Universität-Gesamthochschule Siegen

Fachbereich 12 – Elektrotechnik und Informatik

Dissertation

zur Erlangung des Grades eines

Dr.-Ing.

Call Path Sensitive Interprocedural Alias Analysis of C Programs

Dirk Schmidt

Eingereicht am: 08.03.1999
Mündliche Prüfung am: 28.04.1999

Betreuer: Prof. Dr. W. Merzenich
Prof. Dr. G. Wirtz

Zusammenfassung

Das Thema dieser Arbeit ist ein neuer Ansatz zur Bestimmung von may-alias-Beziehungen im Rahmen von ANSI-C-Programmen. Von einem may-alias spricht man, wenn zwei verschiedene Variablen (oder auch komplexere Ausdrücke) ein und dieselbe Speicherstelle verwenden. Aliase entstehen durch die Verwendung von call-by-reference-Parametern oder Zeigern. Im Rahmen von C-Programmen entstehen hier besonders schwerwiegende Probleme, da C-Programme in der Regel intensiven Gebrauch von Zeigern machen und darüber hinaus nur sehr wenige Restriktionen für den Einsatz von Zeigern existieren.

Im Rahmen dieser Arbeit wird ein Verfahren beschrieben, daß die Zusammenfassung der für die Alias-Analyse relevanten Effekte mit Hilfe von Graphen (function interface graphs) realisiert. Diese Graphen stellen eine statische Repräsentation der von einem Programm benutzten Speicherstellen, sowie der darin enthaltenen Werte dar. Basierend auf einer Reihe von Standard-Verfahren (Normalisierung der Aufrufe und Rückgabewerte einer Funktion, Kontrollflußgraphen, Static-Single-Assignment-Form) wird während der intraprozeduralen Phase zunächst für jede Funktion ein Graph gebildet, der ihre Effekte zusammenfaßt. Hierzu werden einzelne Graphen für die rein sequentiellen Blöcke (basic blocks) einer Funktion berechnet, die dann nachher unter Zuhilfenahme des Kontrollflußgraphen vereinigt werden. Nachdem die Graphen für einzelne Funktionen erzeugt worden sind, werden anschließend die Teile der Graphen entfernt, die nicht zur Repräsentation der von außen sichtbaren Effekten dienen. Die eine Funktion repräsentierenden Graphen werden auf diese Weise deutlich kleiner, was zu einer Effiziensssteigerung für die nachfolgenden Schritte führt. Die reduzierten Graphen werden dann gemäß der existierenden Funktionsaufrufe miteinander vereinigt (interprozedurale Analyse). Hierbei werden indirekte Funktionsaufrufe (über Zeiger auf Funktionen) zunächst ignoriert. Erst wenn im Rahmen der Analyse festgestellt wird welche Funktionen ein solcher Aufruf aufrufen könnte, werden auch die zu diesem Aufruf gehörenden Graphen vereinigt. Funktionen die an mehreren Stellen aufgerufen werden, werden hierbei solange wie möglich unter Berücksichtigung des jeweiligen Aufruf-Kontext betrachtet.

Ein wesentlicher Vorteil dieses Algorithmus besteht darin, daß er für viele reale ANSI-C-Programme verwendet werden kann, da nur sehr wenige Einschränkungen notwendig waren (Assembler-Code, Interrupt-Handling, volatile-Attribute und I/O-basierte Aliase). Der Algorithmus behandelt die Effekte von Strukturen und Unions, beliebiger Verwendung von Zeigern, Typ-Umwandlungen und Zeigern auf Funktionen. Da dies dazu führt, daß sich der Algorithmus nicht auf die vorhanden Typ-Informationen stützen kann, basieren die erzeugten Graphen auf einem Speichermodell auf niedriger Ebene. Der Algorithmus wurde mit Hilfe des SUIF-Compilers implementiert und erfolgreich mit einer Reihe von realen Programmen getestet.

Abstract

This work presents a new approach to detect may-aliases within ANSI-C programs. A may-alias occurs if two variables (or more complex expressions) use the same memory location. Aliases may be created by call-by-reference function calls or by pointer usage. In the context of C programs alias analysis becomes very complex because of the extensive pointer usage and the very few restrictions concerning pointer manipulations.

Within this work it is described how the effects of a program can be summarised by so-called function interface graphs which are a static representation of the memory locations and the values stored at these locations. Based on standard techniques (function call and return statement normalisation, control flow graphs, static single assignment form), the intraprocedural step creates a function interface graph for each function individually by computing and merging the information for all its basic blocks according to the CFG structure. Afterwards these graphs will be reduced to their externally visible effects. This reduces the size of these graphs and will hence allow the following computations to be carried out more efficiently. Within the next step the reduced graphs will be merged according to the corresponding functions and their calls (interprocedural analysis). This is done by ignoring indirect function calls (through function pointers) first, and processing these calls only if a function that could be called by one of these calls is detected. In case functions are called by more than a single function call, the calling context will be taken into account as far as possible.

A major benefit of our algorithm is that it can be applied to real ANSI-C programs since it makes only a few restrictions, i.e. effects of assembler code, interrupts, volatile attributes and I/O-based aliases. It deals with structures and unions, arbitrary pointer usage, type casts and function pointers. Under these circumstances, the algorithm cannot benefit from type information and, hence, the function interface graphs are based on a low-level memory representation. The algorithm was implemented using the SUIF compiler, and it has been successfully applied to a set of non-toy C programs.

Contents

1 Introduction **1**

2 Basic concepts **5**

2.1 Restrictions for the analysed programs . . . 5

2.1.1 Input data read from files or entered by the user . . . 6

2.1.2 'Hidden' assignments . . . 6

2.2 SUIF . . . 7

2.3 Normalisation . . . 8

2.3.1 Function call normalisation . . . 9

2.3.2 Return value normalisation . . . 10

2.3.3 Dead code elimination . . . 11

2.3.4 Time and space bounds . . . 11

2.4 Attribute grammars . . . 11

3 Alias computation **15**

3.1 SSA form . . . 15

3.1.1 CFG graphs . . . 17

3.1.1.1 CFG node types . . . 19

3.1.1.2 Sequential statements (*SEQ*) . . . 20

3.1.1.3 *IF-THEN-ELSE* statements . . . 21

3.1.1.4 *FOR-LOOP* statements . . . 22

3.1.1.5 *DO-LOOP* statements . . . 23

3.1.1.6 Function calls (*CALL*) . . . 24

3.1.1.7 *LABEL* and *JUMP* statements . . . 24

3.1.2 Computation of SSA values . . . 25

3.1.2.1 Domination, dominator trees and dominance frontiers . . . 26

3.1.2.2 Algorithm . . . 28

3.1.2.3 Time and space bounds . . . 37

3.2 Intraprocedural analysis . . . 41

3.2.1 Alias information . . . 41

3.2.1.1 Basic definitions . . . 41

3.2.1.2 Aliases computed by the algorithm . . . 53

3.2.2 Function interface graphs . . . 54

3.2.2.1 Offset annotations . . . 58

3.2.2.2 Generation mode annotations . . . 58

3.2.3 Building the function interface graph for statement sequences belonging to the same CFG node . . . 60

3.2.3.1 The attribute grammar generating the function interface graphs . . . 60

3.2.3.2 Order of attribute computations 72
3.2.3.3 An example 75
3.2.3.4 A more precise algorithm 75
3.2.3.5 Handling ϕ-functions 77
3.2.3.6 CFG nodes representing function calls 78
3.2.4 Merging the function interface graphs of different CFG nodes 80
3.2.4.1 An example 86
3.2.5 Using the function interface graph to find may aliases 89
3.2.6 Correctness 91
3.2.7 Time and space bounds 96
3.2.7.1 Time bounds 96
3.2.7.2 Space bounds 96
3.3 Interprocedural analysis 98
3.3.1 Basic definitions 98
3.3.2 Building the call graph 102
3.3.3 Combining the function interface graphs produced by the intraprocedural analysis 106
3.3.3.1 Reducing the function interface graph for the interprocedural analysis 107
3.3.3.2 Merging the function interface graphs for a single function call 108
3.3.3.3 Call path insensitive analysis 113
3.3.3.4 Call path sensitive analysis 116
3.3.4 Correctness 122
3.3.5 Time and space bounds 122
3.3.5.1 Time bounds 122
3.3.5.2 Space bounds 125
3.4 Handling C properties needing exceptional treatment 127
3.4.1 Handling library functions 127
3.4.2 Variable parameter lists 128
3.4.3 The `longjmp` and `setjmp` functions 129

4 Tests, implementation and empirical results 131
4.1 The test suite 131
4.2 Normalisation pass 133
4.3 SSA form computation 134
4.4 Intraprocedural analysis 135
4.5 Interprocedural analysis 137

5 Related work 143
5.1 Other alias analysis algorithms 143
5.2 Other interprocedural analysis algorithms 146
5.3 SSA form 147
5.4 Data structures used by data flow analysis algorithms 148
5.5 Complexity of alias analysis 148
5.6 Possible applications for alias analysis 149

6 Conclusion and future work 151

Chapter 1

Introduction

The subject of this work is the analysis of C programs with the goal of finding possible aliases of variables. Aliases appear in programs when the same memory location can be accessed (read and modified) by different variables. Such situations occur due to call by reference parameters or pointer variables. Since the C programming language does not support call by reference parameters, the aliases produced by the use of pointers are in the centre of this work. The lack of call by reference parameters does not really restrict the language. In fact an effect similar to call by reference parameters can be achieved by passing a variables address instead of the variable itself to a function.

Basically, there are two different kinds of aliases: may and must aliases. May aliases occur only if certain conditions concerning the input data hold, whereas must aliases occur in any case not depending on the programs inputs. The algorithm presented here performs a may alias computation. It is precise in the sense that every possible may alias will be found. Unfortunately, the opposite does not hold as well, and so there may be aliases found by the algorithm that may not occur when the program is executed. However, it is not possible to avoid this kind of imprecision. As was shown in [Hor97, LR91] an exact alias computation is $\mathcal{NP}$-hard, which is far from being acceptable. Therefore, some imprecision has to be accepted if the analysis shall be performed within reasonable time.

There are several other analysis techniques that heavily depend on alias analysis, e.g. live variables, available expressions or constant propagation. In many cases the results of the alias analysis can be used to increase the precision of the results of the other analysis techniques. Sometimes the results gained by the other analysis techniques may also be used to increase the precision of the alias analysis itself. In such cases it might be useful to iterate this process until the results are as precise as possible, or at least precise enough for the application at hand. As shown in [CC95], even better results can be achieved if some of these techniques are combined and not only iterated. The alias analysis presented here could as well be further improved by combining it with some of the other analysis techniques. However, this will not be discussed in detail here, since it is not of specific relevance to the chosen approach.

There are several different applications for alias analysis and other derived analysis techniques:

- Alias analysis is useful if sequential programs have to automatically be transformed into parallel programs (e.g. [ZC91]). E.g. if one wants to detect if a loop can be executed in parallel or not, it is necessary to find out if and how the different loop iterations depend on each other. In this case a good alias analysis can help to detect a lot more loops which have independent iterations than otherwise. The loop iterations that are found to be independent can then lead to a higher level of parallelism, and hence increase the execution speed of the resulting parallel program.

- Programs are not always only analysed with respect to detecting parallelism, but also to improve their speed or to reduce the space needed. In such cases it is sometimes useful to know that certain functions do not modify some data structures, or that there is no memory location that can be accessed by both of two given variables. Such information can be extracted easily from the graphs that are used to represent the alias relations. In some cases this information is sufficient to perform some simple automatic code optimisations. In any case it can aid the programmer in optimising the program by providing an alias database. The programmer could then make requests to the database, which might prevent him from having to scan huge amounts of code just to find out that it does not modify one of the data structures passed to a particular function.

- Sometimes programs have to be optimised under certain helpful conditions. Partial evaluation analyses a program under the assumption that some of the programs inputs are known (e.g. [And92]). In this case constant propagation is used to find as many of the variables which have known values as possible. Those statements which calculate the values that are known at compile time, can then be removed and hence increase the speed of the resulting program. Here alias analysis can be used to improve the precision of the constant propagation, and hence the partial evaluation will produce better results.

- Further, alias analysis may prove useful during program development to detect errors. E.g. strange effects may occur if a function that is designed to work with non-aliased variables, is invoked with aliased variables. If the programmer does not completely overview the effects of his program he could invoke the alias analysis to have a look at the aliases that may be generated by the program. If he then detects aliases that have not been intended, this could point out an error that might be found more easily then.

The major advantage of the algorithm presented here is that it can deal with almost all the properties of the C programming language. This makes the algorithm applicable for a wide variety of real C programs with only a few restrictions. Many of the other approaches for alias analysis are based on a less problematic language like Fortran where many problems do not even occur because of the language properties. Others analyse C code as well, but make considerable restrictions to the C programming language. They only use a C subset that cannot deal with many real programs, because some commonly used language features are not supported. Our approach is able to deal with all the properties of the C programming language given below. These properties are as well those that are most frequently restricted by other approaches because they cause the major difficulties for the alias analysis.

- pointer (single and multi-level)
- arrays
- structures and unions
- type casting (even between pointers and integral types)
- function calls using function pointers
- variable parameter lists
- dynamic memory allocation
- jumps into other functions (`longjmp`)

For reasons of efficiency our analysis is only call path sensitive, but not fully execution path sensitive. This means that it only keeps track of the function producing a certain may alias, but not of the exact statement. If e.g. one alias may hold in the then-part and another one in the else-part of an if-then-else statement, it is (falsely) assumed that both aliases may hold after the if-then-else statement at the same time. However, our approach is able to avoid such effects if two different functions call another function. In this case the possible aliases of one of the calling functions are not assumed to hold at the same time as the aliases of the other calling function. Nevertheless, the intraprocedural control flow is not left completely unconsidered. Although the intraprocedural analysis parts of the algorithm leave some effects of the control flow unconsidered, this will at least partially be compensated by the preceding transformation of the program into SSA form (SSA = Static Single Assignment). As will be explained in section 3.1 in detail, the computation of the SSA form makes parts of the control flow visible by renaming occurrences of the same variable that do not influence each other.

Basically, there are two different representations used by most of the existing alias analysis algorithms: the first is based on sets of aliased expressions that are propagated through a possibly interprocedural control flow graph (e.g. [Deu94]), whereas the second is based on some kind of memory representation (e.g. [CWZ90]). Our algorithm uses the second concept. So-called function interface graphs, which are a compressed representation of the memory locations used by the program (or more strictly speaking a representation of the values stored at these locations), will be used. During the intraprocedural part of the algorithm such a graph is generated for every function occurring in the program. These graphs are used as a static description of the functions effects, and can later (during the interprocedural analysis parts) be combined to represent the effects of the whole program.

