programming syntax but we can easily accommodate this to while-statement in our grammar. We show one example of a machine learning algorithm Logistic Regression implemented in Spark [3] 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
 })

 w -= gradient.value
}

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

Listing 4.2: Logistic Regression in Spark

We show that we are able to accommodate 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 implementation into 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

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

Chapter 5

Conclusion

This research is set out to help achieve an important goal that is to build a WMS on top of Stratosphere to support more and more complex large-scale data processing processes. We define a workflow DSL to enable the programmer to easily implement their algorithm without having to manually specify the dependencies e.g. control and data dependencies between tasks in the workflow. The workflow DSL is deeply embedded in Scala high-level programming language, thus avoids overhead for the programmer to learn completely new syntax to write their program. The main contribution of this research is generating a control-flow-enriched data flow and target code from user program via static analysis of the program code. Using Scala macros, an experimental Scala library which expands the Scala AST of the program at compile time, we are able to manipulate the AST of the program to generate the IR that is required in this process. We establish an algorithm for each of the three stages of the translation process as follows: (1) create CFG from AST, (2) adding data flow to the CFG, and (3) generate code for the underlying system. The full implementation of the first stage of the overall process is also covered in this thesis.

To evaluate, we compare our workflow DSL to another existing workflow system for large-scale data processing framework, Oozie workflow system for Hadoop by implementing a complex use case on both systems. It is shown that our workflow DSL is able to increase productivity with far less lines of codes produced compared to the implementation of the same use case in Oozie. Unlike the case with Oozie, our

workflow DSL does not require the programmer to write manually the dependencies between tasks since our compiler frawework will detect these control flow and data dependencies with the technique described previously, thus making the programmer job easier. Another advantage of our workflow DSL is its extensibility to be run on top of other framework such as Spark.

We can always improve our workflow DSL by extending it to also include more grammar definition, for instance, adding a for-comprehension which is currently supported in Spark to our grammar. Another limitation is that our workflow DSL is currently only designed to run on top of Stratosphere. However, in general it is possible to generate the code or job scripts of the workflow for any execution framework as long as we provide the mapping of the Stratosphere primitives to the target system which will be an important future work. For example, it will be a challenge to map a join operator, which is supported by Stratosphere, to Hadoop MapReduce framework which does not provide the join operator. In a more global view, future direction should include the ability to run a program written in our workflow DSL on multiple platforms. For example, a user may want to use Spark streaming for some data analysis and then send the result to Stratosphere to perform a delta iteration. Another important future direction is to utilize the data dependencies information in our workflow for parallelization optimization. For example, we can run multiple jobs in parallel if there are no data dependencies between them, a feature that is already supported by some existing solutions such as Jet.

Appendix A

Oozie Implementation of Ingestion Process

Listing A presents the Oozie workflow implementation of main process of the use case described in Chapter 4.

```
<workflow-app xmlns='uri:oozie:workflow:0.1' name='processDirsWF'>

        <start to='getDirs2Process' />
        <!-- STEP ONE -->
        <action name='getDirs2Process'>
                <!--writes 2 properties: dir.num-files: returns -1 if dir doesn't
  exist,
                        otherwise returns # of files in dir dir.age: returns -1
  if dir doesn't exist,
                        otherwise returns age of dir in days -->
                <java>
                        <job-tracker>${jobTracker}</job-tracker>
                        <name-node>${nameNode}</name-node>
                        <main-class>com.navteq.oozie.GenerateLookupDirs</main-
  class>
                        <capture-output />
                </java>
                <ok to="forkSubWorkflows" />
                <error to="fail" />
        </action>

<fork name="forkSubWorkflows">
       <path start="processDir0"/>
       <path start="processDir1"/>
```

```
        <path start="processDir2"/>
        <path start="processDir3"/>
        <path start="processDir4"/>
        <path start="processDir5"/>
        <path start="processDir6"/>
        <path start="processDir7"/>
</fork>

<action name="processDir0">
 <sub-workflow>
         <app-path>hdfs://sachicn001:8020/user/gtitievs/workflows/ingest</app-path>
         <configuration>
          <property>
              <name>inputDir</name>
<value>hdfs://sachicn001:8020/user/data/probedev/files/${wf:actionData('
  getDirs2Process')['dir0']}</value>
          </property>
          <property>
              <name>outputDir</name>
<value>hdfs://sachicn001:8020/user/gtitievs/probe-output/${wf:actionData('
  getDirs2Process')['dir0']}</value>
          </property>
          <property>
              <name>jobTracker</name>
              <value>${jobTracker}</value>
          </property>
          <property>
              <name>nameNode</name>
              <value>${nameNode}</value>
          </property>
          <property>
              <name>activeDir</name>
              <value>hdfs://sachicn001:8020/user/gtitievs/test-activeDir</value>
          </property>
          <property>
              <name>dirName</name>
              <value>${wf:actionData('getDirs2Process')['dir0']}</value>
          </property>
      </configuration>
  </sub-workflow>
  <ok to="joining"/>
  <error to="fail"/>
 </action>

 <action name="processDir7">
```

```
  <sub-workflow>
         <app-path>hdfs://sachicn001:8020/user/gtitievs/workflows/ingest</app-path>
         <configuration>
           <property>
             <name>inputDir</name>
<value>hdfs://sachicn001:8020/user/data/probedev/files/${wf:actionData('
  getDirs2Process')['dir7']}</value>
           </property>
           <property>
             <name>outputDir</name>
<value>hdfs://sachicn001:8020/user/gtitievs/probe-output/${wf:actionData('
  getDirs2Process')['dir7']}</value>
           </property>
           <property>
             <name>dirName</name>
             <value>${wf:actionData('getDirs2Process')['dir7']}</value>
           </property>
       </configuration>
  </sub-workflow>
  <ok to="joining"/>
  <error to="fail"/>
 </action>

 <join name="joining" to="end"/>

         <kill name="fail">
                  <message>Java failed, error
                         message[${wf:errorMessage(wf:lastErrorNode())}]</message>
         </kill>
         <end name='end' />
</workflow-app>
```

