

Workflow Management System for Stratosphere

by

Suryamita Harindrari

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

Abstract

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

Thesis Advisor: Asterios Katsifodimos

Thesis Supervisor: Volker Markl

Contents

1	Introduction	5
1.1	Related Work	5
1.1.1	Dataflow Systems	5
1.1.2	Workflow Systems	5
1.2	Control Flow vs Dataflow	5
1.2.1	Iteration	5
1.2.2	Control Flow	5
2	Generating Control-Flow-Enriched Data Flows from User Programs	6
2.1	Preliminaries	7
2.1.1	Syntax Directed Translation	7
2.1.2	Abstract Syntax Trees	8
2.1.3	Control-Flow Graphs	10
2.1.4	Data-Flow Analysis	12
2.2	Translating ASTs to CFGs	15
2.3	Scala AST	16
2.3.1	Scala Macros	17
2.3.2	AST Classes	18
2.3.3	Generating Scala AST	19
2.4	Create CFG from AST Algorithm	20
2.5	Generate Control Flow Enriched Data-Flow	25
2.5.1	Generate Def-Use Pair	26
2.5.2	Enrich CFG with Data-Flow	26

List of Figures

2-1	Context-Free Grammar	10
2-2	CFG of Various Statements	11
2-3	Intermediate Representations	16
2-4	Scala AST of Program in Listing 2.3	21
2-5	CFG of Scala AST of sample Program	25
2-6	CFG with Def-Use Pair Information of Program in Listing 2.3	27
2-7	Control Flow enriched Data Flow of Program in Listing 2.3	28

List of Tables

Chapter 1

Introduction

1.1 Related Work

1.1.1 Dataflow Systems

1.1.2 Workflow Systems

1.2 Control Flow vs Dataflow

1.2.1 Iteration

1.2.2 Control Flow

Chapter 2

Generating Control-Flow-Enriched Data Flows from User Programs

This chapter revolves around intermediate code generation part of the compiler in which translation of the source program into target code takes place. In the process of translating a program written in a given language into code for a given target machine, a compiler typically constructs a sequence of intermediate representation which can have a variety of forms [5]. High-level representations are close to the source language and low-level representations are close to the target machine. Syntax tree is one of the most commonly used forms of high-level intermediate representation during syntax and semantic analysis. During intermediate code generation phase of the compiler, another set of intermediate representation i.e. control flow graph is produced. In the preliminaries section of this chapter, we present overview of the theory around syntax trees and graph in order to equip the reader with necessary knowledge to understand how to generate the control-flow enriched data flows from given user programs which we talk about later in this chapter.

2.1 Preliminaries

2.1.1 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 2.1.1). The parser uses the components produced by the lexical analyzer to create a tree-like intermediate representation depicting the grammatical structure of the source program [5]. 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 [5]. Information is associated with the syntax tree by attaching attributes to the grammar symbols representing the programming construct.

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

A context-free grammar in itself specifies a set of terminal symbols (inputs), another set of nonterminals (e.g. symbols representing syntactic context), and a set of productions. Each of these gives way in which strings represented by one nonterminal can be constructed from terminal symbols and strings represented by other nonterminals. Production consists of a head which is the nonterminals to be replaced and a body which is the replacing string of grammar symbols. There are two kinds of attributes for nonterminals, which are Inherited and Synthesized Attributes. The differences between the two are as follows [5]:

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

Definition 2.1.1 (Context-Free Grammars). *A context-free grammar has four components as follow [5]:*

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.*

2.1.2 Abstract Syntax Trees

During syntax-analysis or parsing, the compiler creates syntax-tree nodes to represent significant programming constructs (e.g. operators, classes, control flow etc). This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure that reflects the structure of the program. The purpose of the syntax analysis or parsing phase is to recombine the tokens produced by the lexical analysis as a result of input splitting into a form that reflects the structure of the source program. This form is typically a data structure called the syntax tree or parse tree of the program. As the name indicates, syntax tree is a tree structure [6].

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 [5].

During syntax-directed translation phase in the compiler, a syntax-directed translator is constructed to translate arithmetic expressions into postfix form (see Definition 2.1.2). 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 [6]. 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 [5]. The AST represents an expression formed by applying the operator **op** to the subexpressions represented by E_1 and E_2 . It can be created for any construct, not limited to expressions. Each construct is represented by a node, with children for the semantically meaningful components of the construct. An operator in the abstract syntax is defined for every statement construct. For constructs that begin with a keyword, the keyword is used for the operator (e.g. operator **while** for while-statements).

We define the scope of the language grammar for our thesis as depicted in Figure 2-1 and explain the syntax tree for a language based on this grammar. Syntax-tree *Expr* is used to represent all kinds of expressions, and *stmt* to represent all kinds of statements. Since a block is a grouping of a program, a syntax tree for a program is of type block. When a statement is a block, it has the same syntax tree as the block (hence, $stmt \rightarrow block$). The syntax tree for nonterminal block is simply the syntax tree for the sequence of statements in the block. A sequence of statements is represented by using a leaf **null**. We ignore declarations in our grammar definition since they are not used in the syntax tree. Blocks, with or without declarations, appear to be just another statement construct in intermediate code. Conditionals can be handled by defining two operators **ifelse** and **if** for if-statements with and without an else part, respectively. The syntax tree node for a while-statement and a

do-while statement has an operator, which we call **while** and **dowhile**, respectively and two children - the syntax trees for the *expr* and the *stmt*.