The rest of this work is organised as follows. Chapter 2 gives a detailed overview of the restrictions of the algorithm. Besides defining which programs can and which cannot be analysed, the most important properties of the tools used are explained and some basic terms (attribute grammars) are defined. The central part of this work is chapter 3, where the algorithm which computes the alias information is described. The algorithm has been implemented and tested with real-life programs. The results of these tests are presented in chapter 4. Finally, related and future work will be discussed in chapters 5 and 6.

Chapter 2

Basic concepts

This chapter deals with the basic concepts that have been used to implement the alias analysis algorithm, and that are not specific to the algorithm itself. Further, it gives a detailed overview of which kind of programs can and which cannot be analysed correctly.

2.1 Restrictions for the analysed programs

One of the main aims of the alias analysis presented here is to deal with as many language properties of the C programming language as possible. Unlike many other approaches, the algorithm correctly handles single- and multi-level pointers, type casting as well as structures and unions. Even function calls using function pointers and functions with variable parameter lists can be handled by the algorithm, which makes the analysis applicable for almost every C program. Besides this, the algorithm can handle jump and label statements correctly without restrictions, even if this results in non-reducible control flow graphs. The algorithm does not avoid the problems caused by label and jump statements by transforming the program into an equivalent program without such statements as shown in [EH93]. Such a transformation would lead to some considerable changes in the program structure and may result in comparably unreadable code. Further, it is shown how the correct handling of the `longjmp` function can be integrated into the algorithm.

Although almost everything that might occur in an ANSI conform C program ([Ame89a]) can be handled correctly, it is not possible to properly handle the effects of every possible statement in general. Programs containing one of the following items cannot be analysed correctly.

- functions containing assembler code
- interrupt handling functions (signals)
- variables declared using the **volatile** attribute

Only functions based on one of the above described constructs cannot be analysed properly. This is because such functions heavily depend on the target machine architecture used. Therefore they cannot be handled by an algorithm which is intended to be portable, and hence has to be independent from the architecture currently used. Furthermore, the effects of such functions are not or only partially defined by the ANSI standard (for the same reasons). They should usually be avoided whenever possible to keep programs portable. Nevertheless, the use of these functions is not forbidden, since it is often useful to analyse

```
int a, b, c;
...
c = 0;
for (a = 0; a < b; a = a + 1) c = c + 1;
```

Figure 2.1: 'Hidden' assignments

programs which contain such functions as well. In this case the user has to know that the computed alias sets may be incomplete since the effects of the previously mentioned statements are ignored. However, this leads to valuable results in most cases, because the effects of low-level routines like assembler statements or interrupt calls do not affect many data structures in general.

Besides the above-mentioned language constructs, there are two further problems that are not taken into account by our algorithm.

- aliases based on data read from files or user inputs
- 'hidden' assignments

Although we do not know of any other algorithm that is able to solve such problems, they will nevertheless be discussed in short within the following two sections.

2.1.1 Input data read from files or entered by the user

Data entered by the user or read from files is beyond the control of our analysis algorithm. Usually user or file input is not involved in the process of building aliases. However, it is nevertheless possible to hide aliases from the analysis by writing data to a file and later reading the same data from the file. Such effects are almost impossible to control. If one would like to take such effects into consideration as well, one would have to assume aliases between all values that have been loaded from or saved to a file (including user inputs from `stdin`). This would surely worsen the precision of the analysis extremely, and so it has to be left to the programmer to think about possible aliases caused by user and / or file input and output.

2.1.2 'Hidden' assignments

An example of a 'hidden' assignment is shown in figure 2.1. Here `a` and `c` are both 'assigned' the value of `b`, even though no explicit assignment statement was used. Since it is possible to produce similar effects with arbitrary complicated code it would be necessary to treat every comparison of two values like an assignment, which surely would worsen the analysis results significantly without gaining much.

Both of the above described problems will be ignored by our algorithm since there seems to be no appropriate way to deal with such effects without being overly conservative. Anyway, it is very unlikely that undetectable aliases depending on such effects will occur in programs that have not explicitly been designed to produce such aliases. It is surely not possible to design an efficient and useful alias analysis algorithm that will not fail to find aliases if the programmer intends to trick the algorithm. Hence, it seems to be acceptable to disregard the above two problems.

2.2 SUIF

The alias analysis algorithm that will be described in detail in the following chapters was implemented and tested using the SUIF compiler ([Com94]). The SUIF compiler transforms a C program into its own intermediate format (Standford University Intermediate Format) which is quite similar to three operand instructions as mentioned in [ASU88]. This code can then be used to do a wide variety of things like optimising or analysing the code for different purposes. The major advantage of using the SUIF compiler is that a lot of work like programming the lexical and syntactical analysis passes can be avoided, and it is possible to concentrate on matters of interest right from the start. The different passes of the SUIF compiler are clearly separated, and they communicate only through the files which contain the intermediate code. If necessary, additional information can be annotated to the SUIF code, and so it is fairly easy to add new passes to the compiler wherever necessary. There are two front ends (C and Fortran) as well as back ends available for different target architectures. However, only the front end and another early pass of the SUIF compiler are used here, since it is not intended to generate code for a certain target machine but to find the possible aliases of a program. Once the aliases are known, this information could be used in the following steps to optimise the code or as the basis of further analysis steps. Of course the SUIF compiler cannot compete with other professional compilers in speed aspects, but this was never intended anyway. Its major advantage is its flexibility and the clear interfaces for interaction between the different passes, which makes SUIF a valuable tool for developing new analysis or optimisation strategies.

Another advantage of using the SUIF compiler is that the alias analysis can be performed after the original C program has been transformed into intermediate code, and therefore it is not necessary that every kind of C statement is handled by the alias analysis explicitly. Since the alias analysis processes the intermediate code instead of the original C code, a lot of work can be saved because the SUIF code does not contain as many statement types as C code does. Many C code statements can be substituted by others without changing the semantic, and so the SUIF compiler is able to generate the same intermediate code in some of these cases. As an example `a⇔>b` is equivalent to `(*a).b` and produces the same intermediate code. Besides these quite usual and frequently used shortcuts, there are some more quite curious effects. One such effect is that if `f` is a pointer to a function, then the function calls `f(...)`, `(*f)(...)`, `(**f)(...)` and so on are all equivalent and hence correct. Since all these calls produce the same intermediate code this effect has not to be taken into account by our algorithm. For reasons of simplicity it is assumed that only the second (and probably most vivid) version will be used from now on.

As can be seen from the previously given example, there are certain redundancies in the C code leading to the fact that there are expressions that can be replaced by other expressions. The expression given in the previous example was a simple expression which does not influence the programs control flow. However, there are similar effects for statements which influence the control flow as well. As an example the statement

```
a = (b ? c : d);
```

and the equivalent statement sequence

```
if (b) a = c; else a = d;
```

```
while (cond) {              if (cond) {
  ...                         do {
}                               ...
                              } while (! cond);
                            }
```

Figure 2.2: *WHILE-LOOP* and its corresponding transformation

would both be represented by the same intermediate code. Another useful property of the code produced by SUIF is that it does not contain side-effects within conditions. This is reflected by the fact that

```
if (a = b) ...
```

produces exactly the same intermediate code as if

```
a = b;
if (a) ...
```

had been used instead. Therefore, the later analysis can sometimes omit conditions because they are known to have no side-effects and neither produce new nor destroy existing aliases.

The equivalent statements mentioned up to here always result from syntactical shortcuts provided by the C programming language to make its use more comfortable. Besides these, there is a further transformation that is performed by the SUIF compiler: the **while** loop transformation. This is one of the few 'real' transformations to be effected, since there is no equivalent to the **while** loop contained in the SUIF code. **while** loops are not really necessary since they can (and will) be replaced by a **do** loop nested in an **if** statement as shown in figure 2.2.

Obviously, it would not be a problem to transform **for** loops into **do** loops as well, but this would result in a loss of information that might be useful later. The SUIF compiler transforms **for** loops into **do** loops if they do not meet certain conditions anyway, which results in the fact that those loops that are not transformed are 'real' **for** loops like those used in languages like Fortran ([Ame78]). This means that the loop's body does not change the value of the loop index, and that the number of iterations of the loop depends only on the values of the lower and upper bounds and the step size. These loops might be helpful for parallelisation or other purposes later, and it would be awkward to throw away this additional information.

Although the most important aspects of the SUIF code have now been discussed, the SUIF specific properties should not be relied on wherever possible. In most cases the algorithm is described as if the original C code had been processed. In cases where certain C statements are not handled explicitly, because there are equivalent statements that are already handled by the algorithm, this will be re-stated there.

2.3 Normalisation

The alias analysis is done within four passes: normalisation, SSA form computation, intraprocedural analysis and interprocedural analysis. Although all these passes could be united to a single pass, this is not done for two reasons: first, it is possible to compare the time needed for the extremely cheap first pass with the time needed for the more expensive subsequent passes, and second, because this fits quite well into the concept of the SUIF compiler. By splitting up the passes it is much easier to find out how much time is needed by the alias analysis itself and how much time is consumed by the library functions of the SUIF compiler.

The first pass (normalisation pass) is just a simple traversal of the code, where the following steps will be performed:

- function call normalisation
- return value normalisation
- dead code elimination

These three different normalisation types as well as their purpose will now be explained in the following sections.

2.3.1 Function call normalisation

The function call normalisation replaces non-variable expressions by temporary variables and a preceding assignment of the non-variable expression to that temporary variable. This is effected for all function typed expressions and function parameters. Furthermore, function calls within non-trivial assignments are replaced by temporary variables and a corresponding assignment.

Given a certain function call there may be all of the above-described expression types that may have to be replaced: the function typed expression, the expressions passed as parameters and the function call itself. As already mentioned before, the first two kinds of expressions will only be replaced by temporary variables if they are non-variable expressions. Accordingly, the function calls are only replaced if they occur as a real subexpression.

For parameter expressions this means that they have to be either compound expressions like ⇔a or $\texttt{a} + \texttt{b}$ or constants. Since the C programming language does not allow function variables, non-variable function typed expressions can only be replaced by function pointers and so they need particular handling here. According to section 2.2, function calls using function pointers are assumed to be written as $(*\texttt{f})(\ldots)$ and not as $\texttt{f}(\ldots)$ where `f` is a pointer to a function. Therefore, the outermost pointer dereference operation may be ignored in such cases, and so the expression is no longer a non-variable expression. However, expressions like $(*(\texttt{f} + \texttt{1}))(\ldots)$ still need to be replaced by temporary variables, since, even if the outermost pointer dereference operation is ignored which leads to the expression $\texttt{f} + \texttt{1}$, such expressions are still non-variable expressions.

To make things clearer the code shown in figure 2.3 shows a function call before and after it has been normalised. Here `fpa` is an array of pointers to functions. This array is used in the function call contained in the last line of the source code. As can be seen, the function call contains two compound expressions ($(*(\texttt{fpa}[\texttt{n}]))$ and $\texttt{y1} + \texttt{y2}$). After the code has been transformed, three additional variables have been added to hold the values of the compound expressions as well as the functions result value. The first one (`tmp1`) is used to hold the function pointer, the second one (`tmp2`) is used to hold the value of the non-variable first parameter expression ($\texttt{y1} + \texttt{y2}$) and the third one (`tmp3`) holds the functions return value. So, finally, all instructions which contain the function calls of the transformed code do not contain any compound expression besides the function call and the assignment itself.

Source code:

```
int (*(fpa[5]))(int p1, int p2);
int n, m, x, y1, y2, z;

...



x = (*(fpa[n]))(y1 + y2, z) + m;
```

Transformed code:

```
int (*(fpa[5]))(int p1, int p2);
int n, m, x, y1, y2, z, tmp2, tmp3;
int (*tmp1)(int p1, int p2);
...
tmp1 = fpa[n];
tmp2 = y1 + y2;
tmp3 = (*tmp1)(tmp2, z);
x = tmp3 + m;
```

Figure 2.3: Sample code before and after the function call normalisation

Source code:

```
int diff(int x, int y)
{

  if (x > y) {

    return (x - y);
  } else {

    return (y - x);
  }


}
```

Transformed code:

```
int diff(int x, int y)
{
  int tmp_ret;
  if (x > y) {
    tmp_ret = (x - y);
    goto end;
  } else {
    tmp_ret = (y - x);
    goto end;
  }
end :
  return tmp_ret;
}
```

Figure 2.4: Sample code before and after the return value normalisation

2.3.2 Return value normalisation

Like the function call normalisation, the return value normalisation replaces compound expressions by temporary variables and corresponding assignments. This is done for all arguments of **return** statements. Besides the replacement of the compound expressions, the implicit control flow of the **return** statements is made visible by replacing the **return** statements by **goto** statements. Every function is transformed in a way so that there always is only one single **return** statement at the end of the function. This can be done by replacing all **return** statements within the code by **goto** statements that jump to a newly introduced label preceeding the final **return** statement. The example shown in figure 2.4 contains a function and the corresponding normalised function.

The return value normalisation together with the function call normalisation proves useful during the later analysis passes, since every value passed between a calling and a called function, as well as the function itself, can be associated with a variable after the normalisation has been performed. Additionally, the return value normalisation makes **return** statements irrelevant to the control flow since they always only occur as the final statement of a function. Hence, it is sufficient to handle the control flow relevant effects of the **goto** statements which have to be handled anyway.

2.3.3 Dead code elimination

Now that the replacement of more or less complex expressions by temporary variables has been sufficiently dealt with, there is only one further thing that is done during the normalisation pass and which has been left unexplained so far: the dead code elimination. The dead code elimination, that is part of the normalisation pass, removes only structural dead code. A definition of structural dead code is given in the following.

Definition 1 (structural dead code)
Structural dead code is code that cannot be reached during any program execution, even if it is assumed that all the conditions of the program have arbitrary values, which do not depend on the current values of the used variables.

E.g. an assignment following a **break** or **goto** statement will never be executed as long as it is not preceded by a label. Such statements can safely be removed from the program without changing the programs semantic. Later the SSA computation will benefit from the fact that the structural dead code has been eliminated in advance.

2.3.4 Time and space bounds

Since the normalisation is a fairly simple process which just goes through the code once, the time as well as the space needed for the complete normalisation pass grows linearly to the size of the code that is analysed[1]. This is also reflected by the empirical results that are described in chapter 4.2.

2.4 Attribute grammars

Attribute grammars were first mentioned by Knuth in [Knu68]. Later they were used and extended by several other authors (e.g. [ASU88, Gou88, Kai89, KW92, RTD83]). The definition of attribute grammars given below is similar to the extended version used in [KW92]. Attribute grammars will later (in section 3.2) be used to define the intraprocedural part of the alias analysis algorithm.

Definition 2 (context-free grammar)
A context-free grammar is quadruple $G = (T, N, S, P)$, where

T	is the set of terminal symbols,
N	is the set of non-terminal symbols,
S	is the start symbol ($S \in N$),
P	is a set of productions and

$$\forall(p \in P) \quad : \quad p = (\alpha_p \rightarrow \beta_p) \text{ where } \alpha_p \in N \text{ and } \beta_p \in (T \cup N)^*.$$

Definition 3 (reduced grammar)
A grammar is said to be reduced, if every production occurs in at least one derivation of a terminal word from the start symbol.

$$\forall((\alpha \rightarrow \beta) \in P) \quad : \quad \exists(w \in T^*, v_1, v_2 \in (T \cup N)^*) \quad : \quad S \overset{*}{\rightarrow} v_1 \alpha v_2 \rightarrow v_1 \beta v_2 \overset{*}{\rightarrow} w$$

[1]This is the minimum for every pass processing every statement of the program.