Appendix B

Oozie Implementation of Directory Subprocess

Listing B presents the Oozie workflow implementation of the subprocess of the use case described in Chapter 4.

```
<workflow-app xmlns='uri:oozie:workflow:0.1' name='processDir'>

       <start to='getDirInfo' />
       <!-- STEP ONE -->
       <action name='getDirInfo'>
               <!--writes 2 properties: dir.num-files: returns -1 if dir doesn't
  exist,
                   otherwise returns # of files in dir dir.age: returns -1 if dir
  doesn't exist,
                   otherwise returns age of dir in days -->
               <java>
                   <job-tracker>${jobTracker}</job-tracker>
                   <name-node>${nameNode}</name-node>
                   <main-class>com.navteq.oozie.GetDirInfo</main-class>
                   <arg>${inputDir}</arg>
                   <capture-output />
               </java>
               <ok to="makeIngestDecision" />
               <error to="fail" />
        </action>

       <!-- STEP TWO -->
       <decision name="makeIngestDecision">
               <switch>
```

```
                <!-- empty or doesn't exist -->
                <case to="end">
                      ${wf:actionData('getDirInfo')['dir.num-files'] lt 0 ||
                      (wf:actionData('getDirInfo')['dir.age'] lt 1 and
                      wf:actionData('getDirInfo')['dir.num-files'] lt 24)}
                </case>
                <!-- # of files >= 24 -->
                <case to="ingest">
                      ${wf:actionData('getDirInfo')['dir.num-files'] gt 23 ||
                      wf:actionData('getDirInfo')['dir.age'] gt 6}
                </case>
                <default to="sendEmail"/>
          </switch>
    </decision>

    <!--EMAIL-->
    <action name="sendEmail">
          <java>
                <job-tracker>${jobTracker}</job-tracker>
                <name-node>${nameNode}</name-node>
                <main-class>com.navteq.oozie.StandaloneMailer</main-class>
                <arg>probedata2@navteq.com</arg>
                <arg>gregory.titievsky@navteq.com</arg>
                <arg>${inputDir}</arg>
                <arg>${wf:actionData('getDirInfo')['dir.num-files']}</arg>
                <arg>${wf:actionData('getDirInfo')['dir.age']}</arg>
          </java>
          <ok to="end" />
          <error to="fail" />
    </action>

    <!--INGESTION -->
    <action name="ingest">
          <java>
                <job-tracker>${jobTracker}</job-tracker>
                <name-node>${nameNode}</name-node>
                <prepare>
                      <delete path="${outputDir}" />
                </prepare>
                <configuration>
                      <property>
                            <name>mapred.reduce.tasks</name>
                            <value>300</value>
                      </property>
                </configuration>
```

```
                        <main-class>com.navteq.probedata.drivers.ProbeIngest</main-
    class>
                        <arg>-conf</arg>
                        <arg>action.xml</arg>
                        <arg>${inputDir}</arg>
                        <arg>${outputDir}</arg>
                </java>
                <ok to=" archive-data" />
                <error to="ingest-fail" />
        </action>

        <! Archive Data -->
        <action name="archive-data">
                <fs>
                      <move source='${inputDir}' target='/probe/backup/${dirName}'
    />
                      <delete path = '${inputDir}' />
                </fs>
                <ok to="end" />
                <error to="ingest-fail" />
        </action>

        <kill name="ingest-fail">
                <message>Ingestion failed, error
                     message[${wf:errorMessage(wf:lastErrorNode())}]</message>
        </kill>

        <kill name="fail">
                <message>Java failed, error
                     message[${wf:errorMessage(wf:lastErrorNode())}]</message>
        </kill>
        <end name='end' />
</workflow-app>
```

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.