$$\begin{aligned}
\text{program} &\rightarrow \text{block} \\
\text{block} &\rightarrow \{ 'stmts' \}' \\
\text{stmts} &\rightarrow \text{stmts}_1 \text{stmt} \\
\text{stmts} &\rightarrow \epsilon \\
\text{stmt} &\rightarrow \text{expr} \\
\text{stmt} &\rightarrow \mathbf{if}(\text{expr}) \text{stmt}_1 \\
\text{stmt} &\rightarrow \mathbf{if}(\text{expr}) \text{stmt}_1 \mathbf{else} \text{stmt}_2 \\
\text{stmt} &\rightarrow \mathbf{while}(\text{expr}) \text{stmt}_1 \\
\text{stmt} &\rightarrow \mathbf{do}(\text{stmt}_1) \text{expr} \\
\text{stmt} &\rightarrow \text{block}
\end{aligned}$$

Figure 2-1: Context-Free Grammar

Definition 2.1.2 (Postfix Form). *Postfix form for an expression E can be defined as follows [5]:*

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 $E_1 \mathbf{op} E_2$, where \mathbf{op} is any binary operator, then the postfix form for E is $E'_1 E'_2 \mathbf{op}$ 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*

2.1.3 Control-Flow Graphs

The Control-Flow Graph (CFG) is another intermediate representation that is produced during syntax-analysis . Frances E. Allen [1] defines a CFG as "a directed graph in which the nodes represent basic blocks and the edges represent control flow paths". The CFG serves as framework for static analysis of program control flow. Many code generators partition intermediate representation instructions into basic blocks, which consist of sequences of instructions or statements that are always executed together

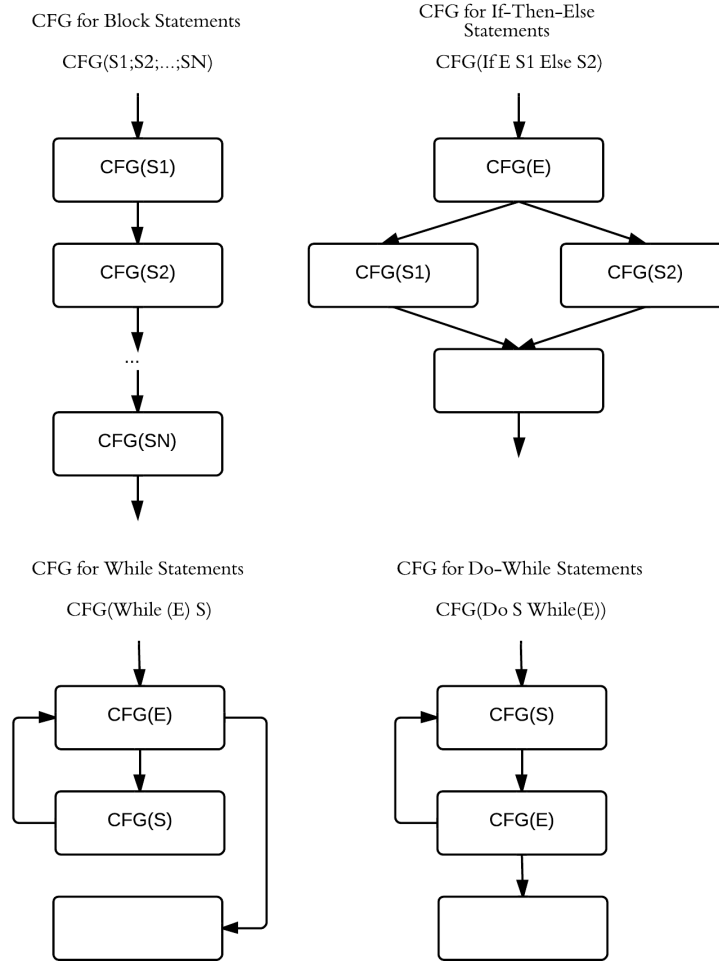


Figure 2-2: CFG of Various Statements

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

Edges represent possible flow of control from the end of one block to the beginning of the other. There may be multiple incoming or outgoing edges for each block [1]. After the intermediate code has been partitioned into basic blocks, the flow of control between them can be represented by a CFG. There is an edge from block A to block B if and only if it is possible for the first statement in block B to immediately follow the last statement in block A. Given a Block statement, If-Then-Else statement, and While-statement, we formulate the expected CFG result from the first stage of the algorithm as shown in Figure 2-2. $CFG(S)$ is a CFG of high-level statement S. $CFG(S)$

is a single entry and single-exit graph with one entry node and one exit node both in the form of a basic block. In the first stage of our algorithm, construction of CFG(S) is recursively defined and will be presented in detailed in the next section.

2.1.4 Data-Flow Analysis

The CFG introduced in the previous section capture one aspect of dependencies among parts of program [9]. Although CFG depicts the control flow of the program, the transmission of information through program variables is missing from the graph. Data flow models resulted from Data-Flow analysis provide a complimentary view, showing relations involving transmission of information [9].

In the second stage of our algorithm, we analyze the graph to detect data dependencies and subsequently add another type of edges which contain information of the data dependencies between the nodes of the CFG. Data dependencies describe the normal situation that the data some statements or basic blocks use depends on the data created by other statements or other basic blocks. Data-flow analysis aims to derive information about the flow of data along with program execution paths. In analyzing the behavior of a program, all the possible sequences of program points ("paths") through a CFG that the program execution can take must be considered. From the possible program states at each point, we extract the information we need for the particular data-flow analysis problem to be solved. The possible execution paths in a CFG can be defined as follows [5].

