Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

Applying slicing algorithms on large code bases

Tibor Brunner
doktorandusz

Olivér Hechtl
programtervező informatikus MSc

Budapest, 2017

Contents

Chapter 1

Introduction

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Brian Kernighan

Nowadays, there are a lot of tools available for debugging. A developer can examine call stacks, variable informations, and so on. There are a lot of times when a bug is when some variable does not behave like the writer of that code part would expect. When the programmer discovers it, he must follow back the path of assignments, and find out where did it get an unexpected value. Program slicing targets this kind of debugging. We can define a program slice as a part of the program which contains a selected statement, and all other statements which influences it, or gets influenced by it. These two types are respectively called backward and forward slicing. Basically this is what many programmers do intuitively, when facing with bugs. In this thesis, I'll introduce a few approaches for computing slices, and describe a working prototype tool application, which can analyze C++ programs, and compute slices of them. I've used the help of the Clang/LLVM compiler infrastructure, which builds the AST of the program and provides an interface to analyse it using C++.

Chapter 2

Slicing, methods and efficiency

2.1 About slicing

For better understanding programs, programmers organize code into structures. They write sub-problems into functions, and organize variables and data to structs. Also, with object oriented design, they put these into classes. These are all good for separating the data, and the procedures on data. But these are not helpful when we need to examine a flow of data in the program. Slicing gets useful in this scenario. This is a program analysis technique introduced by Mark Weiser[1]. In his paper, he wrote: “Program slicing is a decomposition based on data flow and control flow analysis”. We define slicing as a subset of the program, which only includes the statements which have transitive control or data dependency regarding the selected statement.

2.2 Types of slicing

There is two different type of slicing known: static and dynamic. While dynamic slicing gets the statements which could affect the selected statement at a particular execution of the program with a fixed input, static slicing examines it statically, including all possible statements which could affect that selected statement. In this thesis, I’ll focus on static slicing methods. There are two different subtypes of static slicing, backward and forward. They are indicating the relevant statements’ direction from our selected statement.

2.3 Methods for slicing

We can construct slices via various methods on different representations of the program. All of these are using some kind of graph structures, which can be traversed through for searching the transitive data dependences.

2.3.1 Dependences

Before I describe the various methods currently available for slicing, we must get to know what dependences in programs are. There are two kinds of dependences: control and data. They can be defined by the control flow graph (CFG) of the program. Given the following example:

```
int main(){
  int sum = 0;
  int i = sum;
  while (i < 11){
    sum += i;
    i++;
  }
}
```

Figure 2.1: Example program

It's control-flow graph can be seen on 2.2. This type of graph is created from the program by grouping the statements into basic blocks. Each basic block consists a maximal amount of consecutive statements without any jumps. In a high-level language, such as C++, a jump can be either a branch statement, like an `if` and a `switch-case`, or a loop statement, like a `while`. In the example, we can see that it's CFG consists three basic blocks, and two special blocks, namely `entry` and `exit`, which represent the entry and exit points of the program, respectively.

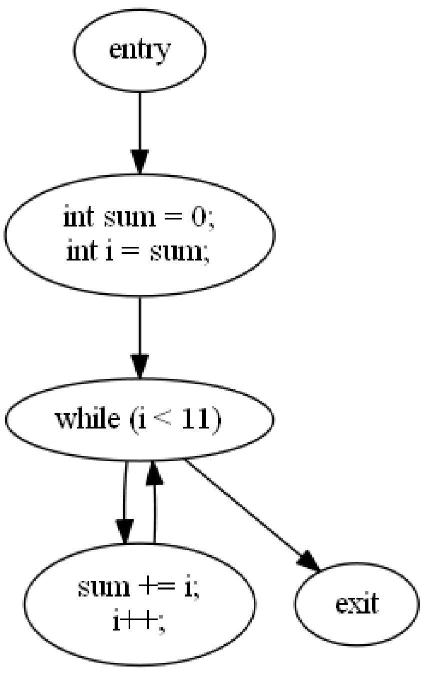

Figure 2.2: Control-flow graph

Therefore, control dependence can be defined in the knowing of post-dominance. As written in [2]:

"A node *i* in th CFG is *post-dominated* by

a node by *j* if all paths from *i* to STOP pass
through *j*. A node *j* is *control dependent* on a node *i* if (i) there exists a path *P* from *i* to *j* such that *j* post-dominates every node in *P*, excluding *i* and *j*, and (ii) is not post-dominated by *j*."

It basically means that every statement under a branch or loop are control dependent of the control statement's predicate. Excluding the unstructured and structured jump statements, `continue`,`break` and `goto`, drawing control dependences between statements creates a tree, with the root being the inspected function, and control statements form the branches. I discuss later in detail the control dependences created by structured jumps.

A data dependence between two statements means that if we change their order, the meaning of the program changes, or becomes ill-formed. There are two types of data dependences: *flow dependences* and *def-order dependences.* According to Horwitz[3], we can define these with the following rules:

There is a flow dependence between statement s1 and s2 if:

1. s1 defines variable x.

2. s2 uses x.

3. There is a path between s1 and s2 in program execution with no intervening definitions of x in between, and s2 can be reached by control from s1.

Definition in this context means a bit different than usual: it can be either an initial definition of a variable but an assignment where x is on the left side is also counts as a definition. In compiler theory, flow-dependence referred as reaching definition of a variable.

