Implementing Optimization Component for Adela Compiler Computing Dominators

Implementing Optimization Component for Adela Compiler Computing Dominators

By

Dekai Meng

Project Report
Submitted to The Department of Computing and Software
and the School of Graduate Studies
of McMaster University
in Partial Fulfillment of the Requirements
for the Degree
Master of Engineering

TITLE: Implementing Optimization Component for Adela Compiler Computing Dominators

AUTHOR: Dekai Meng,
Computing and Software
McMaster University
Hamilton, ON, Canada

SUPERVISOR: Dr. Frantisek Franek

NUMBER OF PAGES: 43

Thanks Prof. Frantisek Franek

Abstract

Compiler optimization is not typically covered in the undergraduate texts on compiler methodologies and techniques. Professor Franek and his former Ph.D. student Professor Liut addressed this problem in their manuscript **Designing a Modern Compiler with Python** in their Adela project. The code for the whole Adela compiler is in Python and the optimization package also needs to be implemented in Python.
Determining dominators in the CFG (Control Flow Graph) of the program being optimized is important for two aspects of optimization:

- loop optimization – the determination of dominators is necessary to identify loops,
- data flow optimization – that is best performed if the program is in the SSA (Static Single Assignment) form. To modify the code to the SSA form, the dominance frontier must be computed.

In this work we present algorithms for computing the dominators and computing dominance frontier implemented in Python to be incorporated into the optimization package of the Adela project.

Contents

List of Figures

Chapter 1

Introduction

Compiler optimization is not typically covered in the undergraduate texts on compiler methodologies and techniques. Professor Franek and his former Ph.D. student Professor Liut addressed this problem in the Adela project. The results of the project are a language Adela and the draft of a textbook **Designing a Modern Compiler with Python** which details the process of building a compiler for the language Adela. The code for the whole Adela compiler is in implemented in Python 3, and so the optimization package also needs to be implemented in Python 3.

The textbook introduces several different optimizations, usually on the level of the intermediate code. Many depends on CFA (Control Flow Analysis) and DFA (Data Flow analysis) which are covered in the textbook in significant depth. CFA investigates the properties of the CFG. DFA is best performed if the code is in SSA (Static Single Assignment) form.

There are two aspects of optimization that employ the same CFA analysis, in particular the computation of dominators in CFG.

- **Loop optimization** is probably one of the most useful of all compiler optimizations. In the modern high-level symbolic languages such as Python or Adela, a loop is a language construct and as such easy to recognize. For instance, consider a for-loop in Python. But once the source code is transformed to the intermediate code, it is not easy to determine what is a loop – not every jump back identifies a loop. Thus, determining a loop is the first step in the

order to optimize it. The notion of dominators is essential and thus being able to compute all dominators (and what they dominate) in a CFG is necessary in order to identify all the loops in the intermediate code.

- Data Flow Analysis. It is a large topic so we are not going to describe it here, the textbook **Designing a Modern Compiler with Python** is a very good source for it. The DFA can be best carried out if the code is in SSA form. The transformation of the code into SSA form requires to be able to compute dominance frontier on a node of the CFG. Even though there are known more efficient algorithms to compute the dominance frontier, an algorithm can be based on computation of dominators.

We decided for this M. Eng. project to implement three algorithms in Python for the Adela compiler:

- to compute the dominators in recursive fashion domComp1 (see Chapter 4, section 4.1), and
- a different way to compute the dominators in recursive fashion domComp2 (see Chapter 4, section 4.2), and
- to compute the dominance frontier DFron (see Chapter 6)

In order to facilitate proper testing and debugging of this project, we also implemented several supporting programs.

- Generator of random CFG's (generateGraph, see Chapter 4, section 4.3)
- Dominator Verifier (see Chapter 4, section 4.4)
- Dominance Frontier Verifier (see Chapter 6).

Thus domComp1, domComp2, DFron were tested on hundreds of random CFG's as generated by generateGraph. The results were then verified by the Dominator Verifier if they are correct (is a node determined as a dominator a true dominator) and complete (are all dominators identified) for domComp1 anddomComp2, and then by Dominance Frontier Verifier for DFron.

The report is structured in the following way:

- Chapter 2 – introduces and discusses CFG
- Chapter 3 – introduces the basics of Adela project and the optimization
- Chapter 4 – is one of the contributions of this M. Eng. project – description and analysis of generateGraph, Dominator Verifier, and domComp1 and domComp2 programs.
- Chapter 5 – analysis and compares domComp1 and domComp2 programs.
- Chapter 6 – another of the contribution of this M. Emg. project – description and analysis of the algorithm DFron to compute dominance frontier, and a verifier.

Chapter 2

Control Flow Graph (CFG)

The Control Flow Graph (CFG) can be understood as a directed graph, which is the representation graph for compiler applications (ex. Figure 2.1).

```
Program Trivial
1.  read(n);
2.  if (n<0) then
3.      write("negative");
    else
4.      write("positive");
    endif;
5.  switch (n)
      case 1:
6.        write("one");
          break;
      case 2:
7.        write("two");
      case 3:
8.        write("three");
          break;
      default:
9.        write("other");
    endswitch;
end Trivial
```

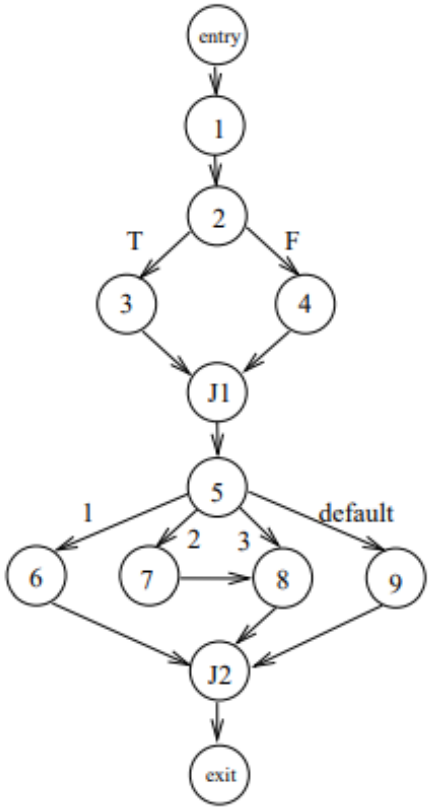

Figure 2.1: Program segment and its CFG [1]

That can be used to depict how the program control is being parsed among the blocks. Control

Flow Graph can help us to easily encapsulate the information per each basic block, and accurately represent the flow inside of a program unit [4].

2.1 Basic Block

The nodes in Control Flow Graph are called the basic blocks.

We can divide the intermediate code of each procedure into basic blocks, and the basic blocks are a piece of straight-line code, and the basic blocks in one procedure are organized as the directed graph called Control Flow Graph [5].

As a direct graph for Control Flow Graph, $G = (N, E)$. Each node $n \in N$ to a basic block. Each edge $e = (n_i, n_j) \in E$ means a possible transfer of control from block n_i to block n_j [6].