- 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 [5], *Execution path* or *path* from point p_1 to point p_n is defined as a sequence of points p_1, p_2, \dots, p_n such that for each $i = 1, 2, \dots, n - 1$, 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 [5]. We denote the data-flows values immediately before and immediately after each basic block B in a CFG of a program by $IN[B]$ and $OUT[B]$, respectively. A transfer function f_B relates the data-flow values before and after a block B. Transfer functions can be either the information that propagate forward along the execution paths, or the information that flow backwards up the execution paths. The **data-flow problem** for a CFG is to compute the values of $IN[B]$ and $OUT[B]$ for all blocks B in the CFG [5].

Suppose block B contains of statements s_1, \dots, s_n . If s_1 is the first statement of the basic block B, then $IN[B] = IN[s_1]$. Similarly, if s_n is the last statement of basic block B, then $OUT[B] = OUT[s_n]$. The transfer function of a basic block B, denoted by f_B is derived by composing the transfer functions of the statements in the block. That is, let f_{s_i} be the transfer function of statement s_i . Then $f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$. [5] defines the relationship between the beginning and the end of the block as follow:

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

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

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

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

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

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

Live Variable Analysis

The purpose of Live variable analysis is to know for variable x and point p whether the value of x at p could be used along some path in CFG starting at p . If so, then x is said to be live at p ; otherwise, x is dead at p . The basic motivation of live variable analysis is to manage register allocation. A program contains an unbounded number of variables and must be executed on a machine with bounded number of registers. Two variables can use the same register if they are never in use at the same time (i.e. never simultaneously live). The result of this live-variable analysis is also used to enrich our CFG from the first stage of the algorithm with the data dependencies information. Live-variable analysis is an example of a backward data-flow problem [5].

The point in a program where a value is produced (called a "definition") is associated with the points at which the value may be accessed (called a "use") by the most fundamental class of data flow model [9]. 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 [9].

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

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

These definitions lead to any variable in use_B considered live when entering Block B . Whereas definitions of variables in def_B are dead at the beginning of B . We associate the use and def to the unknowns IN and OUT problem. No variables are live on exit on the program [5]. 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.

$$\begin{aligned} IN[B] &= use_B \cup (OUT[B] - def_B) \\ OUT[B] &= \cup_{S \text{ a successor of } B} IN[S] \end{aligned}$$

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

Reaching Definition

2.2 Translating ASTs to CFGs

We divide the problem of translating the source program to target code into three stages. The first stage is to traverse the given AST and transform it to a more low-level intermediate representation which is Control Flow Graph (CFG); the steps are depicted in Figure 2-3. This thesis covers the design of the algorithm and implementation to transform the AST into CFG. A sample Scala program with control flow and iteration is presented in this chapter to show the process and result (refer to Listing 2.3). The second stage is to analyze the CFG and identify the data dependencies

between each block of the graph. In the end, we generate the executable code for the underlying system. The driver program of the WMS will then execute this code. The design of the algorithm and expected result of the last two stages are also delivered in this thesis.

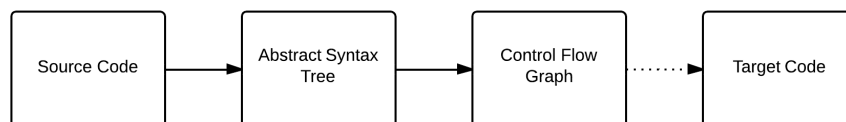


Figure 2-3: Intermediate Representations

2.3 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 [8]. This section gives necessary introduction to Scala AST to allow the reader to understand the transformation process from the source language to the target language. As described in the previous section, AST is one of the most important intermediate representations. The Scala compiler’s parser and type checker provide the Scala AST as intermediate representation that we can directly work on. Additionally, the Scala compiler also provides a tool to traverse and transform an AST [8].

The Scala ASTs are the basis of abstract syntax which is used to represent programs. The Scala compiler uses ASTs as an intermediate representation before generating bytecode [4]. In Scala reflection, Trees can be produced or used by the following APIs¹:

- Scala annotations. This API uses the AST to represent their arguments and is exposed in `Annotation.scalaArgs`.

¹<http://lang.org/overviews/reflection/symbols-trees-types.html>

- **reify**. This special method takes an expression and returns an AST that represent this expression.
- Compile time reflection with macros [2] 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 [2].

2.3.1 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 [3]. With this feature, compiler is enabled to recognize certain methods in Scala programs as metaprograms, or macros, which can be invoked at certain points. When invoked, macros expose a compiler context which includes the compiler's representation of the program being compiled along with an API that provides certain compiler functionality such as parsing and typechecking. Using these API, macros is able to affect compilation e.g. by changing the code being compiled [2].

Defmacros, plain methods whose invocations are expanded during compilation, is the most basic form of compile-time metaprogramming and the one that we work with for this thesis. In the eye of the programmer, def macros appear to look like regular Scala methods with a special property-when a method in a Scala method satisfies this property, that macros definition is expanded by invoking a corresponding metaprogram, called *macroimplementation*. The only fundamental difference with regular method is that macros are resolved at compile time [2]. An example of def macro implementation for **printf** function is depicted in Listing 2.1 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 2.1: Macros Printf Function [2]

2.3.2 AST Classes

This section introduces some of the concrete trees classes that are used in traversing the trees in our implementation. All concrete classes are case classes, thus their parameters are listed following the class name as follows [8].