In the presence of loops, there are two further classification of flow-dependence: loop-carried and loop-independent. A flow-dependence is loop-carried if it satisfies all rules above and also:

1. includes a backedge to the predicate of the loop.

2. s1 and s2 are enclosed in the loop.

Backedge means that there is a flow dependence between the statement and the predicate of the loop, therefore the loop uses the variable which that statement defines. In the example above, the statement `i++` is like this.

Loop-independent flow-dependences can be described as the common ones which has no backedge to the loop predicate.

On the other hand, def-order dependence between statement s1 and s2 is different from loop-independent flow-dependences only in that s2 instead of using x, it also defines it. The other two rules stays the same.

These definitions are used in all slicing methods, but differently. I will elaborate these in each section.

2.3.2 Data flow equations

2.3.2.1 The algorithm

This method is created by Mark Weiser. It is based on the analysis of the CFG. As he wrote in [1], he defines a slice regarding a *slicing criterion*, which consists of a pair *(n, V)* where n is a statement of the program and V is a subset of the program's variables, which we are slicing on. He also writes that a slice of a program is an executable, which has only the relevant statements in it.

To calculate which statements should be included in the slice, he defines two sets of variables: *directly* and *indirectly relevant* variables. As written in [2], he provides the following equations for them:

For determining *directly* relevant variables and statements:

For each edge $i \rightarrow_{CFG} j$ in the CFG:

$$R^0_C(i) = R^0_C(i) \cup \{v | v \in R^0_C(j), v \notin \text{DEF}(i)\} \cup \{v | v \in \text{REF}(i), \text{DEF}(i) \cap R^0_C(j) \neq \emptyset\}$$
$$S^0_C = \{i | (\text{DEF}(i) \cap R^0_C(j) \neq \emptyset, i \rightarrow_{CFG} j\}$$

And for determining *indirectly* relevant variables and statements:

$$B^k_C = \{b | \exists i \in S^k_C, i \in \text{INFL}(b)\}$$
$$R^{k+1}_C(i) = R^k_C(i) \cup \bigcup_{b \in B^k_C} R^0_{(b,\text{REF}(b))}(i)$$
$$S^{k+1}_C = B^k_C \cup \{\text{DEF}(i) \cap R^{k+1}_C(j) \neq \emptyset, i \rightarrow_{CFG} j\}$$

He also says that a slice is statement-minimal if it couldn't have less statements. The slice then is computed in two steps: First by determining *directly* relevant variables by the equation above. In it, $\text{DEF}(i)$ and $\text{REF}(i)$ means the variables that statement i defines or uses, respectively. $R^0_C(i)$ will contain the directly relevant variables of statement i. It starts with the values in the slicing criterion: $R^0_C(n) = V$, and for all other sets initialized as $\emptyset$. As it can be seen from the first equation above, to get the R^0_C value for a node, it must use the computed values from every following node in the CFG. Weiser states in his work[1], that this is a simplification of the usual data flow information, which would use a PRE set to represent preservation of variable values. Reading the first equation, we see that variable v is in the $R^0_C(i)$ set either if:

1. v is in the slicing criterion V,
2. or v is in REF(i) and the R^0_C set of the following CFG node j has a value which is also in DEF(i)
3. or v is not in DEF(i) but v is in $R^0_C(j)$.

The last constraint ensures the propagation of the 'may be used' variables through statements which does not use nor reference the variables in the slicing criterion.

The second equation S^0_C then contains the statements which are included in the slice. It goes through the CFG from start and collects all variables that may have an influence on the slicing criterion. I've listed the sets of variables which are in DEF, REF and R^0_C of the statements of the example program above in the table below.

Node	DEF	REF	R^0_C
`int sum = 0;`	`sum`	$\emptyset$	$\emptyset$
`int i = sum;`	`i`	`sum`	`sum`
`while (i < 11)`	$\emptyset$	`i`	`i`
`sum += i;`	`sum`	`sum,i`	`i`
`i++`	`i`	`i`	`i`

Table 2.1: Results of the algorithm for the given example and criterion `(i++,{i})`

In this case, the set S^0_C contains lines `2,3,6`. As we see, this slice is incomplete since it does not include the `while` statement. For this to be included, further iterations should be done. In the equations above, Weiser introduces the INFL set for every statement i, which consists all statements which are control dependent on i. In our example, it is the empty set for all statements except the `while`. Using this, he defines the B^k_C set, which includes all statements which have a statement in it's INFL set which is included in S^k_C.

Now using B^k_C, we can add the `while` statement since the slicing criterion is control dependent on it. This is defined in the last equation, which is almost the same as the second, but it also includes the B^k_C set in it. When facing with nested branch, or looping statements, multiple levels of indirect dependences must be considered. The R^{i+1}_C set contains these for each level. Because of this, Weiser's algorithm needs iterations for achieving the correct slice. The R^k_C and S^k_C sets are non-decreasing, and they reach a fixpoint which are called R_C and S_C, respectively. The final S_C set contains the slice for the criterion.

I must note that this method computes the *backward* slice for any criterion, and cannot be used for forward slicing, since it computes only the statements which have influence on the criterion.