Definition 4 (attribute)
An attribute is a tuple (id, v) where id is the name of the attribute and v is the attributes value domain.

Definition 5 (attribute grammar)
An attribute grammar is a quadruple $AG = (G, A, AC, PC)$, where

$$\begin{aligned}
G &= (T, N, S, P) \text{ is a reduced context-free grammar,}\\
A &= \bigcup_{\alpha \in T \cup N} A(\alpha) \text{ is a finite set of attributes}^{2},\\
AC &= \bigcup_{p \in P} AC(p) \text{ is a finite set of attribute computations and}\\
PC &= \bigcup_{p \in P} PC(p) \text{ is a finite set of plain computations.}
\end{aligned}$$

In the definition given above $A(\alpha)$ denotes the set of attributes which are associated with each symbol α. Since a symbol can appear more than once within a single production it is necessary to differ between the attributes of a symbol and the corresponding attribute occurrences.

Definition 6 (attribute occurrence)
Given a production $p \in P$, with $p = \alpha_0 \rightarrow \alpha_1 \ldots \alpha_n$ and $\alpha_i \in (T \cup N)$ for all $(0 \leq i \leq n)$, then $\alpha_i.a$ is an attribute occurrence if $a \in A(\alpha_i)$.

There are three different kinds of attributes: inherited, synthesised and intrinsic attributes. Accordingly the set of attributes (A) can be split up into the three sets A_{inh}, A_{syn} and A_{int} containing the inherited, synthesised and intrinsic attributes ($A = A_{inh} \cup A_{syn} \cup A_{int}$).

Definition 7 (inherited, synthesised and intrinsic attributes)
Given a production $\alpha \rightarrow \gamma_1 \ldots \gamma_m \beta \gamma_{m+1} \ldots \gamma_n$ $(n, m \in \mathbb{N}_0)$, with $\alpha, \beta \in N$ and $\gamma_1, \ldots, \gamma_n \in (T \cup N)$, an attribute x of β is said to be inherited ($x \in A_{inh}(\beta)$), if it is computed by a function depending on the attributes of α and γ_1 to γ_n as well as on the intrinsic and inherited attributes of β itself.

$$\begin{aligned}
&\beta.x = f(p_1.a_1, \ldots, p_s.a_s) \text{ with } s \in \mathbb{N}_0 \text{ and}\\
\forall (1 \leq i \leq s) \ \ : \ &((p_i \in \{\alpha, \gamma_1, \ldots, \gamma_n\}) \wedge (a_i \in A(p_i))) \ \vee\\
&((p_i = \beta) \wedge (a_i \in (A_{int}(\beta) \cup A_{inh}(\beta))))
\end{aligned}$$

Given a production $\alpha \rightarrow \beta_1 \ldots \beta_n$ $(n \in \mathbb{N}_0)$, with $\alpha \in N$ and $\beta_1, \ldots, \beta_n \in (T \cup N)$, an attribute x of α is said to be synthesised ($x \in A_{syn}(\alpha)$), if it is computed by a function depending on the attributes of β_1 to β_n as well as on the attributes of α itself.

$$\begin{aligned}
&\alpha.x = f(p_1.a_1, \ldots, p_s.a_s) \text{ with } s \in \mathbb{N}_0 \text{ and}\\
\forall (1 \leq i \leq s) \ \ : \ &(p_i \in \{\alpha, \beta_1, \ldots, \beta_n\}) \wedge (a_i \in A(p_i))
\end{aligned}$$

[2]It is assumed that the attribute names are unique. This allows to identify the symbol to which the attribute belongs by the attribute's name. In fact this is no real restriction since an attribute grammar which does not meet this condition can be transformed into one which meets the condition by combining the attribute names with the symbols.

An attribute x of α is said to be intrinsic ($x \in A_{int}(\alpha)$), if it does not depend on any other attribute. It's value is computed by earlier passes of the analysis and therefore available before the attribute grammar computations are carried out.

In general there are no restrictions concerning the dependencies of the different grammar attributes. However, this can cause the computation of the attributes to become rather expensive or even impossible if there are circular dependencies. L-attributed grammars ([ASU88, Gou88]) are attributed grammars which restrict these dependencies in a way that ensures that the computation of the attributes can be done in a single depth-first-left-to-right walk through the syntax tree.

Definition 8 (L-attributed grammars)

L-attributed grammars are attributed grammars with the following restrictions:

- The inherited attributes of a symbol may only depend on inherited attributes of its parent, on synthesised attributes of its left siblings in the syntax tree or on intrinsic attributes.
- There are no cyclic dependencies between the attributes for a single symbol.

Note that attribute grammars do not specify the order in which the computations of the attributes have to be performed. Although there are dependencies between the different attributes which influence the order in which the attributes have to be computed, these dependencies do not in general lead to a total order. Therefore the order has to be specified explicitly later.

Now, after what kind of alias information will be computed has been explained, we are ready for the details of the alias computation itself. The basic concepts that will be used in the next chapter to illustrate the algorithm have been given, and which properties of the C programming language can and which cannot be dealt with have been described.

Chapter 3

Alias computation

The alias computation consists of three major parts that have already been mentioned in the previous chapter: the SSA computation as a basic code transformation to increase the precision of the analysis results (section 3.1), the intraprocedural analysis summarising the effects of a single function (section 3.2) and finally the interprocedural analysis combining the results of the different functions (section 3.3).

$$\begin{tikzcd}
code \arrow[r] & \boxed{\begin{array}{c}\text{SSA}\\\text{computation}\end{array}} \arrow[r] & code_{SSA} \arrow[r] & \boxed{\begin{array}{c}\text{intraprocedural}\\\text{analysis}\end{array}} \arrow[r] & \begin{array}{c}code_{SSA}\\+\\G_{f_1},\ldots,G_{f_n}\end{array} \arrow[r] & \boxed{\begin{array}{c}\text{interprocedural}\\\text{analysis}\end{array}} \arrow[r] & G_{all}
\end{tikzcd}$$

As can be seen from the above given graph, the SSA computation is a certain form of code transformation. The initial program code (*code*) is normalised and transformed into another piece of code ($code_{SSA}$) which has some helpful properties. The intraprocedural analysis then takes this transformed code to produce graphs representing the effects of each function ($G_{f_1}, \ldots, G_{f_n}$). Finally, the interprocedural analysis takes the transformed code as well as the previously computed graphs and builds a new combined graph (G_{all}). This graph then contains all the information necessary to find out if two variables may be aliased or not.

Although the algorithm that is described in the following sections is able to handle almost all the problems arising from the use of the C programming language, there are some aspects that are not taken into account during the discussion of the above described three passes. This is done to be able to concentrate on the essentials and to avoid the many exceptions that are necessary to handle some of the less interesting features of the C programming language. These topics will then be dealt with in section 3.4 where it is shown how they can be integrated into the algorithm as well.

3.1 SSA form

A function is said to be in SSA form if every location where a variable is used can be reached by exactly one location where that variable is defined. If the function does not contain any point where two different definitions of a variable reach a point where that variable is used, there is no transformation necessary, since the program is then already in SSA form. However, any non-trivial function contains statements like `if`, `while` or `goto` which lead to different definitions reaching a point where the variable is used in almost every case. If such a function has to be transformed into SSA form, this can be done by renaming the variables using subscripts and adding so-called ϕ-functions at places where

```
a = 5;               a_1 = 5;
b = 5;               b_1 = 5;
c = 5;               c_1 = 5;
p = &c;              p_1 = &c_1;
if (...) {           if (...) {
  a = 6;               a_2 = 6;
  b = 7;               b_2 = 7;
} else {             } else {
  b = 6;               b_3 = 6;
  *p = 8;              *p_1 = 8;
}                    }
d = a + b + c;       a_3 = φ(a_1, a_2)
                     b_4 = φ(b_2, b_3)
                     d_1 = a_3 + b_4 + c_1;
```

Figure 3.1: SSA form for an if-then-else statement

two different reaching definitions meet. Figure 3.1 presents a simple example that shows where the ϕ-functions will be placed in case of an if-then-else statement. As can be seen in the example, it is not necessary to generate ϕ-functions for all the variables occurring in the code. E.g. there is neither a ϕ-function for `c` nor one for `p`, since there is no new value assigned to these variables inside the if-then-else statement. Accordingly there is only one definition which reaches the end of this statement, and so a ϕ-function would be superfluous.

Another important property of the SSA form can be seen if one looks at the variable `p`. The assignment statement `*p = 8` does not get a new subscript since only the value pointed to by `p` is changed instead of `p` itself. This leads to the fact that the more assignments are made through pointers the less benefits can be gained by transforming the program into SSA form. In the worst case, where all but the initialising assignments of a program are made through pointers, no helpful information can be gained by transforming the program into SSA form, since it then already has the properties desired by the SSA form. In this case there is always only one (initialising) assignment for every variable. Although it is possible to modify any given program in a way that the SSA form computation will be useless[1], it is not very likely that such programs will ever occur in real life.

The computation of SSA form is the basis of the presented alias analysis algorithm. It was first mentioned in [RWZ88, AWZ88], and was later used as the basis for several program optimisation problems. There are various different approaches to compute SSA form more efficiently and to get better average or worst case time or space bounds ([CFR+89, CFR+91, CF95]). The algorithm presented here was chosen with respect to ease of implementation, but it could be replaced by any other (faster) algorithm if necessary. There are already several good algorithms to compute SSA form and it is not the subject of this work to try to improve them. Anyway, the computation of the SSA form itself is only of minor interest for this work, and so there were only little changes made to the above-mentioned algorithms to deal with the particular situation found in our setting.

Note, that the SSA form used here is built as an initial step of the alias analysis and, hence, does not take pointer induced aliases into account. Although that may result in programs where more than one (indirect) definition reaches a use, we refer to the result

[1]E.g. this could be done by replacing every declaration of a variable v having type t by the declarations of v' which has type t and v which has type $t*$. Additionally, v has to be initialised to hold the address of v' ($v = \&v'$), and every further occurrence of v in the program code has to be replaced by $*v$.

of this step as being in SSA form. A detailed discussion of this problem can be found in [HH98].

Before going into the details of the SSA form computation, the SSA variables itself will be defined.

Definition 9 (SSA variables)

Given the set of variables used in the original program (VAR) then the set

$$VAR_{SSA} = (VAR \times \mathbb{N}_0)$$

is the set of SSA variables. A SSA variable is a pair containing a variable of the original program and a number, the SSA value. These components can be accessed by the two functions *base_var* and *ssa_val* which are defined as follows.

$$base\_var \ : \ VAR_{SSA} \twoheadleftarrow\!\!\!\twoheadrightarrow VAR \ , \ ssa\_val \ : \ VAR_{SSA} \twoheadleftarrow\!\!\!\twoheadrightarrow \mathbb{N}_0$$

$$\begin{aligned} base\_var((v,n)) &= v \\ ssa\_val((v,n)) &= n \end{aligned}$$

As already done in the preceding example, a notation differing from the one used in the definition of the SSA variables will be used. In most cases $(\mathtt{v},\mathtt{n})$ will be written as $\mathtt{v}_\mathtt{n}$. Furthermore SSA variables will frequently be referred to as variables. They will only be called SSA variables if it is not exactly clear what is meant due to the context.

3.1.1 CFG graphs

To compute the SSA form for a given function the algorithm first builds a control flow graph (CFG).

Definition 10 (CFG graph)

A control flow graph $G = (V, E)$ is a directed graph whose finite number of nodes represent parts of the code that are executed in sequential order. A directed edge $(n_s, n_d) \in E$ in this graph represents the flow of control and indicates that under certain circumstances[2] the destination node (n_d) will be executed after the execution of the source node (n_s). Besides that, there are two special nodes: *START* and *END*. *START* does not have any predecessors and *END* has no successors. An edge exists from the *START* node to every node where the execution of the program can begin, and there is another edge from every node where the execution may stop to the *END* node.

As can be derived from the above given definition, the following simple fact holds.

Fact 11

There is no non-trivial control flow contained in the code which belongs to a certain CFG node, and hence the code is a so-called basic block.

The term 'basic block' will be properly defined soon. Figure 3.2 contains a simple example program, and figure 3.3 shows its corresponding CFG graph. The numbers in parentheses at each line of the code refer to the number of the corresponding node in the CFG graph.

```
a = 0;             (1)
b = 0;             (1)
do {               (2)
  if (...) {       (3)
    a = a + 1;     (4)
  } else {
    b = b + 1;     (5)
  }
  c = c + 1;       (7)
} while (...);     (7)
```

Figure 3.2: Simple program with non-trivial control flow

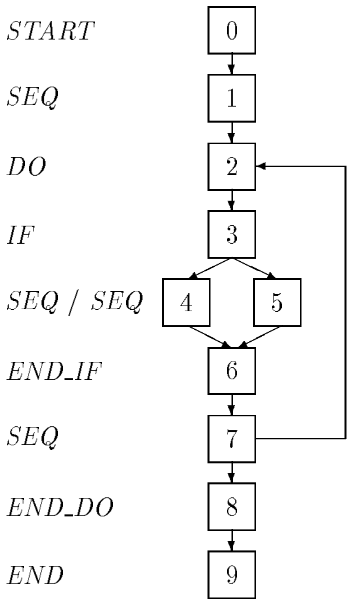


Figure 3.3: Control flow graph of a simple program

Given a control flow graph $G = (V, E)$, where V is the set of CFG nodes and E is the set of edges, the function *succ* returns the set of all immediate successors of a given CFG node. The set of all immediate predecessors of a CFG node is returned by the *pred*-function.

Definition 12 (successors and predecessors)

$$succ \;:\; V \Leftrightarrow\!\!\rightarrow \mathbb{P}(V) \;\;,\;\; pred \;:\; V \Leftrightarrow\!\!\rightarrow \mathbb{P}(V)$$

$$\begin{aligned} succ(v) &= \{v' \in V \mid (v, v') \in E\} \\ pred(v) &= \{v' \in V \mid (v', v) \in E\} \end{aligned}$$

Here $\mathbb{P}(V)$ denotes the set containing all subsets of V. The function *path* returns the set of all paths from the first node to the second (a path is represented by the sequence of the visited nodes, i.e. words over V^*). Note that this set will contain infinitely many paths if there are cycles on the way from the first node to the second node.