- **Block(stats: List[Tree], expr: Tree).** A Block consists of a list of statements and returns the value of expr. As the name indicates, this class represents Block in the language grammar.
- **ValDef(mods: Modifiers, name: Name, tpt: Tree, rhs: Tree).** Value definitions are all definitions of vals, vars (identified by the MUTABLE flag) and parameters (identified by the param flag). In Scala, aside from value definitions, ValDef can also contain If statement.
- **LabelDef(name: Name, params: List[Ident], rhs: Tree).** The LabelDef class is used to represent While and Do-While statement. The Scala language specification [7] 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 2.2.

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

Listing 2.2: WhileLoop Function

- **Assign(lhs: Tree, rhs: Tree).** Assign trees are used for non-initial assignments to variables. The lhs typically consists of an Ident(name) and is assigned the value of the rhs which normally contains an application (Apply) of a function.
- **If(cond: Tree, thenp: Tree, elsep: Tree).** An If statement consists of three parts: the condition, the then part and the else part. If the else part is omitted, the literal () of type Unit is generated and the type of the conditional is set to an upper bound of Unit and the type of the then expression, usually Any.

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

2.3.3 Generating Scala AST

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

```

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

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

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

e4

```

Listing 2.3: Workflow with Conditional

As shown in Scala AST in Figure 1.2, the program is represented by a Block which consists of list of statements and an expression which holds the final return value. Each of the variable definition is presented by a ValDef. The LabelDef in the AST represent the While statement in the program and consists of a name and a rhs of type If. The If statement consists of the three parts: condition, then part, and else part. In the while or LabelDef case, the else part which is of type Literal only contains an empty constant value. The then part is expanded to another list of statements

and expression. Given that in the sample Scala program, there is a control flow inside the body of the loop, the statement then consists of another If statement. The then and else part of this If statement are of type Assign since in the program we assign a map function in the rhs to a variable name in the lhs.

2.4 Create CFG from AST Algorithm

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

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

- $nCurr$ refers to the node that is currently being visited in the AST. In the beginning, $nCurr$ is initialized to root of the full AST of the program. Since we traverse the tree from top to bottom, the $nCurr$ becomes the root of the subtree of the initial root node. Depending on the type of the Tree, the $nCurr$ is either pushed to the current Vertex $vCurr$ or the subtree of the $nCurr$ is visited recursively.
- $G = (V, E)$ refers to the CFG produced by the procedure and is continuously being updated whenever recursion takes place. The CFG consists of a set of vertices and a set of directed edges. Each vertex of the resulted CFG is a sequence of statements or nodes in the AST. The vertices set V of the graph has initial member of $vCurr$ whereas the edges set E is initialized to an empty set.
- $vCurr$ refers to the vertex of the CFG that is currently being built. If the $nCurr$