2.3.2.2 Efficiency

Weiser states that the complexity for computing S_C is $O(n\ e\ log\ e)$. He notes that this slice is not always the smallest, since it could include statements which cannot influence the slicing variable. He shows it in the following example program:

```
a = constant
while(p(k)){
  if (q(c)){
    b = a;
    x = 1;
  }else{
        c = b;
        y = 2;
  }
  k++;
}
z = x + y;
write(z)
```

For criterion `C = (write(z),{z})`, the set S_C will contain the first statement, but it does not have an influence on `z` since on any path by which `a` can influence `c` will execute `x = 1` and `y = 2`, therefore `z` becomes a constant value in this case.

However, computing slices with Weiser's algorithm needs little information from the program, since it requires only the CFG and the REF and DEF sets for every statement. The downside is that it only computes backwards slices, and since the criterion is fixed, it needs to compute every set in the equations if we need a slice for a different slicing criterion of the program.

2.3.3 Information flow relations

2.3.3.1 The algorithm

This algorithm is created by Bergeretti and Carré[4]. It has a different approach than Weiser's: it associates the program's statements with relations: that which expression may affect which variable. These relations has the definitions in the table below.

Empty statements
$D_S = \emptyset$
$\lambda_S = \emptyset$
$\rho_S = \iota$
$P_S = V$
$\mu_S = \emptyset$
Assignment Statements, in a form of `v = e`
$D_S = \{v\}$
$\lambda_S = \Gamma(e) \times \{e\}$
$\rho_S = \Gamma(e) \times \{v\} \cup (\iota - \{(v, v)\})$
$P_S = V - \{v\}$
$\mu_S = \{(e, v)\}$
Sequences of statements, in a form of A;B
$D_S = D_A \cup D_B$
$\lambda_S = \lambda_A \cup \rho_A \lambda_B$
$\rho_S = \rho_A \rho_B$
$P_S = P_A \cap P_B$
$\mu_S = \mu_A \rho_B \cup \mu_B$
Conditional statements, in a form: `if (e) A else B`
$D_S = D_A \cup D_B$
$\lambda_S = (\Gamma(e) \times \{e\}) \cup \lambda_A \cup \lambda_B$
$\rho_S = (\Gamma(e) \times (D_A \cup D_B)) \cup \rho_A \cup \rho_B$
$P_S = P_A \cup P_B$
$\mu_S = (\{e\} \times (D_A \cup D_B)) \cup \mu_A \cup \mu_B$
Conditional statements, without else: `if (e) A`
$D_S = D_A$
$\lambda_S = (\Gamma(e) \times \{e\}) \cup \lambda_A$
$\rho_S = (\Gamma(e) \times D_A) \cup \rho_A \cup \iota$
$P_S = V$
$\mu_S = (\{e\} \times D_A) \cup \mu_A$
Repetitive statements, in a form: `while (e) A`
$D_S = D_A$
$\lambda_S = \rho_A^*((\Gamma(e) \times \{e\}) \cup \lambda_A)$
$\rho_S = \rho_A^*((\Gamma(e) \times D_A) \cup \iota)$
$P_S = V$
$\mu_S = (\{e\} \times D_A) \cup \mu_A \rho_A^*((\Gamma(e) \times D_A) \cup \iota)$

Table 2.2: Table of information flow relations

The statements are classified into two primitive types: Empty and Assignment, and three hiearchical types: Sequence, Conditional with `else` branch, without `else` and the Repetitive statements. They are constructed from the control-flow graph. For analysis, they define *variables* and *expressions.*For Assignment, it's left side is the variable, and it's right side is the expression associated with it. Only this type of statement has a variable. For Conditional and Repetitive statements their conditional expression used, and Empty and Sequence has no variable or expression. There's also the definition of $\Gamma(e)$ which consists the variables which appear in e. We define E and V which contains all of the variables and expressions of the program. In D_S there are the variables which S *may define.* In P_S are the variables which S *may preserve.* The 'multiplication' sign which we do not write between relations is the *relational composition*: For relations X and Y, where $X \subseteq A \times B$ and $Y \subseteq B \times C$:

$$XY = \{(a, d) \in A \times C | (a, b) \in X, (c, d) \in Y, b = c\} \tag{2.1}$$

The main three binary relations are the following:

1. λ_S, from V to E
2. μ_S, from E to V
3. ρ_S, from V to V

For a statement S, $v \in V, e \in E$: $v\lambda_S e$ means that 'the value of v on entry to S *may be used* in the evaluation of the expression e in S.' Taken the following program:

It can be seen that in case of the `while` statement, variable n and the predicate of `while` is in the $\lambda_{\texttt{while}}$ relation.

Taking a look at μ_S, it means that for expression e and variable v : $e\mu_S v$ 'a value of e in S *may be used* in obtaining the value of the variable v on exit from S'. Looking at our example again, for the expression of `if` statement and variable x, $\mu_{\texttt{if}}$ holds, since the execution of assignments to x are control dependent on the `if` statement.

It is basically the dual relation of λ_S, which will be very useful in aspect of slicing.