Thus, in Control Flow Graph, the vertices represent the basic block and the edges represent the possible transfer of control flow from one basic block to another.

If block A can branch to block B, we can say that block A is an immediate predecessor of block B. If we consider the reverse relationship, we can say block B is an immediate successor of A.

On the other hand, if either A is an immediate predecessor of B, or A is an immediate predecessor of a predecessor of B, we can say block A is a predecessor of B. Similarly with the successor [2]. We will use those relationships in our algorithms later.

2.2 Structure of Control Flow Graph

The Control Flow Graph contains a source node Enter that is connected to every first block, and with a node Exit to which all nodes with no out-edges are connected. For each graph, there is exactly only one node Enter, and one node Exit, and every node has exactly one in-edge and one or more out-edges [2].

2.3 Dominate Relationship and Dominators

The Dominate is a relationship for graph theory in the computer science area. This relationship is reflexive, antisymmetric and transitive [2].

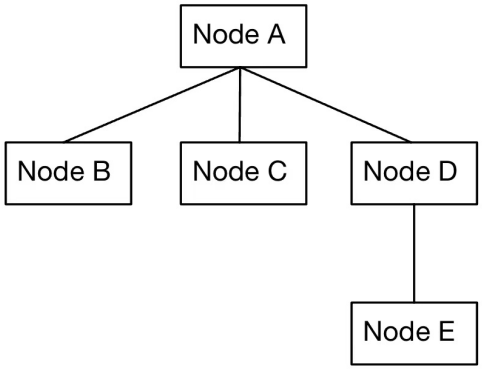

Figure 2.2: Example graph of the Dominate relation

In the control-flow graph, if node A dominates node B, that means every path from the Enter node to B must pass node A. We can write this as A dominate B, $A \gg B$ or $B \ll A$, in which A is the dominator for node B, or node A dominates node B. There are three types of dominant relationships as follows.

2.3.1 Immediately Dominate

Immediately dominant is a local dominance, which means node A immediately dominates node B if there is no intervening node C which is dominated by node A but dominates B such that $A \gg C \gg B$. In addition, there does not exist a node C such that $C \neq A$, $C \neq B$, A dominate C and C dominate B (Figure 2.3). To be specific, A is the first node which dominates B. See in figure 2.2, Node A dominates Node B, C, D, and E, and immediately dominates Node B, C, and D. But Node A does not immediately dominate Node E, because there exists a Node D such that $A \gg D \gg E$. Similarly, Node D dominates and immediately dominates Node E (Figure 2.2).

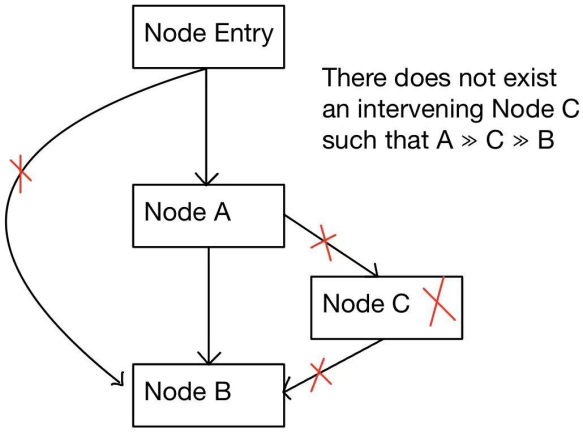

Figure 2.3: Immediately dominate example: Node A immediately dominate Node B

2.3.2 Strictly Dominate

Strictly dominant means in Control Flow Graph, if Node A dominates Node B(A $\gg$ B), and A $\neq$ B, we say that Node A is strictly dominant Node B. In addition, the Node can not strictly dominate itself.

We use $s \ll$ or $\gg s$ to denote strict dominance [2]. For example, in figure 2.2, Node D Strictly dominates the Node E, in which there are no other Nodes N such that D $\gg$ N $\gg$ E.

2.3.3 Postdominate

Node A is postdominate to Node B if every path from B to Exit includes Node A (Figure 2.4). We use $p \ll$ or $\gg p$ to denote the postdominance [2].

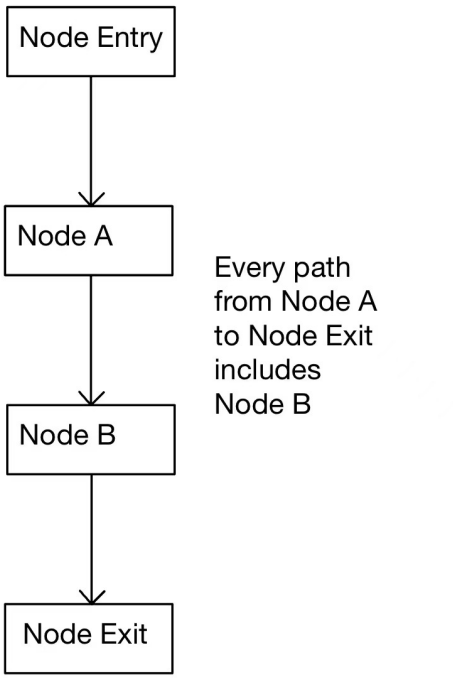

Figure 2.4: Postdominate example: Node A postdominate Node B

Chapter 3

Project Statement and the Background of the Project

3.1 The Adela Compiler

A compiler is a computer program which transforms the source code written in a computer language or the source language into another computer language [7]. This project is based on the Adela Compiler in Advanced Optimization Laboratory. Adela is called ADvol Experimental LAnguage, which is a case-sensitive language, used as the language of choice to demonstrate compiler building. That is designed to be a typical objected-oriented imperative strongly typed language, relatively simple and illustrative from the compilation point of view.

In order to avoid the necessary complexity of a real hardware platform, we chose a similar approach to Java, the compiler will translate the Adela program into assembler-like code called Adela intermediate code, or Aic. Which will be interpreted by the Aic interpreter, Aicint (Figure 3.1) [2].

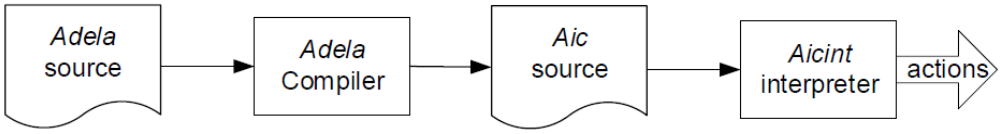

Figure 3.1: Adela Compiler translation [2]

3.2 Control Flow Optimization

The Control Flow Graph and the Basic Block are all belonging to the part of Control Flow Optimization, which can analyze the flow of control inside a method, and reassign the paths of code to improve their efficiency. The optimization included the code corporation(reorder, split, remove...), the Loop, and the exception-directed optimization (IBM). In the textbook, for most of the code translated by ICgen() and peephole optimization, it will be hard to realize there is a loop in the code, which is hard to distinguish. But the flowchart diagram of the given method will be clear. This is the idea behind the control flow graph. We could improve the flowchart by lumping together the instructions that do not affect the flow of control into so-called basic blocks which we discussed previously.