Definition 13 (paths)

$$path \;:\; V \times V \Leftrightarrow\!\!\rightarrow \mathbb{P}(V^*)$$

$$\begin{aligned} path(n_{src}, n_{dst}) = \{n_1 \ldots n_k \in V^* \mid\; & (n_1 = n_{src}) \;\wedge\; (n_k = n_{dst}) \;\wedge \\ & (\forall (1 \le i \le (k \Leftrightarrow 1)) \;:\; n_{i+1} \in succ(n_i))\} \end{aligned}$$

It is sometimes interesting to know how long a certain path is. Therefore there exists a function *length* which takes a path as its input and returns the length.

Definition 14 (length of a path)

$$length \;:\; V^* \Leftrightarrow\!\!\rightarrow \mathbb{N}_0$$

$$length((n_1 \ldots n_k)) = k \Leftrightarrow 1$$

3.1.1.1 CFG node types

Besides the two special nodes *START* and *END* that are not related to the program code, there are other nodes which represent one or more statements of a program. There are eight different types of CFG nodes which refer to the statement(s) they represent. The different node types are explained in the following box.

SEQ	:	Sequential nodes (without any non-trivial control flow inside)
IF	:	Start node for *IF-THEN-ELSE* statements
END_IF	:	End node for *IF-THEN-ELSE* statements
FOR	:	Start node for *FOR-LOOP* statements
END_FOR	:	End node for *FOR-LOOP* statements
DO	:	Start node for *DO-LOOP* statements
END_DO	:	End node for *DO-LOOP* statements
CALL	:	Node representing function calls (*CALL* statements)

[2]This means that in case of a branch node the corresponding condition evaluates to a certain value. This does not ensure that this will really happen during program execution, which leads to the fact that there might be branches that will never be executed.

Note that the type of a CFG node does not play any role for the SSA computation itself. However, it is nevertheless useful to introduce the different node types, since it becomes much easier to understand and explain the computation of the SSA form then. The node types make it much easier to find out which part of the program code belongs to a certain CFG node if a program and its corresponding CFG graph are given.

As already mentioned before, there are sequences of statements related to a certain CFG node. These sequences are always so-called basic blocks.

Definition 15 (basic blocks)
A basic block is a sequence of statements that are always executed in the order of their textual appearance in the program.

Note that there are no CFG nodes for *LABEL* and *JUMP* statements, since these statements are implicitly handled by generating edges from the nodes containing the *JUMP* statements to those containing the corresponding *LABEL* statements. *LABEL* statements are always at the beginning of a nodes basic block, whereas *JUMP* statements (conditional or unconditional) are always the last statement belonging to a nodes basic block. This means that a basic block contains at most one *JUMP* statement (at the end) and that if there is more than one *LABEL* statement contained in a basic block, they are placed in front of all other statements[3]. There are no *LABEL* or *JUMP* statements between the other statements of a basic block. This ensures that control flow always enters a basic block at the first (non-label) statement and leaves it after the execution of the last statement.

The CFG graph that is being built is based on the high-level intermediate code that is generated by the SUIF compiler. As explained in chapter 2, one of the advantages of using this intermediate code is that it is not necessary to handle each kind of C statement separately. To make it easier to distinguish between the C statements and the SUIF statements, they are printed in different styles. E.g. **if** is a C statement, whereas *IF-THEN-ELSE* is a SUIF statement.

The following list gives an overview of the SUIF intermediate code statements that are relevant to the control flow. All other statements are sequential and not of any interest for building the CFG graph.

- *IF-THEN-ELSE*
- *FOR-LOOP*
- *DO-LOOP*
- *CALL* (function calls)
- *LABEL*, *JUMP* (conditional or other)

Note that there exists no *WHILE-LOOP* since they can (and will) be replaced by a *DO-LOOP* inside an *IF-THEN-ELSE* by the SUIF compiler as explained in chapter 2.

3.1.1.2 Sequential statements (*SEQ*)

Sequential statements do not influence the flow of control, and there can be several of them which belong to a single sequential node (basic block). In most cases sequential nodes have

[3]It does not seem to make much sense to have two *LABEL*s marking exactly the same location, but this situation can arise due to preprocessor commands like `#ifdef` hiding parts of the code.

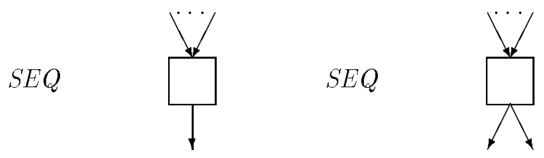


Figure 3.4: Control flow graphs for sequences of sequential statements

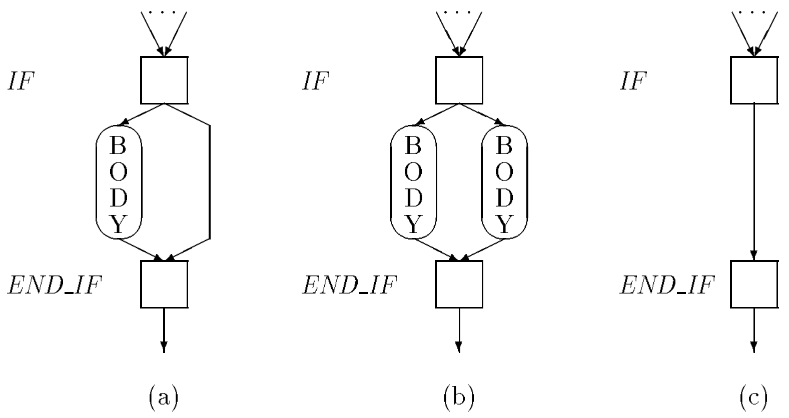


Figure 3.5: Control flow graphs for IF statements

only one incoming and one outgoing edge in the flow graph. Only if the code belonging to the sequential node starts with a *LABEL*, may there be several incoming edges. However, there will never be more than two outgoing edges for a sequential node (figure 3.4). Two outgoing edges occur if the last statement belonging to a sequential node is a conditional *JUMP* statement. In this case the program continues either at the next statement which occurs in the function (condition was false) or after the corresponding *LABEL* statement (condition was true). Note that the C programming language has no direct equivalent to the conditional *JUMP* statements of the SUIF code. A conditional *JUMP* statement is used by SUIF when an **if** combined with a **goto** statement occurs in the C code.

3.1.1.3 *IF-THEN-ELSE* statements

The CFG graph for an *IF-THEN-ELSE* statement with a non-existing (or empty) else-part is shown in figure 3.5 (a). If the then-part as well as the else-part of the *IF-THEN-ELSE* statement is empty, there will be no nodes besides the *IF* and the *END_IF* nodes and only one edge connecting the *IF* node with the *END_IF* node as shown in figure 3.5 (c). Obviously it does not make much sense to use *IF-THEN-ELSE* statements with empty bodies, since their only effect is to evaluate the condition, which could be easier done by using the condition as a statement instead. Nevertheless they have to be handled correctly because the ANSI standard allows them, and they do not only occur in senseless programs. E.g. they could be caused by preprocessor directives that ignore the code contained in the body under certain circumstances. Figure 3.5 (b) shows a standard *IF-THEN-ELSE* statement with non-empty then- and else-parts. If the then-part is empty and the else-part is not, the graph that is generated is the same as in the opposite case (figure 3.5 (a)).

Note that there are no control flow edges which connect a node outside of an *IF-THEN-ELSE* statement with a node in its body. This results from the fact that the SUIF compiler does not generate *IF-THEN-ELSE* statements for **if** statements containing **goto** or **label** statements that would lead to such a kind of graph. In these cases (jumps into or out of the body of an **if** statement) the SUIF compiler transforms the **if** statement into a semantically equivalent sequence of statements using *JUMP* and *LABEL* statements.

Nevertheless there can be *JUMP* and *LABEL* statements in the body if the source and the target of these jumps are both inside the body. This leads to a special case that may cause the generation of graphs that are different from the ones discussed above. These graphs occur if the last statement of the body is an unconditional *JUMP*. In this case it is not possible to reach the *END_IF* node and there will be no edge which connects the body with the *END_IF* node then. If both the then- and the else-part of an *IF-THEN-ELSE* statement are terminated by an unconditional *JUMP*, the *END_IF* node would be unreachable and will therefore not be generated. In general such *IF-THEN-ELSE* statements are senseless, since the program will never terminate after entering such a statement.

3.1.1.4 *FOR-LOOP* statements

The CFG graphs for *FOR-LOOP* statements are shown in figure 3.6. The first graph (figure 3.6 (a)) represents an usual *FOR-LOOP* without **break** or **continue** statements. Figure 3.6 (b) shows the graph that will be generated if the *FOR-LOOP* contains **break** and **continue** statements. The graph for a *FOR-LOOP* always starts with a sequential node preceding the *FOR* node. This additional node is necessary because **for** loops start by assigning a new value to the loop index, before they start to check the condition, execute the loop's body and increment the loop index repeatedly. This means that the code for the initial assignment cannot be attached to the *FOR* node, since this node is part of the circle that represents the loop's iterations. The graph shown in figure 3.6 (b) represents a loop with two **continue** and two **break** statements. Usually these statements occur only inside of **if** or **case** statements because they would either be superfluous[4] or otherwise produce dead code[5]. For the graph this means that the nodes containing the **break** statements are connected with the *END_FOR* node (arrows to the left), and the nodes containing the **continue** statements are connected to the *FOR* node (arrows to the right). Like the CFG graphs for *IF-THEN-ELSE* statements the graph for a *FOR-LOOP* statement will change if the last statement in the body is an unconditional *JUMP* (**break** or **goto**). In this case the edge connecting the last node of the body with the *FOR* node will not be generated. Note that **continue** statements do not have any effect if they occur as the last statement in the body. Since **continue** and **break** statements are only a special kind of **goto** statements, and will therefore be transformed into *JUMP* statements, they are always the last statement belonging to a CFG node.

FOR-LOOP statements with empty bodies do not occur since the SUIF compiler replaces them by a simple assignment statement that sets the index variable to its final value. In some cases **for**-loops with empty bodies occur in ("ingenious") C programs. E.g. if there are side-effects due to function calls in the loops condition that already do all the necessary computations of the loop. In such cases there is no need for further statements in the loops body. Nevertheless the above-mentioned transformation can be safely done, since the SUIF compiler only generates *FOR-LOOP*s if the loop's upper and lower bounds as well as the step size are known at compile time. Those **for**-loops containing the described side-effects will be transformed into *DO-LOOP* statements, and so it is not necessary to take them into consideration here.

[4]if they occur at the end of the loop

[5]if they occur elsewhere and they are not followed by a *LABEL* that has a corresponding *JUMP* statement somewhere

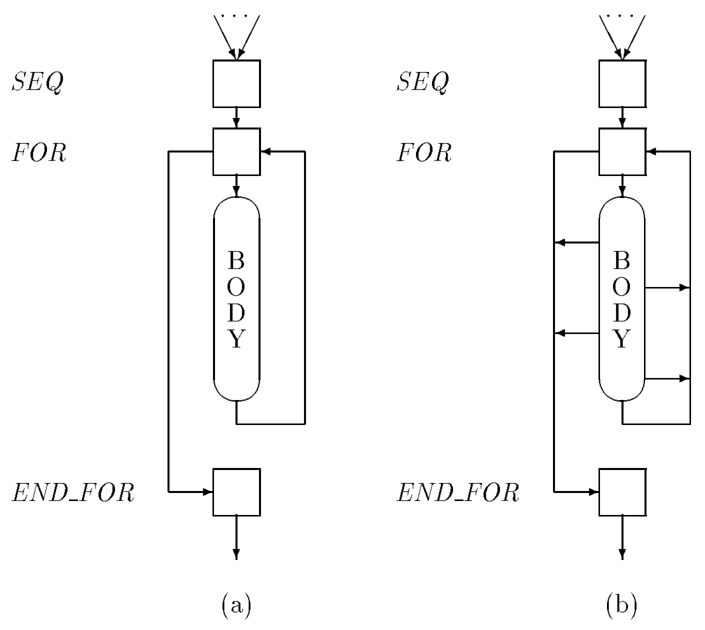


Figure 3.6: Control flow graphs for *FOR-LOOP*s

The code belonging to the *FOR* node is assumed to increment the loop index first and to check the condition afterwards[6], even though this is not exactly the same as what happens during the execution of a **for**-loop. If one wants to represent a **for**-loop correctly, it would be necessary to add another node for the increment statement. However, this is not done to keep the graph smaller. One could change the initial assignment to the loop index (by subtracting the step size) to make sure that the semantic is not changed, but even this is not necessary, because the absolute values of the loop bounds and the step size do not influence the computation of the SSA values anyway.

For the same reasons that have already been mentioned in relation to the bodies of *IF-THEN-ELSE* statements, there are no connections from nodes inside the body of the *FOR-LOOP* to nodes outside the *FOR-LOOP*.

3.1.1.5 *DO-LOOP* statements

DO-LOOP statements differ from *FOR-LOOP* statements in two aspects: they always contain one sequential node for the termination condition and the body of the loop is executed at least once. Figure 3.7 (a) shows the CFG graph for a loop without **break** or **continue** statements. Note that if the body of the loop contains no non-trivial control flow, the sequential node contains the code of the body as well as the code for the termination condition. In this case there are no further nodes necessary and the sequential node is an immediate successor of the *DO* node. In case the loop contains **break** and **continue** statements the graph appears as shown in figure 3.7 (b). In this example there is one **break** and one **continue** statement. The node containing the **break** statement is connected to the *END_DO* node, whereas the node containing the **continue** statement is connected to the sequential node containing the code checking the termination condition. The body of a *DO-LOOP* never contains jumps to targets outside the loop, and vice versa there are no jumps from outside the loop into the body of the loop. This is not really necessary for the computation of the SSA values, but the typical properties of a loop will be lost if

[6]This has to be checked as the second step in order to satisfy the basic block condition.

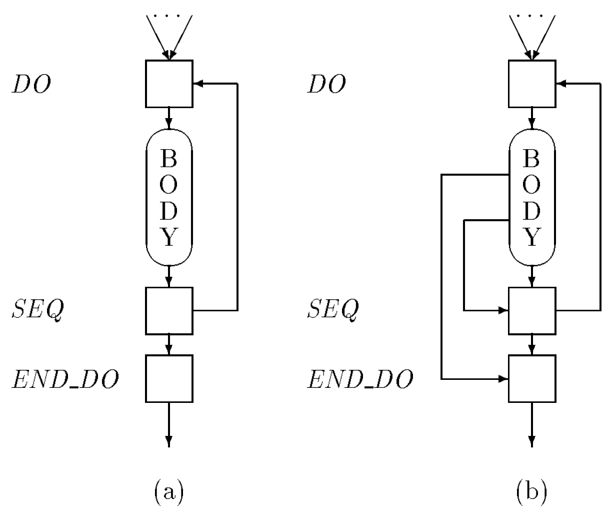


Figure 3.7: Control flow graphs for *DO-LOOP*s

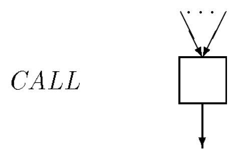


Figure 3.8: Control flow graph for function calls

there are arbitrary jumps into and out of the loop. If there are jumps into or out of a **for**-, **do**- or **while**- loop the SUIF compiler translates the degenerated loop into a semantically equivalent sequence of statements using *JUMP* and *LABEL* statements.