The final relation ρ_S can be constructed from λ_S,μ_S and P_S:

$$\rho_S = \lambda_S \mu_S \cup \Pi_S, \textit{ where } \Pi_S = \{(v, v) \in V \times V \mid v \in P_S\}. \tag{2.2}$$

This combination of λ_S and μ_S creates a relation which is for $v, w \in V$:

1. either the entry value of v to S *may be used* obtaining the exit value of w, or

```
void fn(int x){
  read(n);              //1
  int i = 1;            //2
  while (i <= n){       //3
    if (i \% 2 = 0) {  //4
      x = 17;           //5
    }
    else{
      x = 18;           //6
    }
    i = i + 1;          //7
  }
  write(x);             //8
}
```

Figure 2.3: Example program 2

2. $v = w$, and S *may preserve* v.

In the code above, for the `if` statement we get the `{(i,x),(i,i),(n,n),(x,x)}` set of pairs for it's $\rho_{\texttt{if}}$ relation, since it can be seen that variable i is in the predicate, so it may be used for evaluating x, but also, if the predicate does not hold, then the statement preserves all three variables of the program.

These relations are different for each type of statements.

For Empty, the sets D_S, λ_S and μ_S are the empty set since it has no variable in it. P_S is the whole V because it does not modify any variable, therefore it preserves every variable. Also, for the ρ_S relation we define the identity relation ι for variables $v \in V$:

$$\iota = \{(v, w) \in V \times V | v = w\}. \tag{2.3}$$

Which means the same as the P_S set.

For Assignment statements, the definitions are simple. They have the form of `v = e`, so in D_S, there's v, in P_S, there's every variable minus v, since the statement assigns the value of the expression e to it. In case of λ_S, it is trivial that the variables of e - denoted by $\Gamma(e)$ - may be used in evaluating e. Also, μ_S defines that the right-hand side effects the left-hand side of the assignment. As written above, ρ_S can be expressed as:

$$\begin{aligned} \rho_S = \lambda_S\mu_S \;\cup\; \Pi_S \;=\; (\Gamma(e) \times e)((e, v)) \;\cup\; \Pi_S \;= \\ = (\Gamma(e) \times v) \;\cup\; \Pi_S \;=\; (\Gamma(e) \times v) \;\cup\; (\iota \;-\; \{(v, v\}) \end{aligned} \tag{2.4}$$

In Assignment, Π_S does not contain the variable, because it is being changed.

In the case of Sequence of statements, they have no direct variable or expression associated with them. They represent the single arrow between two nodes in the CFG. So, for the sequence of statements $A; B$, the relation $e\lambda_S v$ contains either

1. e which is in A and $v\lambda_A e$ or
2. e is in B and there is a variable w that $v\rho_A w$ and $w\lambda_B e$.

As defined above, ρ_A gets all the variables from statement A which are preserved or may be used, therefore they are fitting in the definition for the sequence.

Similarly, μ_S contains pairs of (v, e) which either

1. e is in A and there is a variable w that $e\mu_A w$ and $w\rho_B v$ or
2. e is in B and $e\mu_B v$

The duality of λ_S and μ_S can again be seen. For ρ_S, we can substitute the formulas:

$$\rho_S = (\lambda_A \cup \rho_A\lambda_B)(\lambda_A \cup \rho_A\lambda_B) \cup (\Pi_A \cap \Pi_B) \tag{2.5}$$

For brevity, I don't follow through the substitution here. The result of it is $\rho_A\rho_B$. In case of a sequence of statements which has more than two statements in it, we can define it by nesting: $(((A; B); C); ...)$. The ordering of parentheses are irrelevant.

For Conditional statements, there is a control dependence between it's predicate e and the statements A, B in them. Therefore, (v, e) is in λ_S if

1. e is the predicate and $v \in \Gamma(e)$
2. e is in A and $v\lambda_A e$ or
3. e is in B and $v\lambda_B e$.

In case of μ_S, (v, e) is in the relation if:

1. e is the predicate, and either A or B define v, or
2. e is in A and $e\mu_A v$ or
3. e is in B and $e\mu_B v$.

With substituting λ_S and μ_S into the formula of ρ_S above, we get:

$$\rho_S = (\Gamma(e) \times (D_A \cup D_B)) \cup \rho_A \cup \rho_B \tag{2.6}$$

When the Conditional statement has no `else` branch, we can substitute it with an Empty expression, gaining the simplified formulas which can be seen in table 2.2.

Bergeretti in his paper states that Repetitive statements can be expressed as an infinite sequence of nested conditional statements in a form of:

```
if(e){
  A;
  if(e){
    A;
    ...
  }
}
```

Therefore we can get the formula of λ_{Rep} if we repeatedly use λ_{Cond} and $\lambda_{textscSeq}$, gaining the following:

$$\lambda_S = \rho_A^*((\Gamma(e) \times \{e\}) \cup \lambda_A) \tag{2.7}$$

where ρ_A^* is the *transitive closure* of ρ_A.

For any relation R, it's transitive closure is the fixpoint of the following set:

$$R^* = \bigcup_{i \in \{1,2,\ldots\}} R^i \tag{2.8}$$

The other two relations can be similarly derived by applying their Conditional and Sequence versions, thus getting:

$$\begin{aligned} \rho_S &= \rho_A^*((\Gamma(e) \times D_A) \cup \iota) \\ \mu_S &= (\{e\} \times D_A) \cup \mu_A \rho_A^*((\Gamma(e) \times D_A) \cup \iota) \end{aligned} \tag{2.9}$$

For determining a slice of a program with these relations, Bergeretti describes the definition of partial statements, with the formula:

$$E_S^v = \{e \in E \mid e \mu_S v\} \tag{2.10}$$

As it can be seen from the definition of μ_S, this results in the whole slice for variable v, if the given statement S is the 'statement' of the whole program, as a Sequence of top-level statements. It also gets the *expressions* which may affect the variable, and not the statements consisting it. However, using μ_S, we can create formulas which effectively get the desired statements from the program. It can be seen that with the dual nature of μ_S and λ_S, they can be used for backwards slicing,and for forward slicing, respectively.