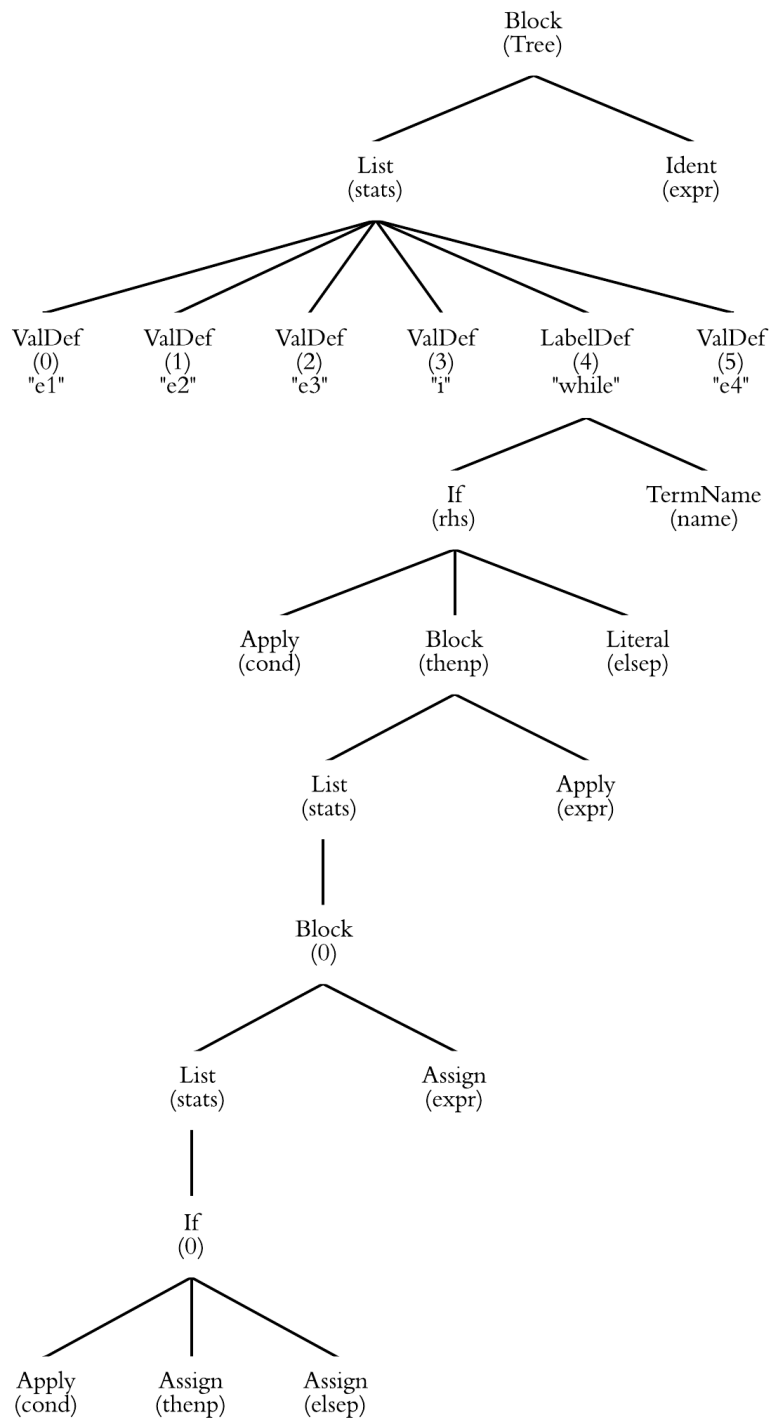


Figure 2-4: Scala AST of Program in Listing 2.3

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

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

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

- $Block(stats, expr)$. A Block typically consists of a list of statements and an expression. For every statement inside the Block, the procedure is called recursively and the Graph $G = (V, E)$, current vertex $vCurr$, and stack of variable X are subsequently updated. After each statement is visited, the procedure is also called recursively for the single expression.
- $ValDef(name, rhs)$ As mentioned in the Scala AST introduction section, ValDef refers to definition of a variable both immutable (val) and mutable (var). In this case, we check whether the ValDef contains branches or control flow. If there is a control flow inside the ValDef, the same ValDef with a mutable identifier is pushed to the current vertex $vCurr$ and the name of the variable of type TermName is stored in the stack variable X . The name of the variable needs to be stored because it must be assigned later on with the value in the Then part or the Else part of the control flow. The rhs then becomes the current

node $nCurr$ and the procedure is called recursively. Sample program that will encounter this case is provided in the Listing 1.3 below. In this case, the variable $e3$ is stored in X and later will be assigned to the function "e1.map .." or function "e2.map ..".

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

Listing 2.4: ValDef with Branches

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

- *Assign(name, rhs)* The procedure for the case Assign is similar with ValDef. We check whether the tree or node contains branches. If it does, the name of the variable is stored in the stack variable X to be assigned later on with the value in the Then part or the Else part of the control flow. The procedure is then called recursively with the rhs as the new current node $nCurr$. Similar to ValDef, if the Tree does not contain branches, the current node $nCurr$ is simply pushed to the current vertex $vCurr$.
- *While(cond, body)* As depicted in Figure 2-3, for While loop, a new vertex $vCondStart$ is created to store the condition of the iteration. A directed edge is defined from the current vertex $vCurr$ to the new vertex $vCondStart$. A new vertex $vBodyStart$ is also created to store the statements in the *body* of the iteration. Both the *cond* and *body* may or may not contain branches. Hence, recursive call both for *cond* and *body* take place which results in the new vertex $vCondEnd$ and $vBodyEnd$ respectively. A directed edge is defined from the end of the *body* which is the vertex $vBodyEnd$ to the beginning of the *cond* which is the vertex $vCondStart$. At the end, a new vertex is created as the new current vertex $vCurr$ and a directed edge is defined from the vertex $vBodyEnd$ to the new current vertex $vCurr$.

- *DoWhile(cond, body)* The procedure for DoWhile case is similar with While case with the differences in the order the vertices are created as well as the directed edges connecting the vertices. The *vBodyStart* is created first and is connected to the *vCurr*. The vertex result of the recursive call to the body of the iteration, vertex *vBodyEnd* is connected to the new vertex *vCondStart* which stores the condition of the iteration. The end of the condition *vCondEnd* is connected to the start of the *body vBodyStart*. Similar to the While loop, at the end, a new vertex is created as the new current vertex *vCurr* and a directed edge is defined from the vertex *vCondEnd* to the new current vertex *vCurr*.
- *If(cond, thenp, elsep)* At first, we create a new vertex *vCond* to store the condition of the If statement. A directed edge is connected from the current vertex *vCurr* to the *vCond*. A recursive call is performed to the procedure since the *cond* may or may not contain branches. We then create two new vertices *vThenp* and *vElsep* to represent the Then part and the Else part of the If statement consecutively. A directed edge is defined from *vCond* to both new vertices *vThenp* and *vElsep* to represent all possible flows. Recursive call is performed both for the Then part *thenp* and Else part *elsep* since they may or may not contain branches. At the end, a new vertex is created as the new current vertex *vCurr* and a directed edge is defined both from the vertex *vThenp* and vertex *vElsep* to the new current vertex *vCurr*.
- *DefaultCase)* We first check whether the stack of variable *X* contains any member. If yes, then we have get the last member inserted to *X* which is a name, and assign the current node *nCurr* to the name. This assignment is then pushed to the current vertex *vCurr*. If the stack variable *X* does not contain any member, which means no assignment is needed, the current node *nCurr* is simply pushed to the current vertex *vCurr*.