The Adela is quite simple and strict in its calling and return conventions, and thus we do not need to concern about the calls when considering basic blocks. And the control flow analysis is concerned with the optimization of a single procedure, so if the corresponding catch is in the same procedure, we must consider throw a branch there, otherwise, we must consider throw as if it were a return. For some aspects of optimization, we want to consider calls as boundaries for basic blocks [2]. The optimization method is significant to reduce redundancy efficiency.

3.3 Data Flow Optimization

The data flow analysis is concerned with how a program or a smaller unit manipulates its data. And that usually leads to a system of equations that must be solved [2]. It is really imperative for many tasks of optimization and reasoning about data flow to know what could be a value of a variable at a certain point and what it is determined by.

For the analysis of the data flow in the Control Flow Graph, which determines the information regarding the definition and use of data in our program, then the optimization will be completed [8].

For example, consider the code in figure 3.2, as we can see, the initial assignment for $c = 3$ is not necessary and the expression of $x + 1$ can be simplified as 7, but that is not obvious how out compiler can find those expressions by checking the consecutive statements. Thus, the Data flow analysis and optimization are required to find that property, and that can be performed on the Control Flow Graph in our program so that we can determine those parts of the program to which a particular value is assigned to a variable might propagate [9].

```
a = 1;
  b = 2;
  c = 3;
  if (....) x = a + 5;
  else x = b + 4;
  c = x + 1;
```

Figure 3.2: Code example for Data Flow Optimization

3.4 Compute Dominators

A compiler cannot justify the graph and its features, we need to provide a programmatic solution of how to identify loops in the control flow graph, and the memory usage can easily find the leaks and identify the high memory usage by the dominator relation. Thus, computing the dominators is really significant, which can benefit us in the compiler design process and optimization.

To be specific, the properties of the dominance relation can guarantee that it forms a tree, and a dominance tree can be computed for any control flow graph. when a control flow graph has an edge that connects a lower node on the dominance tree with a higher node, it will identify a loop.

In addition, the dominator is significant for the reduce the ϕ-function for the Static Single-assignment Form, which can be regarded as a method for the systematic renaming of variables to

ensure that each variable has only a single definition [2]. When we get the dominance relation, we can construct the dominator tree and dominance frontier, so that we can confirm the insert location for the ϕ-function. if node A defines a certain variable, then that definition and that definition alone (or redefinitions) will reach every node A dominates [7].

Chapter 4

Computing Dominators in a Naïve Way

4.1 The domComp1 Algorithm

4.1.1 Introduction

Given a flow graph, the dominators can be recursively computed by the depth-first search algorithm. In this algorithm, all dominators are sorted inside one two-dimensional array, called Dom. Set the length of Dom to be l1. And the length of one array inside Dom to be l2. Then the dominant relationship could be expressed by the node with an indicator l1 dominating the node with an indicator l2.

At the start of this algorithm, all the items inside the Dom two-dimensional array are set to 0. This algorithm will call the function called CompDom(Graph, Enter, Dom) to start.

The output for this algorithm will be a two-dimensional array with 0 or 1. For example, dom[X] means the list at index X, and we will see this index X as the indicator for the corresponding node in our given control flow graph, called node A. And the list means the dominant relationship for all the nodes inside of the control flow graph. For example, if dom[X][Y] is 1 means node X dominates node Y, and 0 means not. When we print the result, we need to find all the nodes' ID by the indicator, and print their ID instead, but not their indicator anymore.

After testing the code in the various control flow graph with the different numbers of nodes, the

code ran success with a fully correct output in multiple graphs with different numbers of nodes. The output dom is justified and valid.

4.1.2 The Data Types

There are three different types of this algorithm:

The input graph will be a dictionary, and the key and value for the graph will be the node's indicator and children.

For the structure of the Node, the indicator is an integer in ascending order for each node starting from the Enter Node, reflecting the number of each node. And the children is a list of indicators, that reflect all the children for the target node. And the ID will be the node name in case the name will be the string type or irregular numbers. For each key and value inside of our given graph, that will transfer to the Node type.

The Dom means the dominant relationship for each node inside of the graph, which is a two-dimensional array type. If node A dominates node B, then the Dom[A.indicator][B.indicator] will set to 1 from 0.

4.1.3 Pseudocode

```
struct Node{
   indicator: Integer,
   ID: String,
   children: List of Integer,
}
fn CompDom(Graph, Node, Dom){
   // if our node'data been this position of Dom before, return
   If dom[node's data][node's data] is equal to 1:
      then return;
```

```
    // get a set of M which are all immediate predecessors of our
    target node
    let M = get all immediate predecessor
    for mi in element of M{
        // recursive call CompDom for the node's immediate
    predecessor
        CompDom(Graph, mi, Dom);
}
    // get all the nodes from Graph
    let allNodes = all the nodes from Graph;
    for n in elements of allNodes{
        // we must ensure n and node are not the same, or that
    will make Dom incorrect
        If n's data not equal to node's data:
            // we also must ensure the length of node's immediate
    predecessor is at least one, or that will make Dom incorrect
            If length of M is greater than 0:
                // we assume n dominate node
                Dom[n's data][node's data] is 1;
                for mi in elements of M{
                    // update the Dom[n's data][node's data] by
    whether n is also dominate mi
                        Dom[n's data][node's data] is dom[n's
    data][mi's data];
                        // u does not dominate mi, we need to break
    because that means it does not dominate the node
                        If Dom[n's data][node's data] is equal to 0:
```

```
                    Break;
            }
        // if M is None:
else:
            dom[node.data][n.data] is 1;
    }
    // now, we need to calculate all the successors of target node
    let Successors = get all successors for the target node;
    for successor in all elements of Successors{
        if Dom[successor's data][successor's data] is equal to 0:
            // recursive call depth-first traversal
            CompDom(Graph, successor, Dom);
    }
}
```

4.1.4 Challenges for this Algorithm

For this algorithm, we need to pay attention to the looping for the nodeList. We must ensure the node and the node looping for the nodeList are not the same, because the node must dominate the node itself. And also, we need to ensure the node has an immediate predecessor. To be specific, if a node without an immediate predecessor, which means that node is the Enter node, thus this node will dominate all the nodes in the given control flow graph.

4.2 The domComp2 Algorithm

4.2.1 Introduction

This algorithm is an upgrade vision for the domComp1 algorithm with two modifications. Firstly, rather than using the two-dimensional array, the Linked List vision will use the one-dimensional array with a list of linked lists of dominators of the node. Also, the output for the dom will be different from the domComp1 algorithm, which is a linked list of dominators of the node. Secondly, a set of Visit parameters is required, which is a doubly linked list of node nodes which not been visited yet, rather than using a recursive traversal anymore. The Visit is initialized to all nodes.

The output for this algorithm will be a one-dimensional doubly Linked List array with nodes. Each doubly Linked List means the different nodes inside of a given control flow graph, but the representation for the dominant relationship is different from the domComp1 algorithm.