2.3.3.2 Efficiency

This algorithm is throroughly defines the flow of data across statements, taking control dependences into account. Further extensions, like applications for `switch-case` expressions can be added by defining the relations for them, on the basis of the current ones. It needs deeper parsing than Weiser's method, for getting all the variables of the expressions, and statements. However, if we parse the program code, and collect the D_S, v,e, $\Gamma(e)$ sets, variables and expressions for each statement, we can compute different slices for different variables and statements after for the program. In the case of large codebases, this is an important aspect of the algorithm.

In the presence of loops, calculating the reflexive closure takes additional time. Bergeretti states the following for the complexity of the calculation of relations: ρ_S has a worst case asymptotic $O(|V|^3)$. Therefore, for all statements it is $O(|E| \times |V|^3)$. For λ_S and μ_S, it is $O(|E| \times |V|^2)$ for each statement, or $O(|E|^2 \times |V|^2)$. If we compare these with Weiser's, it is clear that this algorithm requires more computation for slicing the program for the first time, however Bergeretti noted that these times are not problematic in real life examples.

2.3.4 Dependence graph based slicing

2.3.4.1 The algorithm

The third algorithm is first written by Ottenstein and Ottenstein[10], and has been a base for different slicing techniques, used by Susan Horwitz[3], and [5], [6], [9] and [8].

This algorithm uses the Program Dependence Graph, which is a directed graph constructed either from the control-flow graph or the abstract syntax tree *(AST)* of the program. Horwitz describes this graph thoroughly in her paper[3]. It contains the control dependence and data dependence subgraphs. The nodes of it are the statements of the program. There is a special node called the entry node, which is considered to be before any statement in the CFG. Building the graph is done in two steps: first by creating the control dependence subgraph, then drawing the data dependence edges in it. Control dependence edges has boolean labels: they are either `true` or `false`.

There is a control dependence edge between two nodes v_1 and v_2 if:

1. v_1 is the entry node, and v_2 is a statement which is not under any loop or branch.

2. v_1 is a loop or branch statement, and v_2 is *immediately* nested under v_1. If v_1 is a loop statement, then the label of the edge is `true`, if it is a branch statement, then if v_2 is under the 'then' branch, the label is `true`, and `false` otherwise.

The control dependence subgraph has similar structure as the AST. It is the direct consequence of that the usually the loop and branch statements has other statements nested in them. In [7], there are region nodes created for predicate of each loop or branch. In practice, I have found these unnecessary to deal with. I will later discuss it in the chapter about the implementation.

After constructing the control dependence subgraph, data dependence edges are added. The meaning of this kind of edge between v_1 and v_2 is that if we change the order of these statements then the program may become ill-formed or the computation is changed. In the aspect of slicing, there are two main different type of edges: *flow dependence* and *def-order dependence*. In compiler theory, in data-flow analysis there is the 'use-def' or 'def-use' chain, which has similar definitions what Horwitz describes in her paper. There is a flow dependence edge between nodes v_1 and v_2 if:

1. v_1 is a node which defines variable x and
2. v_2 uses x and
3. Control can reach v_2 after v_1 in an execution path which does not contain any intervening definition of x.

The third rule is easy to verify if the PDG is built using the CFG, but using the 'def-use' chains, it also can be derived from the AST.

In the case of loops, there are two further sub-types of flow dependence edges: *loop-carried* and *loop-independent*. In addition to the three rules above, Loop-carried edges must hold that:

1. The path in the third rule contains a backedge to the predicate of loop L and
2. v_1 and v_2 are both part of loop L.

Loop-independent edges are not keeping the first rule above, so the path does not contain a backedge to the predicate. There is a def-order dependence edge between node v_1 and v_2 if:

1. v_1 and v_2 defines the same variable and
2. There is a execution path between v_1 and v_2 without any intervening definition of that variable.

Taken the following example program:

```
 1  int main(){
 2    int x = 0;
 3    int y = 1;
 4    while(x >= 0){
 5      if (x > 0){
 6        y = x; //2. use here
 7      }else{
 8        x = 3;
 9      }
10      x = 2; //1. define here
11    }
12  }
```

Figure 2.4: Example program 3

We can see the created PDG of it on 2.5. The bold arrows represent the control dependence edges and the others are the data dependence edges. In her paper, Horwitz creates loop edges to the statements which are under a loop, but regarding slicing, they are not making a difference. As it can be seen, the nodes labeled `{}` are the brackets which consist multiple statements. I have created this example to show how a loop-carried edge can be found between different scopes under a loop statement. The comments in the example code shows the order of 'def-use' relationship created by the loop: when control reaches 1., x gets defined, then in the second second iteration control goes into the true branch of the `if` statement and reaches 2., creating a 'def-use' data dependence between the two statements. Among this, there are examples of def-order dependence edges: between line 8. and line 10., and also between the initial definition of x and other definitions later.