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

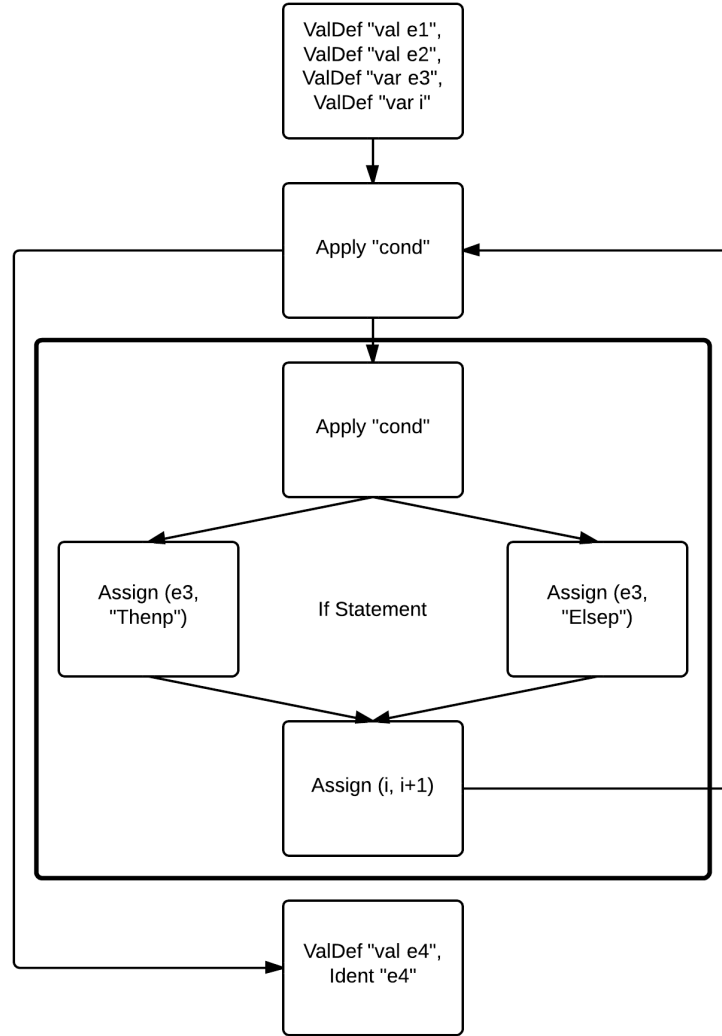


Figure 2-5: CFG of Scala AST of sample Program

2.5 Generate Control Flow Enriched Data-Flow

As described in section 2.1.4, the CFG produced from the first stage of algorithm not yet depicts the transmission of information through program variables. Hence, in this stage of algorithm, we perform data-flow analysis on the CFG with the aim to compliment the existing CFG with the information on the flow of data between each vertex of the CFG. The flow of data is designed as another type of directed edge in the CFG which contains information of the variable that is shared from the one vertex to the vertex.

2.5.1 Generate Def-Use Pair

In order to compliment the CFG with the data flow edges, the first step that we need to perform is to compute the set of variables defined and the set of variables used in each vertex of the CFG, which is denoted by def_B and use_B , respectively (refer to theory in section 2.1.4). Each vertex in the CFG is the same as basic block that we see in the theory - we will use the term vertex instead of basic block in this section.

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

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

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

Using these definitions, we generate the Def-Use pair information for each of the vertex in $G(V,E)$ resulted from the first stage of the algorithm using the sample program in Listing 2.3. The complimented CFG is depicted in Figure 2-6.

2.5.2 Enrich CFG with Data-Flow

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

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

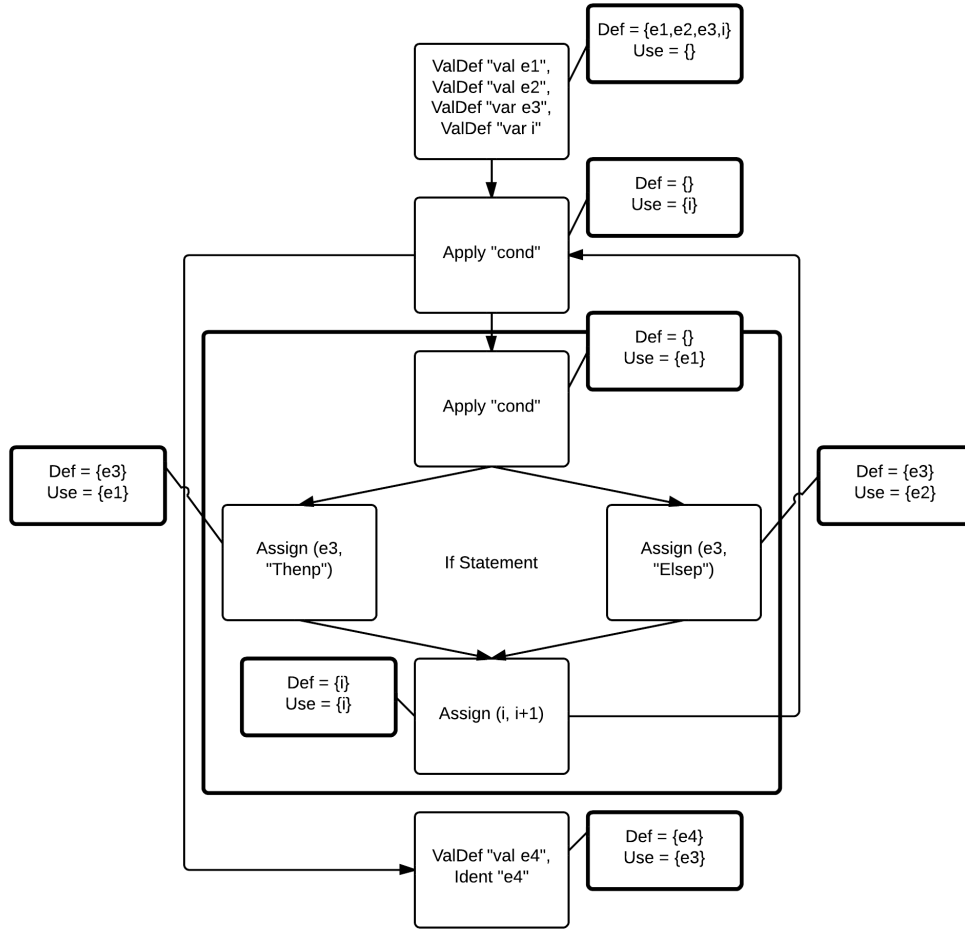


Figure 2-6: CFG with Def-Use Pair Information of Program in Listing 2.3

to the definition clear path problem and can be described as follow:

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

A definition-use pair is formed only if there is a program path on which the value assigned in the definition can reach the point of use without being overwritten by another value. If there is another assignment to the same variable on the path, we say that the first definition is killed by the second. A definition-clear path is a path from definition to use on which the definition is not killed by another definition of the same variable [9]. Hence, the steps to enrich the CFG resulted from previous stage is summarized in the following Algorithm. The input of our algorithm is CFG $G(V,E)$ with set of vertices V and a set of directed edges E as well as computed Def-Use pair

for each vertex. The expected output of our algorithm is Control Flow Enriched Data Flow $G(V,E,DFE)$ with DFE referring to a set of typed-directed edges that depicts the information flow of a variable from one vertex to another vertex. The result of data-flow analysis performed on CFG produced with sample program in Listing 2.3 is depicted in Figure 2-7 below.

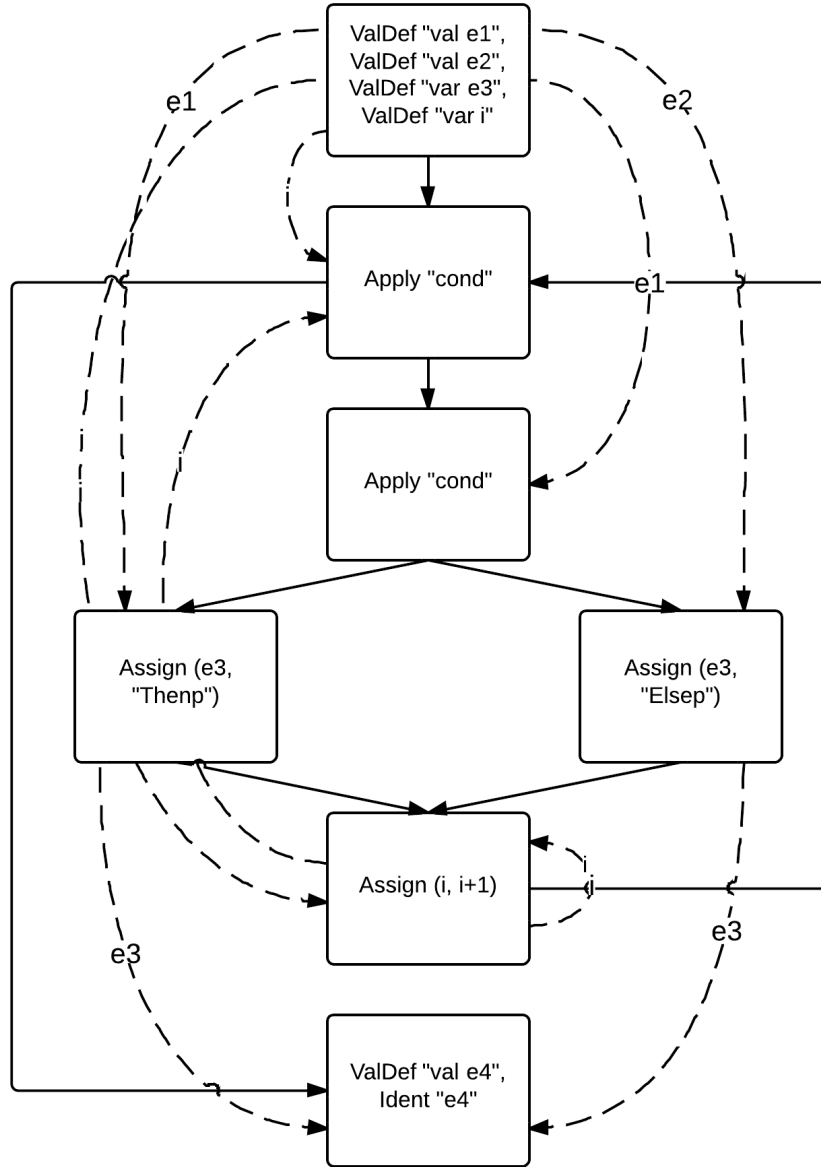


Figure 2-7: Control Flow enriched Data Flow of Program in Listing 2.3

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:          $(G, v_{curr}, X) \leftarrow \text{CREATECFG}(s, G, v_{curr}, \emptyset)$   
7:        $(G, v_{curr}, X) \leftarrow \text{CREATECFG}(expr, G, v_{curr}, X)$   
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:         $(G, v_{curr}, X) \leftarrow \text{CREATECFG}(rhs, G, v_{curr}, X)$   
14:      else  
15:         $PUSH(n_{curr}, v_{curr})$   
16:        return  $(G, v_{curr}, X)$   
17:     case  $Assign(name, rhs)$   
18:       if  $containsBranches(rhs)$   
19:         $PUSH(name, X)$   
20:         $(G, v_{curr}, X) \leftarrow \text{CREATECFG}(rhs, G, v_{curr}, X)$   
21:       else  
22:         $PUSH(n_{curr}, v_{curr})$   
23:        return  $(G, v_{curr}, X)$   
24:     case  $While(cond, body)$   
25:        $v_{condStart} \leftarrow \text{NEWV}()$   
26:        $V \leftarrow V \cup \{v_{condStart}\}; E \leftarrow E \cup \{(v_{curr}, v_{condStart})\}$   
27:        $(G, v_{condEnd}, X) \leftarrow \text{CREATECFG}(cond, G, v_{condStart}, \emptyset)$   
28:        $v_{bodyStart} \leftarrow \text{NEWV}()$   
29:        $V \leftarrow V \cup \{v_{bodyStart}\}; E \leftarrow E \cup \{(v_{condEnd}, v_{bodyStart})\}$   
30:        $(G, v_{bodyEnd}, X) \leftarrow \text{CREATECFG}(body, G, v_{bodyStart}, \emptyset)$   
31:        $v_{curr} \leftarrow \text{NEWV}()$   
32:        $V \leftarrow V \cup \{v_{curr}\}; E \leftarrow E \cup \{(v_{condEnd}, v_{curr}), (v_{bodyEnd}, v_{condStart})\}$   
33:       return  $(G, v_{curr}, X)$ 
```

Algorithm 1 Creating Control Flow Diagram from AST Part 2

```
34:   case DoWhile(cond, body)
35:      $v_{bodyStart} \leftarrow \text{NEWV}()$ 