For each node inside of each doubly Linked List, that will represent if that is being dominated by the current position of doubly Linked List or not, their indicator is called 0 or 1. To be specific, let X be one of doubly Linked List in dom, and Y being the Yth nodes inside of doubly Linked List X. If the indicator of Y is 1, that means Node Y dominate Node X.

Thus, for this algorithm, we replace the two-dimensional array with a one-dimensional array, where dom[Node] is a doubly linked list of dominators of Node, which is different from the output of our domComp1 algorithm [2]. If we need to compare the output with our previous algorithm, transfer each doubly linked list structure to a list of integers, and traversal operation is required.

After testing the code, the code ran success with a fully correct output in multiple graphs with different numbers of nodes. The output dom is justified and valid.

4.2.2 The Data Types

There are three different types of this algorithm:

The input graph will be a dictionary, which is the same as the domComp1 algorithm.

But for the node, there are five instances in the class: indicator, children, ID, prev and next. The

indicator is an integer reflecting the number of each node, and the children is a list of indicators, that reflect all the children for the target node. And the ID will be the node name in case the name will be the string type or irregular numbers because we cannot default the id will be the indicator in ascending order from our Enter node in our Control Flow Graph. For the prev and next, that can be understood as a pointer, prev points to the previous node and the next points to the next node. This structure follows by the doubly linked list structure.

For each key and value inside of our given graph, that will transfer to the Node type for our doubly linked list, which has the indicator, children, ID, prev and next.

The Dom means the dominant relationship for each node inside of the graph, which is a list of doubly linked list types. If node A dominates node B, then the corresponding node in the doubly linked list Dom[B linked List] on the index of A will be set to 1 from 0, which is different from the domComp1 algorithm. We can understand the output of the domComp1 algorithm to be a traversal matrix of the domComp2 algorithm.

4.2.3 Pseudocode

```
struct Node{
   indictaor: Integer,
   ID: String,
   Prev: None,
   Next: None,
   Children: List of Integer,
}
struct doublyLinkedList{
   head: None,
   tail: None,
   fn addNode, // add the node to doublyLinkedList
   fn remove, // remove the target node from doublyLinkedList
```

```
   fn findKey, // find is the indicator inside of
   doublyLinkedList
}
fn CompDom(Graph, Node, Dom, Visit){
   While visit is not None{
      // check node been here before
      If dom[Node] exist:
         Return;
      // find the linked list which starts with the node
      let domNodeList = a douby linked list starting with node
   only;
      // the first column of our dom linked list matrix all
   should be dominated by Enter node
      // check if we have the Enter or not, if have Enter, add
   the Enter node inside of domNodeList
      If Enter exist:
         Add Enter node to domNodeList;
      // assign domNodeList to dom[Node]
      Dom[Node] = domNodeList;
      // remove the current node from Visit
      Remove the node from Visit;
      // get the immediate predecessors by linked List structure
   for the current node
      let ImmediatePredecessorList = node's all immediate
   predecessors by linked list;
      // if node does not have immediatePredecessorList, means
   that node is the Enter
```

```
    If current node does not have immediatePredecessorList{
        // update the Enter from None to correct Enter
        Enter = node;
        // add Enter to all the exist linked list in Dom
        For linkedList in dom{
            If linkedList exist:
                Add current node/Enter to the linkedList;
        }
        // because current node does not have immediate
predecessors, which is Enter, so just return
        Return;
    }
    // loop all the current node's immediate predecessors
linked list
    For mi in immediate predecessors linked list{
        // compute all dominators of mi
        CompDom(Graph, mi, Dom, Visit);
        mi = mi.next;
    }
    // loop the intersection of node's dominators, and add
them into Dom[node]
    let intersection = the intersection of nodes from Dom[m1]
to Dom[mk];
    for n in intersection{
        add n into Dom[Node];
    }
}
```

```
}
```

4.2.4 Challenges for this Algorithm

The output of this algorithm is the transpose matrix of the domComp1algorithm, because the dom[node] stores which node dominates the current node, but not the two-dimensional array which is dominated by the node. In other words, the domComp1 algorithm dom[X][Y] means whether Node X dominates Node Y, but the current algorithm is Node X being dominated by Node Y. Each doubly linked list will only store the node which dominate the current node, the undominated node will not be stored in this doubly linked list.

On the other hand, we need a static variable as global storage, which means a variable ENTER inside of our node data structure. And we will assign the Enter node to this ENTER variable. If we already find the ENTER during our looping, then insert this ENTER into nodeFirstList, which the nodeFirstList is a doubly linked list which only has one node.

4.3 The Graph Generator

4.3.1 Introduction

To test the two algorithms above, that will be inconvenient to create various graphs with the different numbers of nodes manually. Thus, the graph generator is required to help us generate a different graph with a given number of nodes. In addition, this graph must also follow the definition of control flow graphs.

Recall the control flow algorithm, each node inside is called a basic block. Each basic block has a single entry and possibly exit to many blocks. We usually add a special source node Entry that connects to the first basic block, in which all arrows go out, no arrow goes in. and a special sink node called exit that connects to all leaves in the graph.

The output will be a control flow graph with the required number of nodes.

4.3.2 The Data Types

The function defined as generateGraph(nodeNum, root = 1, split Rate, connectRate).

The nodeNum is an integer data type, which is the required number of nodes for the graph.

The parameter root is just the root for our graph, which is an integer type.

For the random control flow graph, we need the split probability of the tree structure which is used to generate our control flow graph, called splitRate. splitRate is a decimal data type, we can modify this parameter to different split probability as required.

connectRate is also a decimal data type, that is the connect probability of nodes besides the construction of the tree, which can also help us generate the graph.

4.3.3 Pseudocode

```
fn generateGraph(nodeNum, root = 1, splitRate, connectRate){
   // initialize a graph
let graph is an empty graph;
   let nodeList = []
   for i in range(0, nodeNum - 1){
      if i + 1 is not equal to root:
         append i + 1 to nodeList;
      let graph[i + 1] is an empty list
}
// Define a queue with root
   let fatherList be an empty list;
   // Initialize this queue with root
   Insert root to the position 0 in fatherList;
   // Define a list (leave only) to store our children's nodes
   let childList be a copy of nodeList;
```

```
while childList is not empty{
   let the child be a random choice from childList;
   remove the child from childList;
   // choose a child, and it might be a new father
   Insert child to position 0 in fatherList;
   // get the father by pop method
   let father be the pop value from fatherList;
   // use splitRate to control the number of children of each
node in the tree
   Generate random floating number in range [0.1, 1.0), check
if that number is less than splitRate:
      // out the used father back, so as to own more children
      append father to the fatherList;
   append child to the graph[father];
}
// add extra links besides the tree for the sink
let sink be the nodeNum;
let graph[sink] be an empty list;
let fatherList be a copy of nodeList;
append root to the fatherList;
let probFatherList be an empty list;
while fatherList is not empty{
let father be the pop value from fatherList;
if the length of graph[father] is 0:
   append sink to the graph[father];
else:
   append father to the probFatherList;
```

```
    }
    While probFatherList is not empty{
        let father be a random choice from probFatherList;
        append sink to the graph[father];
        remove father from probFatherList;
        Generate a random floating number in the range [0.1, 1.0),
    and check if that number is greater than connectRate:
            Break;
    }
    return graph;
}
```

4.4 Dominator Verifier

4.4.1 Introduction

In order to justify our output dom from the domComp1 and domComp2 algorithms being correct or not, we need a dominator verifier. The input will be the dom, which is the two-dimensional list and the originally given control flow graph. For the domComp2 algorithm, we need to transfer the output from the one-dimensional list with a set of the doubly linked list to the same as the domComp1 algorithm, which is a two-dimensional integer array.