3.1.1.6 Function calls (*CALL*)

For every function call contained in the code there will be a corresponding *CALL* node in the CFG graph as well. Figure 3.8 shows such a node. It is essential to have these nodes since a function call can change the value of any global variable[7] as well as the values of local variables whose addresses have been taken before. These effects will be taken into account during the interprocedural part of the alias analysis. There will be no ϕ-functions generated for function calls, since the computation of the SSA values is purely intraprocedural, and it is not known which variables may be affected by the execution of the called function when the computation of the SSA values is in progress.

3.1.1.7 *LABEL* and *JUMP* statements

There are no nodes in the CFG graph to represent *LABEL* and *JUMP* statements. They are represented implicitly by sequential or other nodes and their incoming and outgoing edges[8]. Since there is no non-trivial control flow allowed inside sequential nodes it is necessary to

[7]Here global variable means any variable that is alive during the whole execution of the program. This includes all variables that are declared using the **static** attribute, even if they are declared inside a function. They are stored in the same way as all the other global variables anyway, and the only difference is that their names are not known in the whole program.

[8]*LABEL* statements can as well be the first statements belonging to nodes like *IF* or *CALL*, whereas *JUMP* statements always belong to a sequential node.

```
  ...
lab1 :                    (0)
  if (...) goto lab3;     (0)
lab2 :                    (1)
  a++;                    (1)
  goto lab1;              (1)
lab3 :                    (2)
  b++;                    (2)
  if (...) goto lab2;     (2)
  ...
```

Figure 3.9: Program containing unstructured control flow

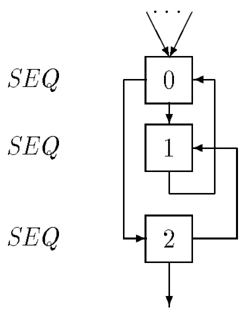
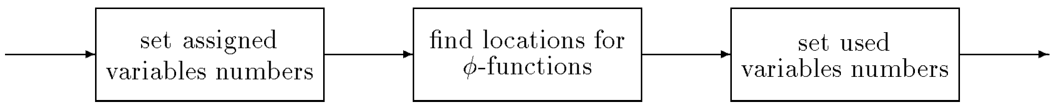


Figure 3.10: Control flow graph for program containing unstructured control flow

generate a new node whenever a *LABEL* occurs in the code. The same happens after conditional or unconditional *JUMP* statements. Figure 3.9 shows a small example containing conditional and unconditional *JUMP* statements[9]. Again the numbers in parenthesis are the numbers of the corresponding nodes in the CFG graph. The CFG graph corresponding to this example is shown in figure 3.10.

3.1.2 Computation of SSA values

As can be seen from the graph shown below, the computation of the SSA values is done within three major steps. Before going into the details, a short overview of these steps will be given.


set assigned variables numbers
find locations for ϕ-functions
set used variables numbers

First Step: During the first step all variables that are assigned a new value (assigned variables) receive an unique SSA value. Additionally, track is kept of the variables that are re-assigned for each single CFG node.

[9]Note that the '**if** (...) **goto**' statements are transformed into a single conditional *JUMP* statement, and that there will be no *IF-THEN-ELSE* statements generated by the SUIF compiler in this case.

```
a₁ = 1;
b₁ = &a₁;
*b₁ = 2;
/* now a₁ == 2 ! */
c₁ = a₁;
```

Figure 3.11: Program with side-effect in SSA form

Second step: The second step takes the CFG graph and the collected assigned variables to compute the places where ϕ-functions are necessary. Furthermore, the SSA values for the left hand side of the ϕ-functions are set.

Third step: The third step computes the SSA values for the remaining non-numbered occurrences of variables (used variables), and the right hand sides of the ϕ-functions are completed by adding the variables of the reaching definitions.

The computation of SSA values is done for every function in a program without considering the effects of function calls that are contained in the code. Since the SSA form only ensures that there is always only one assignment which reaches a use of a variable, and not that the value of the used variable is the same value that was assigned to it by the assignment reaching this use, it is not necessary to insert ϕ-functions for every variable that might be affected by the function call. As can be seen in the program in figure 3.11, it is possible that the value of a variable (a) has changed due to side-effects, even if there is no new SSA value introduced for this variable. Nevertheless, there is only one reaching definition ($a_1 = 1$) for that variable even though its value has already been changed when the use ($c_1 = a_1$) is reached. The last assignment uses the variable (a_1) even though the value (2) is not the same value that was assigned to a_1 by the first assignment (1).

Anyway, it would extremely increase the number of generated ϕ-functions, if there had to be ϕ-functions for every variable that might be influenced by a function call. Since every function call can change almost every variable, namely every global variable and every local variable whose address has been taken, this would lead to a huge amount of ϕ-functions for function calls without producing better results. Obviously it is not possible to consider these effects during the computation of the SSA values, since it would be necessary to know about all side-effects and aliases of a statement at this early state of the analysis. These effects will be taken into account in the later (interprocedural) passes of the alias computation. For the same reason the side-effects of function calls can be ignored at this point. During the computation of the SSA values only those changes to variables are considered that will take place no matter which side-effects occur during the program execution.

3.1.2.1 Domination, dominator trees and dominance frontiers

Dominance frontiers yield an easy and efficient way to compute the locations where ϕ-functions have to be placed. The computation of the dominance frontiers is based on the dominator tree which has to be computed first. An algorithm computing the SSA values by using dominance frontiers can be found in [CFR+89, CFR+91].

Definition 16 (dominators)

A node n' is said to dominate another node n ($n' \in dom(n)$), if n' appears on every path from the start node $START$ to n.

$$\begin{aligned} dom &: V \Leftrightarrow\!\!\!\rightarrow \mathbb{P}(V) \\ dom(n) &= \{n' \in V \mid \forall n_1 \ldots n_k \in path(START, n) : \exists (1 \leq i \leq k) : n_i = n'\} \end{aligned}$$

If n' dominates n ($n' \in dom(n)$) and $n \neq n'$, n' is said to strictly dominate n.

Fact 17

Domination is reflexive as well as transitive.

Definition 18 (post-dominators)

A node n' is said to post-dominate another node n ($n' \in pdom(n)$), if n' appears on every path from n to the end node END.

$$\begin{aligned} pdom &: V \Leftrightarrow\!\!\!\rightarrow \mathbb{P}(V) \\ pdom(n) &= \{n' \in V \mid \forall n_1 \ldots n_k \in path(n, END) : \exists (1 \leq i \leq k) : n_i = n'\} \end{aligned}$$

Note that the post-dominance relation becomes equivalent to the dominance relation, if every edge in the control flow graph is reversed and the role of the $START$ and END node are exchanged.

Every node n has an unique immediate dominator ($idom(n) \in V$) which is defined as follows.

Definition 19 (immediate dominators)

$$\begin{aligned} idom &: V \Leftrightarrow\!\!\!\rightarrow V \\ idom(n) \in (dom(n) &\Leftrightarrow \{n\}) \quad \wedge \quad \forall (n' \in (dom(n) \Leftrightarrow \{n\})) : n' \in dom(idom(n)) \end{aligned}$$

The immediate dominator relation ($idom$) builds a tree which is called the dominator tree. Looking at this tree it can be seen that a node n' is dominated by another node n if and only if n' is contained in the sub-tree rooted with node n. The root of the dominator tree is always the start node $START$, which is quite obvious since every node is dominated by the start node.

Definition 20 (dominance frontiers)

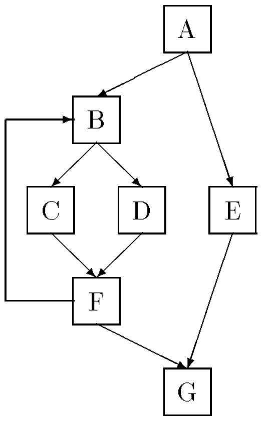
The dominance frontier of a node n ($df(\{n\})$) is the set of all nodes n' having a predecessor n'' that is dominated by n without being strictly dominated by n itself. Accordingly, the dominance frontier of a set of nodes is defined as the union of the dominance frontiers of each node.

$$\begin{aligned} df &: \mathbb{P}(V) \Leftrightarrow\!\!\!\rightarrow \mathbb{P}(V) \\ df(\{n_1, \ldots, n_m\}) &= \{n' \in V \mid \exists (n \in \{n_1, \ldots, n_m\}, n'' \in pred(n')) \ : \\ &\qquad \neg((n \in dom(n')) \wedge (n \neq n')) \quad \wedge \quad (n \in dom(n''))\} \end{aligned}$$

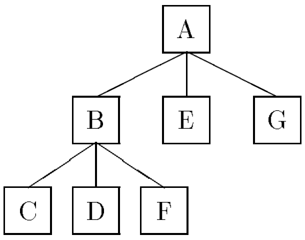
The dominance frontier of a node contains exactly those nodes where ϕ-functions have to be placed, if a variable is changed at this node.

The example in figure 3.12 shows the dominator tree as well as the dominance frontiers for an example graph[10]. If e.g. a variable is changed in the code belonging to node F,

[10]Note that the graph shown in this example is not one of the CFG graphs whose properties have been described in the earlier sections. Here A serves as the start node, whereas G is the end node of the graph.



CFG graph



Dominator tree

n	dom(n)	df({n})
A	{ A }	{ }
B	{ A,B }	{ G }
C	{ A,B,C }	{ F }
D	{ A,B,D }	{ F }
E	{ A,E }	{ G }
F	{ A,B,F }	{ B,G }
G	{ A,G }	{ }

Dominance frontiers

Figure 3.12: CFG graph, dominator tree and dominance frontiers

ϕ-functions have to be placed at the nodes B and G. Note that placing a ϕ-function at a certain node is a new assignment for this variable, and therefore further ϕ-functions caused by this assignment might become necessary then. E.g. if there is a variable changed at node D a new ϕ-function has to be placed at node F. This newly generated ϕ-function makes it necessary to place further ϕ-functions at the nodes B and G.

All nodes where ϕ-functions have to be placed ($phi(\{n_1, \ldots, n_m\})$) can be obtained by applying the df-function to the set of nodes where a certain variable is changed ($\{n_1, \ldots, n_m\}$) until stabilisation is reached.

Definition 21 (phi functions)

$$\begin{aligned} phi &: \mathbb{P}(V) \Leftrightarrow\!\!\!\rightarrow \mathbb{P}(V) \\ phi(\{n_1, \ldots, n_m\}) &= \bigcup_{i=1}^{\infty} df^i(\{n_1, \ldots, n_m\}) \end{aligned}$$

In our example this would mean that if a variable is changed at the nodes C and D, then ϕ-functions will have to be placed at the nodes $phi(\{C, D\}) = \{B, F, G\}$. The repeated computation of the dominance frontiers stabilises after the third iteration since no new nodes are added to the set of nodes that need a ϕ-function.

$$\begin{array}{lll} df(\{C, D\}) & & = \{F\} \\ df^2(\{C, D\}) & = df(\{F\}) & = \{B, G\} \\ df^3(\{C, D\}) & = df(\{B, G\}) & = \{G\} \end{array}$$

Dominance frontiers are computed only once for each node of a CFG graph, whereas the computation of the ϕ-function sets has to be repeated for every variable modified in the corresponding function.

3.1.2.2 Algorithm

The algorithm used to compute the SSA form that was chosen here, is based on the algorithm presented in [CFR$^+$91]. The algorithm was mainly chosen for reasons of easier implementation, and it could easily be replaced by a faster one, if necessary.

```
{
  int a_0, a_1, a_2;
  do (...) {
    a_2 = φ(a_0, a_1);
/* the next statement is useless!! */
    c_2 = φ(c_0, c_1);
    int c_1;
    c_1 = ...;
    ...
    a_1 = c_1;
  }
}
```

Figure 3.13: Program with ϕ-function for non-accessible variable

However, most of the algorithms usually used to compute SSA form do have one significant disadvantage. A problem that cannot be handled by these algorithms arises due to the fact that the C programming language allows functions containing blocks with variable declarations. This leads to variables with a limited scope and hence these variables are not valid throughout the whole function. Since blocks do not influence the CFG graph[11], it is not possible to consider the effects of blocks when calculating the dominance frontiers. This is because the computation of the dominance frontiers is only based on the CFG graph and does not depend on the program code itself. Altogether this may lead to the creation of ϕ-functions at places where the corresponding variable is not even accessible, when using the usual SSA algorithms.

The program shown in figure 3.13 demonstrates what would happen if the effects of blocks containing variable declarations were not taken into consideration. The ϕ-function generated for a is correct, whereas the one generated for c is superfluous as well as wrong, since c is not accessible at that place. This effect may lead to huge amounts of useless ϕ-functions being generated.

Since it is an essential property of algorithms based on dominance frontier calculation that the dominance frontiers are computed only once for all variables, it is impossible to take blocks into account when calculating the dominance frontiers. If one wants to consider the effects of blocks correctly, it would be necessary to calculate dominance frontiers for every block containing variable declarations separately, which would significantly slow down the computation in almost every case. Besides, it would be necessary to make sure that code belonging to different blocks always belongs to different CFG nodes, which would increase the number of nodes as well as the time needed to calculate the dominance frontiers.

To prevent the superfluous ϕ-functions, an algorithm which differs from the one in [CFR$^+$91] has to be used. Since it is not possible to prevent the creation of ϕ-functions or at least the computation of the locations where they have to be placed, needless ϕ-functions will be removed by the algorithm instead. As already mentioned before, the algorithm consists of three steps. They will now be explained in detail in the following.

First step: During the first step of the algorithm, the CFG graph is being built, and the generation of the SSA numbers for all assignment targets takes place. Additionally,

[11]Blocks do not force the creation of another node when they are reached during the step creating the CFG graph. It is possible to have *SEQ* nodes containing code from different blocks, if there is no non-trivial control flow contained in that code.

the assigned variables are collected and annotated to the CFG node which belongs to the code. In case a variable is changed twice or even more often in the code belonging to a single CFG node, only the last definition is annotated to the CFG node, since this is the one that is valid after the execution of the code belonging to the CFG node. All this can be done by traversing the code once.

Second step: The second step computes the dominator tree, the dominance frontiers and the locations where ϕ-functions are necessary. In [LT79] and [Har85] it is described how the computation of the dominator tree and the dominance frontiers can be done efficiently. Since the second algorithm is more complicated, and the time spent for the computation of the ϕ-functions is only of minor interest for the alias analysis, the first algorithm was chosen. The computation of the locations where ϕ-functions have to be placed generates only incomplete ϕ-functions, because the reaching definitions are not yet computed. This means that only the left hand side of the ϕ-function assignment is known, and that the arguments of the ϕ-function have to be filled in later. Nevertheless they are already placed in the code, since the left hand side of the ϕ-function assignments is necessary for the next step.