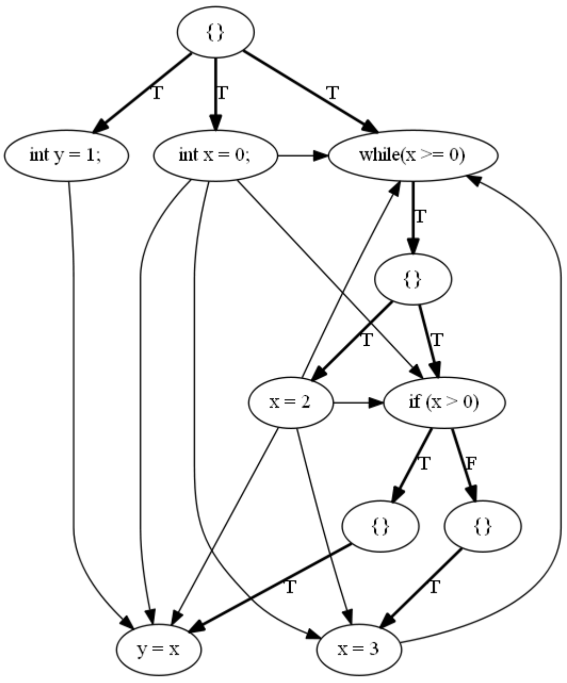

Figure 2.5: PDG of 2.4

Drawing data dependence edges are a non-trivial task. I will explain the details in the implementation chapter. However, the PDG itself when built, is a very useful abstraction of the program. To slice regarding a statement and variable in the program, we need to do only a graph reachability search in the graph over the control and data dependence edges.

If we store the edges of the graph bidirectionally, the difference becomes only the direction of search between creating backward and forward slices. We can see an example of a backward slice on 2.6, regarding the x variable on line 9.

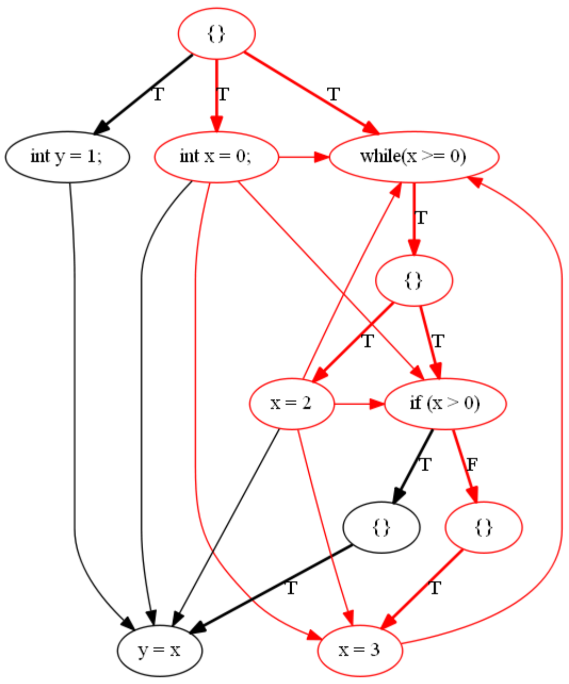

Figure 2.6: Backward slice of 2.4 regarding line 9.

Chapter 3

LLVM/Clang infrastructure

3.1 About Clang

LLVM is a compiler infrastucture which is created mainly for program optimization during compile, link and runtime. It is written in C++ and language-agnostic itself, and uses an intermediate representation (LLVM IR) of the programs created in numerous languages. LLVM stands for **Low Level Virtual Machine** which was the initial target for the project, but since then it has grown to a massive infrastructure for program compilation and toolchains. LLVM does not handle the parsing and syntactic error checking in the languages, and separates this part via the IR. It was initially developed for C and C++, but since then there are numerous language frontends for a wide variety of languages including Haskell, Ada, Ruby, Python, etc.

Clang is a compiler frontend for C,C++ and Objective-C, using LLVM as a backend for code generation. Clang is also written in C++, and has a library-based architecture which allows the compiler for not just compiling, but to build useful tools with it. For example, QtCreator IDE - an Integrated Development Environment - can use Clang as a 'code model', for syntactically and semantically check the code in the editor, making error correcting much easier for the developer. There are also numerous tools built on the Clang using it as a library for parsing C++ code, for gaining information of it or performing source-to-source transformations.

3.2 The Clang AST

The internal representation of C++ source code in Clang is the **Abstract Syntax Tree** - AST**??**. Since it is written in C++, it uses the language's object-oriented features for encoding the structure of the program into a large class hiearchy, keeping all relevant information of a language element in an AST node.