There are two verifiers are created in this part in total, both of them used breath first search algorithms.

The first dominator verifier will go through each element of the input two-dimensional array dom and use the breadth-first search algorithm to go through the graph to find all the possible paths from the Enter node to the target node. Then, we need to justify our start node always being the first to occur before the target node. For example, for the element dom[x][y], we will use the breadth-first

algorithm to go through the given graph from Enter node to y and find a list of the possible paths. For each path, we will justify is that node x being occurred first than node y, if any of the paths is not, or did not find the start node x from any of our paths, the value of dom[x][y] must be 0, but not 1. Vice versa. If any of our value inside our dom is incorrect, which means our dom is not fully correct, we will not passing this test and a warning will be given by this verifier.

The second dominator verifier will be similar to the above, but create a new two-dimensional array called newDom, which is exactly the same size as our output dom instead. All the values inside of our newDom will be initialized by 0. Then, we will do the same process as previous, using the breadth-first search algorithm to go through all the possible paths from the Enter node to the target node, and find is our required start node always occurs first then the target node for all the paths inside of the list. If so, we will assign the value of 1 to our newDom[start Node][target Node], else that will keep 0 instead. Lastly, we will use this newDom compare with our output dom from our naïve algorithm, if that is exactly the same, means our naïve algorithm output is correct. Else that will give a warning message from our verifier.

After testing both of the naïve algorithms with various nodes of random control flow graphs as input, they have the same relationship for the output dom, after transformation of the domComp2 algorithm to the same format as the domComp1 algorithm, their output is exactly the same and being verified correctly.

4.4.2 The Data Type

There is only two data type in total, one is a graph, which is our input control flow graph. The second is the two-dimensional array dom, which is the output from our naïve algorithm. For the domComp2 algorithm, the transformation from the one-dimensional doubly linked list to the two-dimensional integer array is required.

4.4.3 The Pseudocode

```
fn bfs(start, end, graph){
   // the breadth-first search to find all the paths from start
   to end in our given graph
   // define a queue two-dimensional structure with start first
   let q = [[start]];
   let paths be an empty list;
   let target be the end;
   // loop the queue which we defined until that is empty
   while q is not empty{
      let temp be the pop list from q
      if the last character in temp is not equal to the target:
         append the temp to our paths;
      else:
         // loop the children inside of our graph[last character
   if temp]
         for child in temp:
            // avoid load the nodes which we already visited
            if child in temp:
   }           continue;
            Append temp + [child] to our q;
   return paths;
}
fn testingDom(dom, graph){
   // loop each element in the dom
   for i in range(len(dom)){
      for j in range(len(dom[0])){
         // node itself must dominate itself
```

```
            if i == j:
                if dom[i][j] is 0: return False
                else: continue
            // find all the possible paths by bfs from 1 to j
            let paths be the output from bfs from Enter to j;
            let result be a Boolean to check is our i always occurs
    first then j;
            if dom[i][j] is 1:
                // if dom[i][j] is 1, means the result must be True,
    else means the dom is incorrect
                if result is True: continue;
                else: return False;
            else:
                // if dom[i][j] is 0, means the result must be
    False, else means the dom is incorrect
                if result is False: continue;
                else: return False;
        }
}
// return True if the above steps all passes
return True;
}
fn testingByNewDom(dom, graph){
    let testDom be a new two-dimensional array with all values of
    0, and that is exactly the same size as dom
    // loop each element insider of the testDom
    for i in range(len(testDom)){
```

```
        for j in range(len(testDom[0])){
            // node must dominate node itself
            if i is equal to j:
                if dom[i][j] is not 1: return False;
                else: continue;
            // find all the possible paths by bfs from 1 to j
            let paths be the output from bfs from Enter to j;
            let result be a boolean to check i always occurs first
    before j;
            // if result is True means i should dominate j, else
    means i does not dominate j
            if result is True:
                assign the value of newDom[i][j] to 1;
            else:
                assign the value of newDom[i][j] to 0;
        }
    }
    if all the values in newDom are exactly the same as the
    corresponding value in dom, our dom will be correct;
    else any of value is not the same, means our dom is not
    correct;
}
```

Chapter 5

Comparison Between Two Algorithms

In this chapter, I will summarize the similarities and differences between two naïve algorithms by comparison of their algorithms, data structures and output format.

5.1 Similarities

As we have mentioned previously, both of the algorithm's main ideas are to use the depth search algorithm to find the dominant relationship for each node inside of the given control flow graph, which is an algorithm for traversing or searching a tree structure, or a given graph structures. This algorithm will start at the Enter node of our given control flow graph and explores as far as possible along each branch before backtracking. To be specific, the main idea is to start from a node, then store the current node and move to its adjacent nodes which did not been stored yet, then continue this process until there is NO adjacent node which is not stored. Lastly, the backtrack and check for other nodes and traverse nodes will be processed [10].

Both of the algorithms need to calculate the immediate predecessors of the current looping node. The immediate predecessor means given a control flow graph, we could say that a basic block A is an immediate predecessor of basic block B if A can branch to B (figure 5.1).

In order to calculate the immediate predecessor, we only need to go through the given graph, and justify which node's children contain the target node, if so, that node will be one of the immediate