Third step: During the third and last step the reaching definitions are computed. The previous steps ensure that there is always only one reaching definition for every used variable, and that all variable definitions (including the ϕ-function assignments) already received a unique SSA number. Now, every used variable receives the SSA number of the (unique) reaching definition. Additionally, the superfluous ϕ-functions are removed and the remaining ϕ-functions are completed by adding all corresponding reaching definitions to the right hand side of the ϕ-functions. This step requires the code to be traversed once more.

The algorithm shown in figure 3.14 computes all the three steps that have been described above. There are counters that store the actual SSA number for every variable[12]. They can be accessed by using the *set_act_ssa_val* and *get_act_ssa_val* functions.

Computation of SSA numbers for defined variables (first and second step)

At the beginning all counters are initialised, which means they are set to zero (3-4). Now the statements contained in the function *func* will be processed in textual order (7,8,16). If the processed statement is an assignment to a variable (10,11) a new SSA number will be given to the assigned variable. This is done by using the corresponding counter, which has to be incremented then (12-14). In addition, the computation of the CFG graph is effected (9). Now, after the first step is taken and all statements have been processed, the ϕ-functions are computed during the second step using the algorithms described in [CFR+91] and [LT79] (19). Now the SSA numbers for all variable definitions (usual assignments as well as ϕ-function assignments) have been computed, and the SSA numbers for all used variables have to be computed subsequently as the next step. This is done during the third step that will be explained in detail soon.

Note that not every assignment is an assignment to a variable, e.g. an assignment like `*a = b` does not necessarily change the value of a variable. This assignment might change

[12]Note that the temporary variables that have been produced by the function call normalisation are excluded here. They are assigned a new value once and this value will be used only once (right after it was assigned). Hence it does not make sense to produce ϕ-functions for these variables because they are not used anywhere else.

```
function compute_ssa_numbers(func)
{
   for all ($var \in used\_vars(func)$) {
      set_act_ssa_val(var, 0);
   }
   /* first step */
   stmt = first_stmt(func);
   while ($stmt \neq NOSTMT$) {
      generate_cfg_graph(stmt);
      var = assigned_variable(stmt);
      if ($var \neq NOVAR$) {
         ssa_no = get_act_ssa_val(var) + 1;
         var.ssa_val = ssa_no;
         set_act_ssa_val(var, ssa_no);
      }
      stmt = next_stmt(func);
   }
   /* second step */
   compute_phi_funcs();
   /* third step */
   for all ($var \in used\_vars(func)$) {
      if (is_block_var(var)) set_act_ssa_val(var, UNDEF);
      else set_act_ssa_val(var, 0);
   }
   compute_reaching_defs(main_stmt_seq(func));
}
```

Figure 3.14: Function computing the SSA numbers for assigned variables

a variables value due to side-effects, but such effects are not taken into consideration here. Only those assignments that immediately assign a new value to a variable will cause the generation of a new SSA number. Since the C programming language does not really distinguish between arrays and pointers ([Ame89a]), the computation of the SSA values treats arrays and pointers equally. This means that the SSA number of an array variable will never change, because it is not permitted to assign a new value to an array. If an array is used it is treated like a dereferenced pointer ($\texttt{arr}[\texttt{n}] \cong *(\texttt{arr} + \texttt{n})$), and so there is no new SSA number necessary. Because the semantic of the array dereference operator is defined to be the same as the expression shown above using the pointer dereference and the addition operator, it is only a syntactical alternative, and it would therefore be senseless to distinguish between them. The fact that C does not really distinguish between pointers and arrays is reflected in the quite unusual convention that the C programming language allows `arr[n]` as well as `n[arr]` to access an array element, which is caused by the commutative addition operator ([Ame89b]). All variables that do not receive a SSA number during the first step are treated as uses rather than as definitions. These variables will receive their SSA numbers during the last step, even if they appear on the left hand side of an assignment.

Computation of SSA numbers for used variables (third step)

Before the computation of the SSA numbers for the used variables can start, all the variable counters have to be re-initialised. The counters are either set to *UNDEF*, if they are defined in local blocks and therefore not accessible yet, or to zero otherwise (21-23). The major part of the computation of the missing SSA numbers is done by the function *compute_reaching_defs* that will be explained next. Since this function takes a sequence of statements as its input, the statement sequence contained in the body of the analysed function has to be passed instead of the function itself (25).

The *compute_reaching_defs* function itself is shown in figure 3.15. The main problem that has to be handled occurs if a statement is followed by another statement in textual order, but those statements are never executed in this order. E.g. the first statement contained in the else-part of an **if** is never executed after the last statement of the then-part, even though they appear successively in the code. This implies that the reaching definitions for the else-part cannot be derived from the reaching definitions that are valid after the processing of the then-part. Since the reaching definitions for the first statements of the then- and the else-part are exactly the same, it is necessary to save the sets of reaching definitions before the then-part is processed, and to restore them before the else-part is processed. The second problem that has to be dealt with is the removal of the useless ϕ-functions which may occur outside the scope of a variable.

At the beginning the fields that are used to store the actual state of the SSA counters at a certain statement have to be initialised (3,4). As can be seen, the function *compute_reaching_defs* takes a statement sequence as its input and goes through these statements one by one (6,7,42). These statement sequences contain only those statements that appear at the same nesting level. The statements contained in blocks, loop bodies etc. are processed by explicitly calling the function again for those sub-sequences.

It is essential for this algorithm that the statements are processed in a certain order, which assures that *LABEL* statements are never processed before at least one of their predecessors (execution order) has been processed. This predecessor may be either a corresponding *JUMP* statement or the predecessor in textual order if this statement is no unconditional *JUMP*. If there were *LABEL* statements that could be reached without processing one of their predecessors first, there would be no information about the reaching definitions available, and hence it would be impossible to compute the correct SSA numbers then. Therefore it is not possible to simply go through the statements in textual order, if there are arbitrary *LABEL* and *JUMP* statements contained in the code. This kind of proceeding implies that all corresponding *LABEL* and *JUMP* statements have to be at the same nesting level, because it would be possible to jump into blocks without taking care of the possibly changed set of accessible variables otherwise. However, it is not necessary to explicitly handle this kind of *JUMP* and *LABEL* statements, because the SUIF compiler already does all the work by transforming the code in such cases.

At the beginning of the algorithm all used variables appearing in the statement that is currently processed receive the SSA number that is valid at this point of the program (8-9). Remember that the initialisation of the counters keeping track of the SSA numbers for the reaching definitions has already been done by the *compute_ssa_numbers* function before the *compute_reaching_defs* function was called. At this stage the above-mentioned problems arising from the control flow and blocks have to be handled. In case the currently processed statement is an assignment that directly changes a variable (13,14), the counter that keeps the value of the actual SSA number has to be set to the new value (16). If the assignment is a ϕ-function assignment and the value of the actual SSA number is *UNDEF* (15), the assignment is superfluous, since the actual SSA number is only set to *UNDEF* if the currently processed statement is outside the variables scope. In this case the statement will be removed (17).

Every time a new block is reached the counters of the variables declared in this block

```
function $compute\_reaching\_defs(stmt\_seq)$
{
  for all $(stmt \in stmt\_seq)$ {
    $stmt.act\_vars = NOVARS$;
  }
  $stmt = first\_top\_level\_statement(stmt\_seq)$;
  while $(stmt \neq NOSTMT)$ {
    for all $(var \in used\_vars(stmt))$ {
      $var.ssa\_val = get\_act\_ssa\_val(var)$;
    }
    switch $(type(stmt))$ {
      case $ASSIGNMENT$ :
        $var = assigned\_variable(stmt)$;
        if $(var \neq NOVAR)$ {
          if ((not $is\_phi\_assign(stmt)$) or $(get\_act\_ssa\_val(var) \neq UNDEF)$) {
            $set\_act\_ssa\_val(var, var.ssa\_val)$;
          } else $remove(stmt)$;
        }
      case $BLOCK$ :
        for all $(var \in block\_vars(stmt))$ $set\_act\_ssa\_val(var, 0)$;
        $compute\_reaching\_defs(block\_stmt\_seq(stmt))$;
        for all $(var \in block\_vars(stmt))$ $set\_act\_ssa\_val(var, UNDEF)$;
      case $IF\text{-}THEN\text{-}ELSE$ :
        $mod\_vars = save\_mod\_vars()$;
        $compute\_reaching\_defs(then\_stmt\_seq(stmt))$;
        $restore\_mod\_vars(mod\_vars)$;
        $compute\_reaching\_defs(else\_stmt\_seq(stmt))$;
      case $FOR\text{-}LOOP$ :
        $mod\_vars = save\_mod\_vars()$;
        $compute\_reaching\_defs(for\_stmt\_seq(stmt))$;
        $restore\_mod\_vars(mod\_vars)$;
      case $DO\text{-}LOOP$ :
        $compute\_reaching\_defs(loop\_stmt\_seq(stmt))$;
      case $LABEL$ :
        if $(stmt.act\_vars \neq NOVARS)$ $restore\_all\_vars(stmt.act\_vars)$;
      case $JUMP$ :
        $dest\_stmt = jump\_dest(stmt)$;
        if ($(dest\_stmt.act\_vars = NOVARS)$ and $(no\_text\_order\_pred(dest\_stmt))$) {
          $dest\_stmt.act\_vars = save\_all\_vars()$;
        }
    }
    $stmt = next\_top\_level\_statement(stmt\_seq)$;
  }
}
```

Figure 3.15: Function computing the SSA numbers for used variables

have to be set to zero (20) before the statements contained in the block are processed (21). Afterwards it is necessary to set the counters back to *UNDEF* since the statements that are processed afterwards are outside the scope of the variables declared in the block (22).

To process an *IF-THEN-ELSE* statement it is necessary to store the actual values of the SSA number counters (24) before the statements contained in the then-part are processed (25). It is not necessary to save the actual SSA numbers of all variables, since there will only be a few changes in many cases. Here it is sufficient to save the SSA numbers of those variables for which a ϕ-function was generated after the *IF-THEN-ELSE* statement (*mod_vars*), since these are exactly those variables that are changed by the statements contained in either the then- or else-part of the *IF-THEN-ELSE* statement. Before the statements contained in the else-part of the *IF-THEN-ELSE* statement are processed (27), the previously saved SSA numbers have to be restored (26). This ensures that changes made to variables in the then-part cannot affect the else-part, and that the counters are exactly in the same state they were in when the processing of the then-part began. The variables contained in the condition of the *IF-THEN-ELSE* statement already received the correct SSA numbers (8-9) before the statements contained in the then- and else-part have been processed, so they do not have to be handled explicitly here.

The processing of *FOR-LOOP* statements requires similar actions. Before the statements contained in the loop's body are handled (30), the actual SSA numbers are saved (29). Again only those values are stored that are changed in the body and for which ϕ-functions have been generated. This is necessary because the program continues at the top of the loop after the loop's body was executed. Here the loop's index is incremented and the loop's condition is checked. In case there are no further iterations necessary, the program continues after the end of the loop. This means that the SSA counters have to be restored (31) after the processing of the statements contained in the body is done, which guarantees that the values are again like they were at the beginning of the body. Like the condition of the *IF-THEN-ELSE* statement the condition of the *FOR-LOOP* does not have to be handled explicitly here, since the SSA numbers are already set to the correct values before the body of the loop is processed.

DO-LOOP statements are much easier to handle. Since the program continues in textual order after the last iteration, it is not necessary to save and/or restore the actual SSA numbers here. All that has to be done is to process the statements contained in the loops body (33). *DO-LOOP*s always evaluate their condition after the body has been executed. This has to be taken into account during the computation of the reaching definitions and hence the condition is processed after processing the statements contained in the body.

The last problem that has to be dealt with arises if a *LABEL* statement is preceded by an unconditional *JUMP* statement[13], since the program will never continue in textual order in this case. When a *JUMP* statement is processed the first thing that has to be done is to locate its destination (the corresponding *LABEL* statement) (37). If this *LABEL* is preceded by an unconditional *JUMP* statement it is necessary to save the actual values of all SSA counters, as far as this has not yet been done before (38,39). These values are later restored when the *LABEL* statement is processed (35). If there are multiple *JUMP* statements with the same destination only the statement that is processed first stores the actual SSA counters. One might wonder why the possibly different SSA counters of the other *JUMP* statements are ignored here, but this can safely be done because there are ϕ-functions following the *LABEL* statement for exactly those variables where different SSA

[13]Strictly speaking: the last non-*LABEL* statement preceding a *LABEL* statement is an unconditional *JUMP* statement. This takes multiple *LABEL* statements marking the same program location into account as well.

counters appear at the corresponding *JUMP* statements. Those SSA counters that are not changed by the following ϕ-functions are the same for every *JUMP* statement, and hence any *JUMP* statement can be chosen to store the SSA counters. As can be seen here, it is very important that either the statement preceding the *LABEL* statement is not an unconditional *JUMP*, or at least one of the corresponding *JUMP* statements has been processed before the *LABEL* statement itself is processed. In the first case the actual SSA numbers are not changed, and in the second case there are saved SSA counters available that can be restored then.

When the algorithm wants to find out if it is necessary to save the actual SSA counters when a *JUMP* statement is processed, the corresponding *LABEL* statement is analysed as described above. The decision depends on the statement preceding the *LABEL* statement. If this statement is not an unconditional *JUMP* statement, it is assumed that it is not necessary to save and restore the actual values of the SSA counters, because the *LABEL* statement is processed after processing its predecessors and therefore correct SSA counter values will be available then. However, this is not correct if the label is preceded by structural dead code (see chapter 2). This code will never be processed and therefore it would be necessary to store the actual SSA counters in such cases as well. Fortunately, structural dead code has already been removed before the calculation of the SSA numbers takes place. Hence the above described problem cannot arise anymore, and so it is sufficient to check for unconditional *JUMP* statements here.

Processing order of statements

The two functions that are shown in figure 3.16 are used to determine the correct order in which the statements of a statement sequence have to be processed.

The function *first_top_level_statement* initialises the iteration process and returns the first statement that is processed afterwards. At the beginning the worklist that contains branches which have not yet been processed is emptied (3). Additionally, the flags indicating whether a statement has already been processed are all set to false (5,6). The first statement that is processed is the first statement in textual order (4,8), and therefore this statement has to be marked as already processed now (9).