Taken the example from Chapter 2:

```
int main(){
  int sum = 0;
  int i = sum;
  while (i < 11){
    sum = sum + i;
    i++;
  }
}
```

```
FunctionDecl 0x595bd10 <horwitz.cc:1:1, line:8:1> line:1:5 main 'int (void)'
`-CompoundStmt 0x2716a50 <col:11, line:8:1>
  |-DeclStmt 0x2716868 <line:2:3, col:14>
  | `-VarDecl 0x2716810 <col:3, col:13> col:7 used sum 'int' cinit
  |   `-IntegerLiteral 0x2716848 <col:13> 'int' 0
  |-DeclStmt 0x27168e8 <line:3:3, col:14>
  | `-VarDecl 0x2716888 <col:3, col:11> col:7 used i 'int' cinit
  |   `-ImplicitCastExpr 0x27168d8 <col:11> 'int' <LValueToRValue>
  |     `-DeclRefExpr 0x27168c0 <col:11> 'int' lvalue Var 0x2716810 'sum' 'int'
  `-WhileStmt 0x2716a38 <line:4:3, line:7:3>
    |-<<<NULL>>>
    |-BinaryOperator 0x2716940 <line:4:10, col:14> '_Bool' '<'
    | |-ImplicitCastExpr 0x2716930 <col:10> 'int' <LValueToRValue>
    | | `-DeclRefExpr 0x27168f8 <col:10> 'int' lvalue Var 0x2716888 'i' 'int'
    | `-IntegerLiteral 0x2716910 <col:14> 'int' 11
    `-CompoundStmt 0x2716a20 <col:17, line:7:3>
      |-BinaryOperator 0x27169d8 <line:5:5, col:17> 'int' lvalue '='
      | |-DeclRefExpr 0x2716958 <col:5> 'int' lvalue Var 0x2716810 'sum' 'int'
      | `-BinaryOperator 0x27169c0 <col:11, col:17> 'int' '+'
      |   |-ImplicitCastExpr 0x27169a0 <col:11> 'int' <LValueToRValue>
      |   | `-DeclRefExpr 0x2716970 <col:11> 'int' lvalue Var 0x2716810 'sum' 'int'
      |   `-ImplicitCastExpr 0x27169b0 <col:17> 'int' <LValueToRValue>
      |     `-DeclRefExpr 0x2716988 <col:17> 'int' lvalue Var 0x2716888 'i' 'int'
      `-UnaryOperator 0x2716a08 <line:6:5, col:6> 'int' postfix '++'
        `-DeclRefExpr 0x27169f0 <col:5> 'int' lvalue Var 0x2716888 'i' 'int'
```

Figure 3.1: Clang AST of the example

Clang enables us to dump it's AST via a compiler option, which can be seen on figure 3.1.

The type of the nodes has a class hierarchy which has no common ancestor. There are a several base types, but only two are important for implementing slicing algorithms: the `Decl` and `Stmt` class. As it is logical, they encode a generic declaration, and statement, respectively. For example in the AST above, `FunctionDecl`, VarDecl are Decls, and WhileStmt, CompoundStmt and DeclStmt are Stmts. Also, it can be seen that Stmts create a clean tree hierarchy, where a predicate of a while loop belongs under it, enabling the AST to describe complex expressions. There is an API created for accessing the nodes' properties, which supports iterators on nodes with `begin()` and `end()` much like the C++ standard library. For variable declarations, there is a wrapping `DeclStmt`, and for every block of code denoted by curly braces, there is a `CompoundStmt` associated with it.

The Clang AST has a bit different logic in naming language elements than we would expect. It encodes assignments in a more generic `BinaryOperator`, and the API provides the `IsAssignment()` boolean method for it. It also encodes a defined variable's access in a `DeclRefExpr`, and if they are used on a right-hand side of an operator, it uses implicit cast expressions for casting them from lvalue to rvalue.

The source positions are also encoded in the nodes, but for accessing them, we need the `SourceManager` class, which can get the exact location of a node.

3.2.1 The RecursiveASTVisitor class

Clang provides the `LibTooling` library to work with the AST. First, Clang parses the source code, and performs the compilation steps necessary to build the AST for us. Therefore if there's an error in the code, we won't get a partial AST. After then, we must provide a few classes which then can take care to run our action as a compilation step. The entry point is the `ASTFrontendAction` class, which creates an `ASTConsumer`, which can access the AST of the whole translation unit. Clang provides the `RecursiveASTVisitor` class, where we can create two types of methods for processing nodes: `Visit<NodeType>` and `Traverse<NodeType>`, where we put the node type which we want to process in the name of the method. With Visit, we can access the node, but Traverse can be used for controlling the traversal of the nodes, for example, if we do not want to visit any node under an If Statement, or to swap the traversal order of the two branches.

3.3 AST Matchers

Although the RecursiveASTVisitor class could be enough for processing the AST, it could get easily complicated and require a lot of repeated code, making the tool less maintainable. However, there is a very useful API included with LibTooling of Clang, the ASTMatchers. With them, we can define the nodes we look for in the AST in a chain of matcher calls, which are a small domain-specific language defined for this specific task, but written in C++, naturally. For example, the following expression:

```
binaryOperator(isAssignmentOp(),
    hasLHS(ignoringImpCasts(declRefExpr(to(varDecl().bind("lval"))))),
    anyOf(
      hasRHS(forEachDescendant(
        expr(declRefExpr(to(varDecl().bind("rval")))))),
      hasRHS(anything())
    )).bind("binop");
```

Has a `binaryOperator` matcher, so with the internal boolean predication `isAssignmentOp`, we can specify that it should match on assignment operators only. Via `hasLHS` and `hasRHS` inner matchers, we can specify what should be on the left and right side of the matched operator. And finally, the `anyOf` matcher can take arbitrary amount of arguments - which are also matchers - and becomes true if any of it's matchers are true.