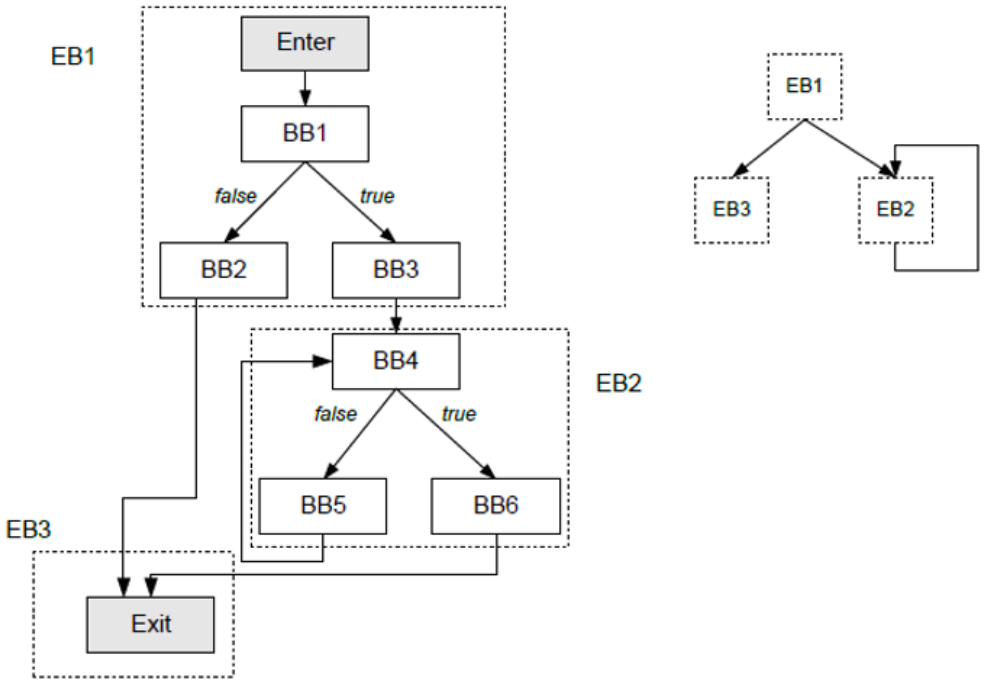

Figure 5.1: Extended blocks and the tree of extended blocks [2]

predecessors for the target node.

```
fn getAllImmediatePredecessors(graph, targetNode){
   let immediatePredecessorList = []
   for each node in the graph{
      if targetNode in the node's children:
         append the node to immediatePredecessorList
   }
   Return immediatePredecessorList
}
```

5.2 Differences

For the Differences, the data structures for the two naïve algorithms are not the same. The domComp1 algorithm only used the nodes, but the second used both the nodes and the doubly linked list structure.

The output of the domComp1 algorithm and domComp2 algorithm is a two-dimensional array with all element integers 0 and 1 (figure 5.2), and a one-dimensional array with a set of the doubly linked list respectively (figure 5.3).

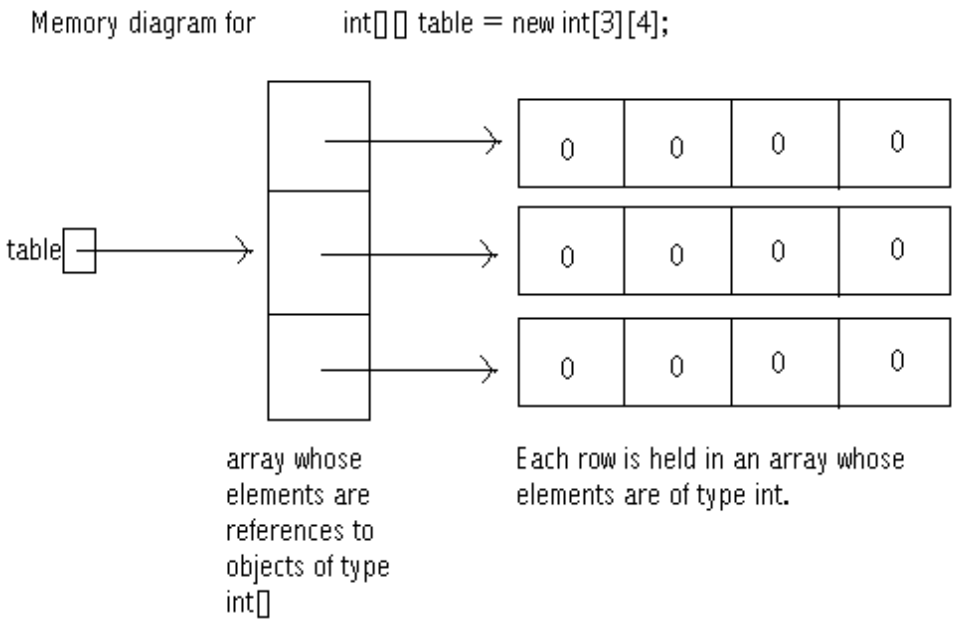

Figure 5.2: Two-Dimensional Array [3]

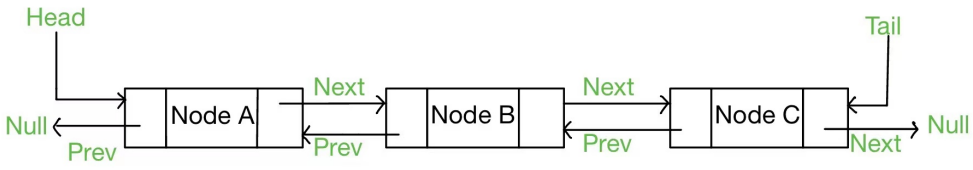

Figure 5.3: Douby Linked List

On the other hand, as we mentioned previously, we need to avoid revisiting our node twice to achieve the depth-first search correctly. In the domComp1 algorithm, we used the dom[Node][Node] to justify did we visit this node before. If so, we will simply return our function to end the algorithm. But in the domComp2 algorithm, we used a doubly linked list called visit, which contained all the nodes inside our given control flow graph. We will do the depth-first search algorithm until the visit is empty and removing the nodes which have been visited already is required for each looping node. In addition, we will also return and end the function under the following conditions:

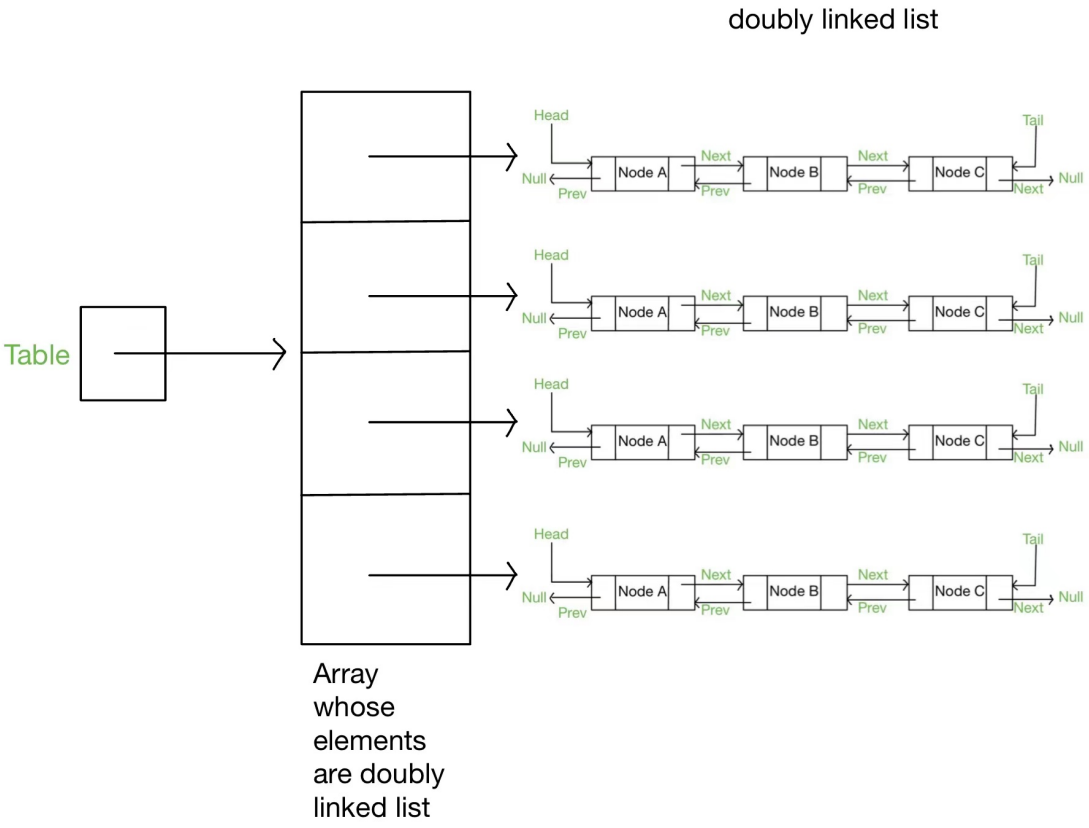

Figure 5.4: One-Dimensional Doubly Linked List

1. The doubly linked list is empty as we mentioned before, which means we had already gone through all the nodes inside of our control flow graph.

2. If the Dom is 1, which means we had been here before.

3. If the Node is the Enter node, we will end our function because there is no predecessor anymore.

For the output of the two algorithms, the data structure of dom is not the same as we mentioned in Chapter 4. For the domComp1 algorithm output dom, the data structure is a two-dimensional array, each value will be integer 0 or 1, which means whether the node which corresponds to the row index in the array dominates the node which corresponds to the column index in the array. For the domComp2 algorithm, the one-dimensional array with a set of doubly linked lists means whether the current index of nodes is dominated by which node is inside its doubly linked list. In other words,

the dom[] where dom[Node] is a linked list of dominators of Node [2].

Chapter 6

Dominance Frontier

6.1 Description

When we construct the Static Single-assignment Form (SSA Form), we need to compute the Φ function to the right position. The dominance frontier ensures the right position for the insertion of Φ-function. If a certain variable is defined by a Node N, then that definition and that definition alone will reach every node which the Node N dominates.

We only need to account for other flows bringing in other definitions of the same variable when leaving these nodes and entering the dominance frontier.

The Dominance Frontier of a basic block B in Control Flow Graph means a set of all basic blocks C which are satisfied the condition as follows:

- B dominates an immediate predecessor of C
- B does not strictly dominate C
- B $\neq$ C

In a more mathematical formulation:

$$Df(\mathrm{B}) = \{\mathrm{C} \mid (\exists \mathrm{D} \in Pred(\mathrm{C}))\,(\mathrm{B} \gg \mathrm{D}\ \&\ \mathrm{B} \not\gg_s \mathrm{C})\}$$

Which the $Df(\mathrm{B})$ is the dominance frontier of B, and the $Pred(\mathrm{C})$ means a set of immediate predecessors of C [2].

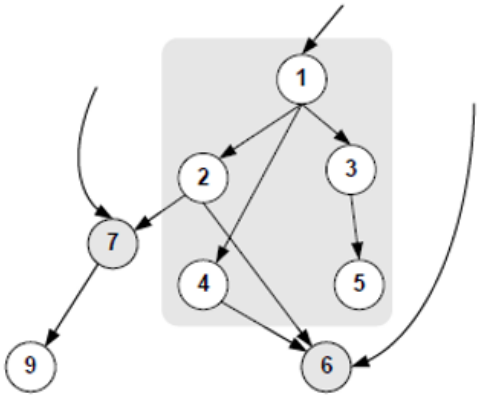

Figure 6.1: Dominance Frontier of node 1 (gray nodes) [2]

In figure 6.1, the dominance frontier for node 1 is Nodes [7, 6], which satisfied all the dominance frontier conditions which we mentioned before.

For node 7, the immediate predecessor is Node 2, which is dominated by Node 1. And there is still another path to Node 7 rather than Node 1 only, which means Node 1 does not strictly dominate Node 7, and obviously Node 1 $\neq$ Node 7. Thus, Node 7 is a dominance frontier of Node 1.

Similar to Node 7, for Node 6, Node 1 dominates the immediate predecessor(Node 2) of Node 6, Node 1 does not strictly dominate Node 6, and Node 1 $\neq$ Node 6. Thus, Node 6 is also a dominance frontier of Node 1.

6.2 Computing the Dominance Frontier

To compute the dominance frontier, two auxiliary sets are need if we want to efficiently compute the dominance frontiers of all the nodes of our control flow graph in one pass. The Df_{local} and the Df_{up}.

6.2.1 The $Df_{local}(\mathrm{B})$

The $Df_{local}(\mathrm{B})$ is the successors of n that are not strictly dominated by B. To compute the $Df_{local}(\mathrm{B})$ easily, we need to consider this function as the set of those successors of n whose immediate dominator is not B.

$$Df_{local}(\mathrm{B}) = \{\mathrm{C} \in Succ(\mathrm{B}) \mid \mathrm{B} \not\gg_s \mathrm{C}\}$$

6.2.2 The $Df_{up}(\mathrm{B})$

The $Df_{up}(\mathrm{B})$ denotes nodes in the dominance frontier of n that are not dominated by B's immediate dominator [2].

$$Df_{up}(\mathrm{B}) = \{\mathrm{C} \in Df(\mathrm{B}) \mid (\forall \mathrm{D} \gg_i \mathrm{B})\,(\mathrm{D} \not\gg_i \mathrm{C})\}$$

6.2.3 The DFron Algorithm Pseudocode

This algorithm will call on the Enter node of the given control flow graph. That will go through the Control Flow Graph computing $Df(\mathrm{B})$ for every node B by computing the $Df_{local}(\mathrm{B})$ and $Df_{up}(\mathrm{B})$ which we discussed above. Then combines both to get our final output. This algorithm will work in linear time approximately, which proportional to the number of edges in our Control Flow Graph, with the size of the dominance frontiers it computes [11].

$$Df(\mathrm{B}) = DF_{local}(\mathrm{B}) \cup \bigcup \{Df_{up}(\mathrm{C}) \mid \mathrm{C} \in Idom(\mathrm{B})\}$$

The $Idom(\mathrm{B})$ means the set of nodes immediately dominated by B. So that the computation of the dominance frontier is just in terms of immediate dominators and immediate successors.

6.3 The Data Types

The DFron algorithm will have three types of parameters in total.

The Graph is the same type as our previous domComp, which is a dictionary type to express the input control flow graph. The key and value for the graph will be the node's indicator and a list of children. The input Node will be the Enter node for our control flow graph.

The Visit is the set type, which will avoid infinity recursion for some of the corner control flow graphs in our algorithm. We will return W if our current Node is already inside our Visit.

The W will be initialized as an empty list, all the dominance frontiers will be stored inside of this list. That is the output for our algorithm.