From here on the statement that has to be processed next is determined by subsequent calls to the function *next_top_level_statement*. Unless the actual statement is not the last one in the sequence, the next statement that is processed is the next statement in textual order (3,4). Only if the actual statement is an unconditional *JUMP* statement will the next statement that is to be processed have to be determined differently, because the next statement in textual order might be a *LABEL* statement whose corresponding *JUMP* statements have not yet been processed. Instead of the next statement in textual order the destination of the *JUMP* statement is processed as next then (5,6). In case the actual statement is a conditional *JUMP* statement, the destination statement is added to the worklist, to make sure that this statement and its successors are processed as well (7,8). To prevent statements from being processed twice it is checked whether the next statement has already been processed before (10). In the latter case new statements are taken from the worklist until either an unprocessed statement is found or the worklist is empty (10,11). If all statements have already been processed there is nothing left to do (17). Otherwise the found (unprocessed) statement is the next statement to be processed. Hence its processed-flag will be set, and the statement will be returned afterwards (14-16).

```
function $first\_top\_level\_statement(stmt\_seq)$
{
  $worklist = EMPTYLIST$;
  $act\_no = 0$;
  for all $(stmt \in stmt\_seq)$ {
    $stmt.processed =$ false;
  }
  $first\_stmt = stmt\_seq[act\_no]$;
  $first\_stmt.processed =$ true;
  return $first\_stmt$;
}

function $next\_top\_level\_statement(stmt\_seq)$
{
  if $(act\_no \neq last\_stmt\_no(stmt\_seq))$ $next\_no = act\_no + 1$;
  else $next\_no = act\_no$;
  if $(is\_uncond\_jump(stmt\_seq[act\_no]))$ {
    $next\_no = dest\_stmt\_no(stmt\_seq[act\_no])$;
  } else if $(is\_cond\_jump(stmt\_seq[act\_no]))$ {
    $enqueue(worklist, dest\_stmt\_no(stmt\_seq[act\_no]))$;
  }
  while $(stmt\_seq[next\_no].processed$ and (not $is\_empty(worklist)))$ {
    $next\_no = dequeue(worklist)$;
  }
  if (not $stmt\_seq[next\_no].processed)$ {
    $act\_no = next\_no$;
    $stmt\_seq[act\_no].processed =$ true;
    $next\_stmt = stmt\_seq[act\_no]$;
  } else $next\_stmt = NOSTMT$;
  return $next\_stmt$;
}
```

Figure 3.16: Functions determining the processing order of statements

3.1.2.3 Time and space bounds

The time as well as the space needed to compute the SSA numbers of a function is influenced by several properties of the function. The following list gives an overview of these properties:

- number of generated ϕ-functions ($|PHI|$)
- number of occurring variables[14] ($|VAR_{occ}|$)
- number of nodes of the CFG graph ($|V|$)
- number of edges of the CFG graph ($|E|$)

Of course these properties are not completely independent of each other, and so some of them can be used to build an upper bound for one of the others. However, using all of them leads to more exact measures for the time and space bounds.

Time bounds

The first step of the entire algorithm traverses the code once to set the SSA numbers of all assigned variables, which takes time proportional to the size of the input code ($O(|STMT|)$). The counters that keep track of the actual SSA numbers are modified several times during the first step of the algorithm, but since they can be accessed within constant time they hardly influence the time bounds. The counters can be accessed that fast because every variable occurring in the code has a link to the corresponding entry in the symbol table, which as well contains the counters.

The time needed for the second step is mainly influenced by the computation of the dominance frontiers. Their computation can take time in $O(|E| + |V|^2)$ in the worst case ([CCF91]). However, such behaviour is not expected in general. Even though certain programs which reach this worst case bound exist, it is very unlikely that a real program will ever come close to this bound. After the dominance frontiers are calculated the locations where ϕ-functions have to be placed will be computed. This takes time proportional to the number of ϕ-functions that are generated ($O(|PHI|)$).

The last step of the algorithm has to traverse the complete code again, which takes time proportional to the size of the output code. Since the ϕ-functions have already been placed in the code this takes time in $O(|STMT| + |PHI|)$. The time needed for all calls to the functions *first_top_level_statement* and *next_top_level_statement* is in $O(|STMT| + |PHI|)$ as well, since every statement can add at most one element to the worklist. Hence there are never more than $|STMT|$ elements enqueued to or dequeued from the worklist during these calls. Additional time is necessary to save and restore the actual SSA numbers for some of the statements. For all programs which do not contain *LABEL* statements that are preceded by unconditional *JUMP* statements, this will take time proportional to the number of generated ϕ-functions ($O(|PHI|)$). Remember that only those variables are saved that are changed in the body of the statement, and that there will be ϕ-functions that have then been generated for all these variables. In case there are *LABEL* statements preceded by unconditional *JUMP* statements, the time needed to save the SSA numbers for such statements is proportional to the number of variables that are used in the code ($O(|VAR_{occ}|)$), since the SSA values of all these variables have to be saved then. Because there can be $O(|STMT|)$ statements like these, this leads to an overall worst case time

[14]Here 'occurring variables' means all variables occurring in one of the functions statements (defined and used ones). Variables that are declared but not used in the particular function are not included.

bound of $O(|STMT| \cdot |VAR_{occ}|)$. Of course it is more than unlikely for real programs to reach this time bound.

Summing up, this leads to the following overall worst case time bound:

$$\begin{array}{rll} & O(|STMT|)\ + & \text{(first step)} \\ & O(|E| + |V|^2)\ +\ O(|PHI|)\ + & \text{(second step)} \\ & O(|STMT| \cdot |VAR_{occ}|) & \text{(third step)} \\ = & O(|STMT|^2) & \end{array}$$

This calculation is based on the assumption that there is only a limited number of variables occurring in a single statement[15], which is true for almost every program. To prove that the above calculation is correct, it is sufficient to prove that the following equations are correct.

$$\begin{array}{ll} O(|STMT|) \subseteq O(|STMT|^2) & (1) \\ O(|E| + |V|^2) \subseteq O(|STMT|^2) & (2) \\ O(|PHI|) \subseteq O(|STMT|^2) & (3) \\ O(|STMT| \cdot |VAR_{occ}|) \subseteq O(|STMT|^2) & (4) \end{array}$$

The first equation is obviously true. The number of nodes in the CFG graph ($|V|$) is in $O(|STMT|)$, and since there are no nodes having more than two outgoing edges the number of edges is in $O(|STMT|)$ as well, and hence the second equation holds. The above made assumption (that there is a constant upper bound for the number of variables which occur within a single statement), leads to the fact that there are at most $O(|STMT|)$ used variables in a function. Since there are at most $O(STMT)$ locations where ϕ-functions may be placed, there can be at most $O(|STMT|^2)$ ϕ-functions per function (third equation). As shown before $O(|VAR_{occ}|) \subseteq O(|STMT|)$ holds, and so the fourth equation does as well.

Although the time needed to compute the SSA numbers has a quadratic worst case time bound, this behaviour does not occur when dealing with usual programs. For programs which do not contain unstructured control flow it is impossible to reach this time bound[16], and even for programs containing *LABEL* and *JUMP* statements this time bound is only reached by programs that are explicitly made to reach this bound. However, such programs are only of theoretical interest and do not have any practical relevance.

It is impossible to build an algorithm with better worst case time behaviour since the number of generated ϕ-functions may grow quadratically with the input size. This happens for programs like the one shown in figure 3.17. The corresponding CFG graph is shown in figure 3.18. Here the nodes labelled from I_1 to I_n represent the first part of the program containing the **if** and **goto** statements, whereas the nodes L_1 to L_n represent the second part containing the labels and the assignment statements. Every time n will be incremented (by adding only a constant amount of statements to the program), $n \Leftrightarrow 1$ new ϕ-functions will be generated, which leads to $|PHI| = \frac{n \cdot (n-1)}{2}$.

The algorithms presented in [CCF91] and [CFR$^+$91] both have a $O(|E|+|V|^2)$ worst case time bound for the computation of the locations where ϕ-functions have to be placed. As well both of them are based on the dominance frontier concept. In [CF95] Cytron and Ferrante

[15]The variables occurring inside the bodies of *DO-LOOP*s, *FOR-LOOP*s and *IF-THEN-ELSE* statements are excluded, since it is quite obvious that there is no limit for the number of variables that may occur in the body of a big loop.

[16]For these programs the time needed to compute the third step is in $O(|STMT| + |PHI|)$, which is optimal since this is the size of the output code.

```
  if (...) goto lab1;
  if (...) goto lab2;
  if (...) goto lab3;
  ...
  if (...) goto labn;
lab1 :
  $var^1 = \ldots;$
lab2 :
  $var^2 = \ldots;$
lab3 :
  $var^3 = \ldots;$
  ...
labn :
  $var^n = \ldots;$
```

```
  if (...) goto lab1;
  if (...) goto lab2;
  if (...) goto lab3;
  ...
  if (...) goto labn;
lab1 :
  $var^1_1 = \ldots;$
lab2 :
  $var^1_2 = \phi(var^1_0, var^1_1);$
  $var^2_1 = \ldots;$
lab3 :
  $var^1_3 = \phi(var^1_0, var^1_2);$
  $var^2_2 = \phi(var^2_0, var^2_1);$
  $var^3_1 = \ldots;$
  ...
labn :
  $var^1_n = \phi(var^1_0, var^1_{n-1});$
  $var^2_{n-1} = \phi(var^2_0, var^2_{n-2});$
  ...
  $var^{n-1}_2 = \phi(var^{n-1}_0, var^{n-1}_1);$
  $var^n_1 = \ldots;$
```

Figure 3.17: Program with quadratically growing number of ϕ-functions

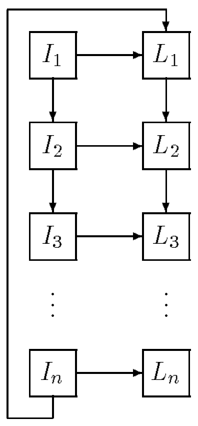


Figure 3.18: Control flow graph for program with quadratically growing number of ϕ-functions

present an alternative concept which avoids the computation of the dominance frontiers. Like in [CCF91], SEGs (Sparse data flow Evaluation Graphs) are computed, which are used to generate the SSA numbers later. The worst case time bound for the construction of the SEG which has previously ([CCF91]) been $O(|E| + |V|^2)$ was improved, and is in $O(|E| \cdot \alpha(|E|))$[17] now. Nevertheless the calculation of the ϕ-functions still takes $O(|E|+|V|^2)$ time in the worst case. Even though the asymptotical bounds of the second algorithm are much better, there are only minor differences in practice because usual programs have a certain structure that almost always prevents the algorithm from reaching its worst case bounds.

It is reported that all of the above-mentioned algorithms are almost linear, as long as they are applied to usual programs ([CCF91, CFR$^+$91, CF95]).

Space bounds

The space needed for the computation of the SSA numbers mainly depends on the number of ϕ-functions that are generated ($O(|PHI|)$). There is some additional space necessary to store the counters that keep track of the actual SSA numbers, but this space is proportional to the number of used variables ($O(|VAR_{occ}|)$), which is in $O(|STMT|)$ as shown before. Altogether this results in a $O(|STMT| + |PHI|)$ worst case space bound. This is already the best reachable worst case space bound as well as the best case space bound since the output program size is $|STMT| + |PHI|$. As shown in the previous section the number of ϕ-functions can still grow quadratically with the number of statements in the input program ($|STMT|$), which leads to a quadratic worst case space bound of $O(|STMT|^2)$.

The average time and space bounds of the SSA computation for a couple of example programs are shown in section 4.3. They underline the assumptions made above on the average case behaviour of the algorithm for real C programs.

Now that the SSA form computation has been defined, the following parts of the alias analysis can benefit from the gained information. The SSA computation is not essential for the following parts of the analysis. These parts could as well be executed without the preceeding transformation of the analysed program into SSA form. However, the SSA form computation will be able to increase the precision of the alias analysis significantly in many cases.

[17] Here α is the very slowly growing functional inverse of the Ackermann function.

3.2 Intraprocedural analysis

The main aim of the intraprocedural analysis is to analyse all the effects a single function can have on a program's variables. For this purpose a so-called function interface graph will be constructed that summarises all these effects. This function interface graph collects all the information necessary to detect every alias that may be introduced while the function is being processed. Even those aliases that are introduced somewhere inside the function and that are in any case destroyed later, are represented in the function interface graph.

Function interface graphs are built within three passes: the first pass processes the ϕ-functions that have been added by the SSA computation, whereas the second pass analyses all the occurring C statements of the code belonging to a given CFG node. Finally, the third pass merges the function interface graphs belonging to the different CFG nodes resulting in the function interface graph for the whole function. During the second pass nodes and edges are added to the function interface graph to represent the C statements. Note that variables with different SSA values are treated as independent variables during this pass. The connections between the variables differing only by their SSA value is taken care of when the ϕ-functions are processed within the first pass. During this pass the nodes corresponding to the SSA variables occurring in a ϕ-function will be joined.

3.2.1 Alias information

The aliases that are found are summarised by the above-mentioned function interface graph. Before going into the details of the function interface graph computation, it is necessary to describe which kind of alias information will be computed (section 3.2.1.2). This will become much easier and clearer if some terms that have already been used before in an informal manner will be formally defined first.

3.2.1.1 Basic definitions

Since aliases are a relation between certain expressions in a certain program state it is not possible to talk about aliases without talking about expressions and program states as well. Therefore it is necessary to formally define some terms related to expressions and program states before a definition of aliases can be given.

Definition 22 (locations, values and memory state)
A memory state $m \in MEM$ of a program is a function which returns the values contained at the memory locations used by the program

$$MEM \subseteq (LOC \rightarrow VAL)$$

where $LOC \subset \mathbb{N}$ is the set of memory locations (addresses) accessed by the program and VAL is the set of possible values (bytes) stored at these locations. The set of memory states (MEM) contains only those memory states that may really occur during the program execution.

Of course the memory state can change during the program execution (e.g. if a new value is assigned to a variable). In this case $|MEM| > 1$ holds.

Definition 23 (program states)
A program state $st \in STATE$ of a program is a memory state together with the

position of the program counter p ($p \in POS$). The program counter indicates which statement is to be executed as next.

$$STATE \subseteq MEM \times POS$$

The set of program states ($STATE$) contains only those program states that may really occur during the program execution. The two functions *mem* and *pos* are used to return the memory state respectively the position of the program counter of a given program state.

$$mem \;:\; STATE \Leftrightarrow\!\!\!\rightarrow MEM \quad , \quad pos \;:\; STATE \Leftrightarrow\!\!\!\rightarrow POS$$

$$\begin{array}{lcl} mem((m,p)) & = & m \\ pos((m,p)) & = & p \end{array}$$

Definition 24 (simple expressions)
A simple expression $exp \in EXP_{simple}$ is a C-like expression containing variables, constants and arbitrary many operators, where all but the following operators may be used: '( )', '++', '⇔⇔', '? :', '=', '+=', '⇔=', '*=', '/=', '%=', '&=', '|=', '^=', '<<=', '>>='. The set EXP_{simple} is the set containing all simple expressions.

Simple expressions are exactly those C-like expressions that neither change the value of a storage location (assignments) nor contain an implicit conditional statement ('? :') or function call ('( )'). Hence simple expressions neither modify the memory state nor influence the control flow of the program.

Definition 25 (base expression)
A base expression $exp \in EXP_{base}$ is a simple expression containing exactly one occurrence of a variable[18]. Here EXP_{base} denotes the set containing all base expressions.

If a variable is accessed in a program the storage locations and values that can be accessed by base expressions are the ones that have to be taken care of, because they can appear in new alias relations.