This only specifies what nodes we want to work with. For processing them, there is the `bind()` method, where we can specify a string identifier for the matched node. In the example above, there is a match for every assign statement, and every variable in them, separately. Every matcher has a callback function which is called every time when a node matching our specified matcher matches. We can define the processing method for the matches, and extract the statements or declarations bound via `bind` in it. The example shows how can we extract the variable declarations referenced in the `DeclRefExprs` in the assignment.

These ASTMatchers have a lot of similarities in common for querying with an NoSQL database query language. The separation of matching and processing enables us to avoid writing a lot of 'boilerplate' code, and provides an intuitive way of specifying the desired information to query from the AST.

There are three types of matchers:

1. Node: match a specific type of AST node,

2. Narrowing: match attributes on AST nodes,

3. Traversal: allow traversal between AST nodes.

In the previous section, in RecursiveASTVisitor, Visit was the equivalent of the Node matchers here. Every match expression starts with a node matcher, and they take arbitrary amount of parameters and match if all parameters returns true. There are a node matcher for every type of node in the AST. They are the ones which support the `bind` method.

Narrowing matchers are boolean expressions which are narrowing down the set of nodes of the current type to match on. For example, `isAssignmentOp` is one of them, which can be used in `BinaryOperator` matchers. There are a few special logical matcher of this kind (`allOf`, `anyOf`, `anything`, `unless`), which can be used to build more complex matchers, as it can be seen from the example that with `anyOf` we specify that the right-hand side of the BinaryOperator has either a variable or something else, with `anything`.

The Traversal matchers are different from the Traverse method from RecursiveASTVisitor. They can be used to specify the child or subexpression of the node we want to match. There are a few special matchers of this kind (`has`, `hasDescendant`, `forEach`, `forEachDescendant`), which work on all type of nodes. In the example, forEachDescendant is used to match all variables on the right hand side of the assignment.

All kind of matchers can be extended by defining them with special macros also provided by the API. All of the default matchers can be found under[12].

For easing the building of complex matchers, there is a tool in the Clang repository named *clang-query* which is an interpreter for evaluating matchers.

There are numerous tools built with this API, the most notable must be the Clang Static Analyzer[11], which is a source code analysis tool that finds bugs in programs. It has a modular pluggable design, implemented with different matchers for finding flaws and potential patterns which would lead to bugs, for example unitialized variables, nullpointer accesses, and other errors which can be detected by analyzing the AST.

Chapter 4

Implementation and algorithm

In this chapter I will introduce a prototype application for slicing in programs written in C++ using the tools described in the previous chapter. I will compare the described algorithms by usability, extendability and efficiency.

4.1 The approach

For analyzing C++ source code, the best tool is the Clang LibTooling library. The GCC compilers internal representation transforms the original source in non-reversible way, so it is not useful in this setting. There are other approaches which implement slicing, notably the Wisconsin Program-Slicing tool[14], and Frama-C has a plug-in for slicing[15].

The three algorithms described in the second chapter are all can be implemented

4.2 Building the PDG

4.2.1 Control dependences

4.2.2 Data dependences

4.3 Implementing slicing

Bibliography

[1] M. Weiser, *Program slicing*, IEEE Transactions on Software Engineering, 10(4):352-357, 1984.

[2] Tip, Frank, *A survey of program slicing techniques*, Journal of programming languages 3.3, 121-189, 1995.

[3] Horwitz, Susan, Thomas Reps, and David Binkley, *Interprocedural slicing using dependence graphs*, ACM Transactions on Programming Languages and Systems, (TOPLAS) 12.1: 26-60, 1990.

[4] Bergeretti, Jean-Francois, and Bernard A. Carré, *Information-flow and data-flow analysis of while-programs*, ACM Transactions on Programming Languages and Systems, (TOPLAS) 7.1: 37-61, 1985.

[5] Horwitz, Susan, and Thomas Reps, *The use of program dependence graphs in software engineering*, Proceedings of the 14th international conference on Software engineering. ACM, 1992.

[6] Reps, Thomas, *Program analysis via graph reachability*, Information and software technology, 40.11: 701-726, 1998.

[7] Harrold, Mary Jean, Brian Malloy, and Gregg Rothermel, *Efficient construction of program dependence graphs*, ACM SIGSOFT Software Engineering Notes, Vol. 18. No. 3. ACM, 1993.

[8] Larsen, Loren, and Mary Jean Harrold, *Slicing object-oriented software*, Software Engineering, Proceedings of the 18th International Conference on. IEEE, 1996.

[9] Agrawal, Hiralal, *On slicing programs with jump statements*, ACM Sigplan Notices, Vol. 29. No. 6. ACM, 1994.

[10] Ottenstein, Karl J., and Linda M. Ottenstein, *The program dependence graph in a software development environment*, ACM Sigplan Notices, Vol. 19. No. 5. ACM, 1984.

[11] *Clang Static Analyzer* `http://clang-analyzer.llvm.org/`

[12] *AST Matcher Reference* `http://clang.llvm.org/docs/LibASTMatchersReference.html`

[13] *Introduction to the Clang AST* `http://clang.llvm.org/docs/IntroductionToTheClangAST.html`

[14] *The Wisconsin Program-Slicing Tool* `http://research.cs.wisc.edu/wpis/slicing_tool/`

[15] *Slicing plug-in for Frama-C* `https://frama-c.com/slicing.html`