6.4 Pseudocode

```
struct Node{
   indicator: Integer,
   ID: String,
   children: List of Integer,
}
fn DFron(Graph, Node, Dom, let Visit = empty set, let W = empty
   list){
   // we will return W if the node is in visit
   if node in Visit: return W;
   add W to the Visit
   // Step for computing the Df_local(B) which we mentioned above
   let immediateSuccessor be all the immediate Successors for
   current Node;
   for each node successor in immediateSuccessor:
      if B is not an immediate dominator of successor:
         add successor to the list W;
   for each child of B in the Control Flow graph:
```

```
        let S = DFron (Graph, child, Dom, Visit, W);
            // then we will compute the Df_up(B)
            for every nodeD in S:
                // Combine Df_local(B) and Df_up(B) to get our final
    output W
                if B does not dominate nodeD:
                    add nodeD to the list W;

    return W;
}
```

A call to the DFron (Enter Node) will calculate all the dominance frontiers for all basic blocks, so we do not need to loop all the nodes in our input Control Flow Graph. The parameters W and Visit will be an empty list and set respectively. If we need to avoid duplicated nodes occurring in our output W, a new condition could be added at the update W step.

6.5 Dominance Frontier Vertifier

6.5.1 Introduction

Recall that we call a set of basic block C a dominance frontier of basic block B if B dominance is an immediate predecessor of block C, and B does not strictly dominate block C with Node B $\neq$ Node C. Thus, to verify our output for our algorithm, which is all the dominance frontier of all the blocks in our Control Flow Graph. Two loops are needed to justify our final output. The first loop is our target node, and the second loop will justify is that a dominance frontier or not. Thus, if we consider the node in the first loop called Node B, and the node in the second loop called Node C. Then we need an if statement to justify if Node B strictly dominates Node C and Node B $\neq$ Node C, because by our definition for dominance frontier, that cannot be dominated by our target Node. Next, we

second condition will be verified, whether Node B dominates one of the immediate predecessors or not. Accordingly, we need to get all the immediate predecessors of our Node C and loop all the immediate predecessors to check if Node B dominates the current predecessor. In addition, we must also ensure that the current predecessor is not Node B, because a Node must dominate the Node itself, which will cause incorrect output. If all the above conditions are satisfied, that means Node C will be one of the dominance frontiers for Node B, then added that to our result list. And we only need to compare the result list with our DFron output W. The graph generation function could also use our previous generateGraph which we discussed in chapter 4, in which a random Control Flow Graph with request node will be created by that program. W is the output from our DFron algorithm.

6.5.2 The Data Types

There are three different types of this algorithm:

The input W will be a list type, which is the output of our DFron function, representing all the dominance frontier for all the blocks in our given control flow graph.

The input graph will be the same as our previous control flow graph to calculate the dominator, which is a dictionary type.

This algorithm will return a Boolean to justify whether our algorithm output W is correct or not.

6.5.3 Pseudocode

```
fn justifyDF(W, graph){
   let resultList = [];
   // loop the target node
   for node1 in all Nodes in graph:
      // loop the other node to check is that a dominance
   frontier or not
      for node2 in all Nodes in graph:
```

```
        // check node1 ≠ node2 and node1 does not strictly
   dominate node2
        if node1 ≠ node2 and node1 dominate node2:
            // check node1 dominate one of a predecessor of node2
                let allPredecessor = all the immediate
   predecessors of node1;
                for predecessor in allPredecessor:
                    if node1 ≠ predecessor and node1 dominate
   predecessor:
                        add node2 to resultList;
                        // we will end the current loop because
   node 2 is satisfied dominance frontier and added to the
   resultList
                        end current loop;
return boolean of resultList is equal to W;
}
```

Chapter 7

Conclusion

In this report, we discussed the definition and component for our Adela Compiler and Control Flow Graph, the Dominate relationship and Dominance Frontier. For computing the dominators, we came up with two naïve algorithms in total, called domComp1 and domComp2. For domComp1, our output will be a two-dimensional array to express the dominators for each block. And domComp2 is an improvement version of domComp1, in which the output will be a list of doubly linked lists, which is different from domComp1. For testing both of our two algorithms, the conversion of domComp2 output(a list of doubly linked lists) to the same with domComp1(two-dimensional array) is required, and a function to generate a random Control Flow Graph with certainly required nodes will be used.

On the other hand, we introduced the dominance frontier with the algorithm called DFron to calculate the dominance frontier for all the blocks in a given Control Flow Graph. That algorithm will go through all the nodes by calling the Enter node and return a list of nodes which are the dominance frontier for all blocks. The graph generation could also be used to justify our output for the dominance frontier.

This report computed the dominator and dominator frontier in a naïve way, which is a component that helps us build the dominator for our Adela Compiler, and that is widely used for compiler optimization area.

Bibliography

[1] M. J. Harrold, G. Rothermel, and A. Orso. Representation and analysis of software. https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/rep-analysis-soft.pdf, n.d. Accessed: 2022-07-26.

[2] F. Franek and M. Liut. Designing a modern compiler with Python, n.d.

[3] Harrington. Introduction to multi-dimensional arrays. http://anh.cs.luc.edu/170/mynotes/2dArrays/ar2.html, n.d. Accessed: 2022-07-27.

[4] pp_pankaj. Software engineering: Control flow graph (CFG). https://www.geeksforgeeks.org/software-engineering-control-flow-graph-cfg/, 2019. Accessed: 2022-07-26.

[5] C. Collberg. Compilers and systems software 15: Intermediate code III. https://www2.cs.arizona.edu/~collberg/Teaching/453/2009/Handouts/Handout-15.pdf, n.d. Accessed: 2022-07-26.

[6] K. D. Cooper and L. Torczon. Control-flow graph. https://www.sciencedirect.com/topics/computer-science/control-flow-graph, n.d. Accessed: 2022-07-26.

[7] Wikipedians. Compiler construction. https://books.google.ca/books?id=nMZnyp_zW8AC&pg=PA588&dq=Wikipedians.+(n.d.).+Compiler+Construction.&hl=zh-CN&sa=X&ved=

2ahUKEwiTiInDoZz5AhUSoFsKHX-nD3kQ6AF6BAgJEAI#v=onepage&q=Wikipedians.%20(n.d.).%20Compiler%20Construction.&f=false., n.d. Accessed: 2022-07-28.

[8] Ankit87. Data flow analysis in compiler. https://www.geeksforgeeks.org/data-flow-analysis-compiler/, 2022. Accessed: 2022-07-28.

[9] Javatpoint. Global data flow analysis. https://www.javatpoint.com/global-data-flow-analysis, n.d. Accessed: 2022-07-28.

[10] GeeksforGeeks. Depth first search or DFS for a graph. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/, 2022. Accessed: 2022-07-27.

[11] Converting to SSA form. http://underpop.online.fr/j/java/help/converting-to-ssa-form-compiler-java-programming-language.html.gz, n.d. Accessed: 2022-08-09.

[12] IBM. Phase 3 - control flow optimizations. https://www.ibm.com/docs/en/ztpf/1.1.0.14?topic=code-phase-control-flow-optimizations, 2021. Accessed: 2022-07-28.