36:      $V \leftarrow V \cup \{v_{bodyStart}\}; E \leftarrow E \cup \{(v_{curr}, v_{bodyStart})\}$ 
37:      $(G, v_{bodyEnd}, X) \leftarrow \text{CREATECFG}(\textit{body}, G, v_{body}, \emptyset)$ 
38:      $v_{condStart} \leftarrow \text{NEWV}()$ 
39:      $V \leftarrow V \cup \{v_{bodyEnd}, v_{condStart}\}; E \leftarrow E \cup \{(v_{bodyEnd}, v_{condStart})\}$ 
40:      $(G, v_{condEnd}, X) \leftarrow \text{CREATECFG}(\textit{cond}, G, v_{cond}, \emptyset)$ 
41:      $v_{curr} \leftarrow \text{NEWV}()$ 
42:      $V \leftarrow V \cup \{v_{curr}, v_{condEnd}\}$ 
43:      $E \leftarrow E \cup \{(v_{condEnd}, v_{curr}), (v_{condEnd}, v_{bodyStart})\}$ 
44:     return  $(G, v_{curr}, X)$ 
45:   case If(cond, thenp, elsep)
46:      $(G, v_{cond}, X) \leftarrow \text{CREATECFG}(\textit{cond}, G, v_{cond}, \emptyset)$ 
47:      $v_{thenp} \leftarrow \text{NEWV}()$ 
48:      $V \leftarrow V \cup \{v_{thenp}\}; E \leftarrow E \cup \{(v_{curr}, v_{thenp})\}$ 
49:      $(G, v_{thenp}, X) \leftarrow \text{CREATECFG}(\textit{thenp}, G, v_{thenp}, X)$ 
50:      $v_{elsep} \leftarrow \text{NEWV}()$ 
51:      $V \leftarrow V \cup \{v_{elsep}\}; E \leftarrow E \cup \{(v_{curr}, v_{elsep})\}$ 
52:      $(G, v_{elsep}, X) \leftarrow \text{CREATECFG}(\textit{elsep}, G, v_{elsep}, X)$ 
53:      $v_{curr} \leftarrow \text{NEWV}()$ 
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:        $\textit{name} \leftarrow X.\textit{head}$ 
59:        $\text{PUSH}(\textit{Assign}(\textit{name}, n_{curr}), v_{curr})$ 
60:     else
61:        $\text{PUSH}(n_{curr}, v_{curr})$ 
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
```

Algorithm 2 Adding Data Flow to CFG

Input: $G(V, E)$ with $Def - Use$ Pair for each vertex

Output: $G(V, E, DFE)$

1: **Initialize:**

$DFE \leftarrow \emptyset$

2: **procedure** CREATECFDFG(G)

3: **for** each vertex $B1$ in G

4: **for** each vertex $B2$ in G other than $B1$

5: **for** each variable v in def_{B1}

6: **if** There exists $use(B2, v)$ and $def(B1, v)$ reaches $use(B2, v)$

7: $DFE \leftarrow DFE \cup \{(B1, B2, v)\}$

8: **return** G

9: **end procedure**

Bibliography

- [1] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.
- [2] 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.
- [3] Eugene Burmako and Martin Odersky. Scala macros, a technical report. In *Third International Valentin Turchin Workshop on Metacomputation*, number EPFL-CONF-183862. Citeseer, 2012.
- [4] Mathieu Demarne, Adrien Ghosn, and Eugene Burmako. Scala ast persistence. Technical report, 2014.
- [5] Monica Lam, Ravi Sethi, JD Ullman, and Alfred Aho. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.
- [6] Torben Ægidius Mogensen. *Basics of Compiler Design*. Torben Ægidius Mogensen, 2009.
- [7] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.
- [8] Mirko Stocker. *Scala Refactoring*. PhD thesis, HSR Hochschule für Technik Rapperswil, 2010.
- [9] Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.