Definition 26 (dereference operation)
A dereference operation is the reference to a value stored at a certain location by a pointer plus an offset. It is specified by a tuple (m, n) where m is the offset value and n is the size of the referenced value in bytes[19]. In some cases the offset value (m) may be unknown at compile time, which will be indicated by a question-mark (*'?'*). This leads to the following definitions for the set of possible pointer offsets (*OFF*) and the set of dereference operations (*DEROP*).

[18]Note that the expression `a−>b` is a base expression although two identifiers occur in this expression. This is because the second identifier is not a variable, but something much more similar to a constant. This identifier specifies the (constant) offset needed to access one of the elements of a structure.

[19]This causes bit-fields and their components to be treated as the same element if they are stored in the same byte. This could be avoided if the sizes were measured in bits instead. Nevertheless this is not done here for two reasons: first to prevent the examples from getting unnecessarily complex, and second since bit-fields are only rarely used anyway.

$$\begin{aligned} \mathit{OFF} &= \mathbb{Z} \cup \{\text{'?'}\} \\ \mathit{DEROP} &= \{(m,n) \mid m \in \mathit{OFF}, n \in \mathbb{N}\} \end{aligned}$$

Depending on whether the offset is known or unknown, dereference operations can be further classified. Those using unknown offsets are called floating dereference operations ($\mathit{DEROP}_{\mathit{fl}}$), whereas the others are called fixed dereference operations ($\mathit{DEROP}_{\mathit{fix}}$).

$$\begin{aligned} \mathit{DEROP}_{\mathit{fl}} &= \{(\text{'?'}, n) \mid n \in \mathbb{N}\} \\ \mathit{DEROP}_{\mathit{fix}} &= \{(m,n) \mid m \in \mathbb{Z}, n \in \mathbb{N}\} \end{aligned}$$

The two functions *offset* and *width* can be used to access the two components of a dereference operation. The *offset* function returns the first component (offset), whereas the *width* function returns the second component (size) of a dereference operation.

$$\mathit{offset} \; : \; \mathit{DEROP} \Leftrightarrow\!\!\rightarrow \mathit{OFF} \;\; , \;\; \mathit{width} \; : \; \mathit{DEROP} \Leftrightarrow\!\!\rightarrow \mathbb{N}$$

$$\begin{aligned} \mathit{offset}((m,n)) &= m \\ \mathit{width}((m,n)) &= n \end{aligned}$$

Dereference operations are mainly influenced by the following operators[20] of the C programming language: '$*_u$', '[ ]', '⇔>', '$+_b$', '$\Leftrightarrow_b$', '.'. The offset of a dereference operation can be changed by adding something to or subtracting something from a pointer. This can be done directly using the '$+_b$' or '$\Leftrightarrow_b$' operators or indirectly by the '.', '⇔>' and '[ ]' operators.

In many cases it is necessary to compare two dereference operations. The most interesting thing to know about two dereference operations is if the values that can be accessed using the first operation can be accessed by the second one and vice versa. This can be easily checked if one knows the interval of offsets that can be accessed for either operation. If these intervals do not intersect, the dereference operations cannot access the same data. The *acc_rng* function that is now defined returns exactly this interval for a given dereference operation.

The access range belonging to a dereference operation is the interval which contains all offset values belonging to the values that are accessed by a particular dereference operation. The *acc_rng* function returns the access range belonging to a dereference operation, whereas the *deref_op* function returns the dereference operation belonging to a given access range.

Definition 27 (access range)

$$\mathit{acc\_rng} \; : \; \mathit{DEROP} \Leftrightarrow\!\!\rightarrow \mathbb{P}(\mathbb{Z}) \;\; , \;\; \mathit{deref\_op} \; : \; \mathbb{P}(\mathbb{Z}) \Leftrightarrow\!\!\rightarrow \mathit{DEROP}$$

$$\mathit{acc\_rng}(d) = \begin{cases} \{i \mid 0 \le i \Leftrightarrow \mathit{offset}(d) < \mathit{width}(d)\} & \text{if } \mathit{offset}(d) \in \mathbb{Z} \\ \mathbb{Z} & \text{otherwise} \end{cases}$$

$$\mathit{deref\_op}(r) = \begin{cases} (\text{'?'}, 1) & \text{if } r = \mathbb{Z} \\ (\mathit{min}(r), 1 + \mathit{max}(r) \Leftrightarrow \mathit{min}(r)) & \text{otherwise} \end{cases}$$

[20]Since some of the operators in the C programming language are used as unary as well as binary operators, a subscript ('u' or 'b') is added in cases where it might not be clear which one is meant.

Definition 28 (access path)

An access path $ap \in AP$ is a (maybe empty) sequence of dereference operations applied to a location specified by a variables address, where

$$AP = \{(v, do_1, \ldots, do_n) \mid (v \in VAR_{SSA}) \wedge (do_1, \ldots, do_n \in DEROP)\}$$

denotes the set of access paths. Here $do_1, \ldots, do_n$ are the dereference operations that will be applied to the address of the variable v. The dereference operations are applied to the variables address in order of appearance, which means do_1 is first applied to (v) and do_2 is then applied to (v, do_1) and so on.

Like dereference operations, access paths can be separated into two different groups: those using floating dereference operations and those using only fixed dereference operations.

Definition 29 (floating and fixed access paths)

An access path ap is called fixed access path ($ap \in AP_{fix}$) if it contains only fixed dereference operations. Otherwise ap is called floating access path ($ap \in AP_{fl}$).

$$\begin{aligned} AP_{fix} &= \{(v, do_1, \ldots, do_n) \mid (v, do_1, \ldots, do_n) \in AP \ \wedge \\ &\qquad \{do_1, \ldots, do_n\} \subset DEROP_{fix}\} \\ AP_{fl} &= AP \Leftrightarrow AP_{fix} \end{aligned}$$

Floating access paths are used if the offsets of one or more of the corresponding dereference operations cannot be specified at compile time. This means that a floating access path summarises a complete set of possible fixed access paths, namely all those that can be obtained by replacing all unknown offsets by constant ones. In this case we say that the floating access path covers the corresponding fixed access path.

Definition 30 (covering access path)

A (possibly floating) access path ap is said to cover a fixed access path ap' ($ap' \in cover(ap)$) if they only differ in the unknown offsets occurring in ap[21].

$$cover : AP \Leftrightarrow\!\!\!\rightarrow \mathbb{P}(AP_{fix})$$

$$cover((v, do_1, \ldots, do_n)) = \{(v, do'_1, \ldots, do'_n) \in AP_{fix} \mid \forall(1 \leq i \leq n) \ : \\ (do_i = do'_i) \ \vee \ ((width(do_i) = width(do'_i)) \ \wedge \ (offset(do_i) = \text{'?'}))\}$$

In some cases it is useful to be able to extend an existing access path. Given an access path ap and a sequence of dereference operations then the function $concat$ can be used to produce an access path ap' having the given additional dereference operations.

Definition 31 (concatenating access paths)

$$\begin{aligned} concat &: AP \times DOSEQ \Leftrightarrow\!\!\!\rightarrow AP \\ concat((v, do_1, \ldots, do_n), (do'_1, \ldots, do'_{n'})) &= (v, do_1, \ldots, do_n, do'_1, \ldots, do'_{n'}) \\ \text{where} \quad DOSEQ &= \cup_{n=0}^{\infty} DEROP^n \end{aligned}$$

[21]Note that this definition implies that fixed access paths may as well cover an access path. Each fixed access path covers exactly one access path: itself.

$$\begin{array}{lll} \text{I} & : & *(((\textbf{int}\ *)\ \texttt{x}) + \texttt{1}) \ = \ \ldots; \\ \text{II} & : & *(((\textbf{char}\ *)\ \texttt{x}) + \texttt{3}) \ = \ \ldots; \end{array}$$

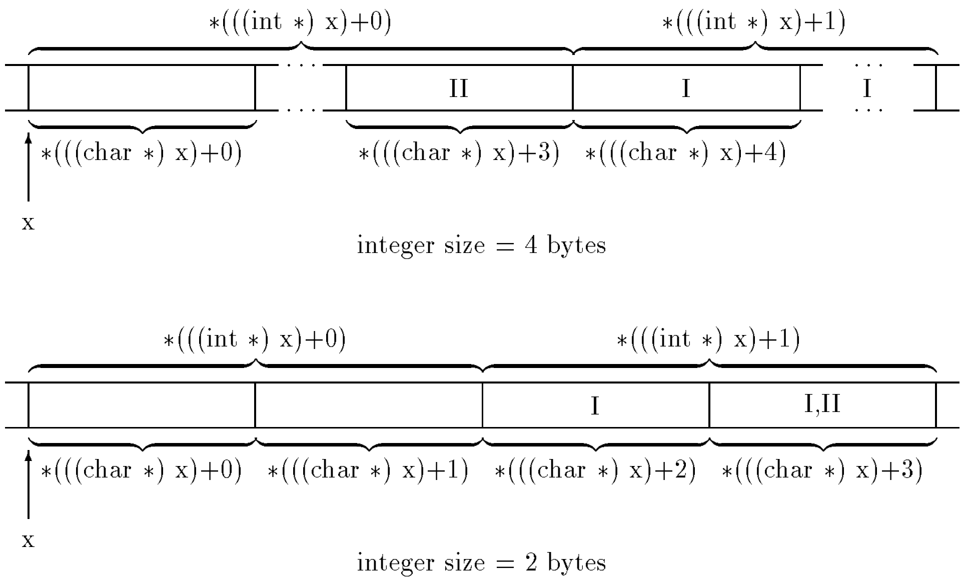


Figure 3.19: Memory layout for different integer sizes

Fixed access paths can be used to represent base expressions. Since they use the sizes of objects as well as the offsets calculated in bytes, access paths are not independent of the used machine architecture. This is necessary to avoid overly conservative assumptions when type casts occur in the analysed program. E.g. the two statements shown in figure 3.19 modify the same storage location if the size of an integer is two bytes, whereas they do not if the size is four bytes. In this example the locations that are modified are marked with the number of the corresponding statement. Only when an integer size of two is assumed, do the sets which contain the modified storage locations of the two statements intersect. When situations like the one described above arise, there are only two choices for the analyser: conservatively assume that the offset is unknown or use the machine dependent type size information. Since type casts are frequently used within C programs, the second alternative was chosen to avoid unnecessary imprecision.

Before further details concerning access paths are discussed, a function which returns the fixed access path corresponding to a base expression will be defined.

Definition 32 (fixed access paths representing base expressions)
Given a base expression $exp \in EXP_{base}$ then the function

$$acc\_path \ : \ EXP_{base} \nrightarrow AP_{fix}$$

returns the fixed access path representing the expression *exp*.

Figure 3.20 shows some examples of base expressions and their corresponding access paths. In the cases where pointer variables are dereferenced, two dereference operations occur. The first dereference operation is always necessary to get the variables value (instead of its address), which becomes clearer if one looks at expressions that explicitly use the variables address instead of the variable itself: e.g. $*(*(\&\texttt{iptr} + \texttt{0}) + \texttt{1})$ is more or less equivalent to `iptr`[1], but makes the first dereference step visible.

```
int i, *iptr, iarr[5];
char **c2ptr, c2arr[5][7];
struct {int i; float f;} s;
union {int i; float f;} u;
```

type	size [bytes]
character	1
integer	2
float	4
pointer	4

exp	*acc_path*(*exp*)
`&i`	(i)
`i`	$(i, (0,2))$
`*iptr`	$(iptr, (0,4), (0,2))$
`*(iptr + 1)`	$(iptr, (0,4), (2,2))$
`iarr[2]`	$(iarr, (0,4), (4,2))$
`c2ptr[2][3]`	$(c2ptr, (0,4), (8,4), (3,1))$
`c2arr[2][3]`	$(c2arr, (0,4), (17,1))$
`s.i`	$(s, (0,2))$
`s.f`	$(s, (2,4))$
`u.f`	$(u, (0,4))$
`*(((char*)i) + 3)`	$(i, (0,2), (3,1))$

Figure 3.20: Base expressions and their corresponding access paths

The C programming language does not allow the use of arrays as the destination of an assignment, and so they can never be assigned a new value. In fact C treats arrays much more like invariable pointers pointing to a certain location at the runtime stack. When they occur as the right hand side of an assignment their address (the above-mentioned pointer) is copied to a new storage location instead of copying the array's contents. Therefore the size of an array, when used in dereference operations, is always the same as the size of pointers. This size corresponds to the amount of storage that is copied if an array occurs on the right hand side of an assignment, which is much more suitable in this case.

Multi-dimensional arrays are handled in a comparatively unusual manner by the C programming language. This results in the fact that the two expressions accessing the two level pointer and the two-dimensional character array, which are shown in the example above, produce completely different access paths. In the first case there are three dereference operations, whereas in the second case there are only two dereference operations. This corresponds exactly to what the code generated by an ANSI-C compiler does in such situations. In case of multi-dimensional arrays the compiler generates an one dimensional array and transforms multi-dimensional array accesses to one-dimensional ones by using the formula shown in figure 3.21. This results in a single array access and hence needs only two dereference operations.

Such a transformation cannot be effected for the two level pointer since there are no known dimensions of the objects pointed to. Therefore the statement accessing the two level pointer needs an additional dereference operation.

Another assumption that has been made in the example given above concerns the alignment inside of structures. Here a byte-wise alignment was assumed. If e.g. a four-byte-wise alignment had been chosen, this would have led to a different access path for `s.f` since the offset of the second component would then have to be four bytes instead of just two. Besides which, the size of the whole structure would increase from six to eight bytes in this case.

Fact 33
Every base expression is represented by exactly one fixed access path, whereas a fixed access path may represent several different base expressions.

E.g. the expressions `a[3]`, `*(a + 3)`, `(a + 1)[2]` and `2*a[3]` are all represented by the same access path, because they all produce the same sequence of dereference operations. Assuming

Given the definition of a multi-dimensional array and a corresponding statement accessing the array like

$$\textit{type}\ \ \texttt{arr}[\texttt{s}_1][\texttt{s}_2]\ldots[\texttt{s}_n];$$

$$\ldots\ \ \texttt{arr}[\texttt{t}_1][\texttt{t}_2]\ldots[\texttt{t}_m]\ \ \ldots$$

with $m \leq n$, then the compiler allocates the same storage as if an one-dimensional array like

$$\textit{type}\ \ \texttt{arr}[\texttt{s}_1 \cdot \texttt{s}_2 \cdots \texttt{s}_n];$$

had been used. If this declaration were used, the statements accessing the array would have to be replaced by equivalent[22] statements like

$$\ldots\ \ \texttt{arr}[\textstyle\sum_{i=1}^{m}(\texttt{t}_i \cdot \prod_{j=i+1}^{n} \texttt{s}_j)]\ \ \ldots$$

if $m = n$ or by

$$\ldots\ \ \&(\texttt{arr}[\textstyle\sum_{i=1}^{m}(\texttt{t}_i \cdot \prod_{j=i+1}^{n} \texttt{s}_j)])\ \ \ldots$$

if $m < n$.

Figure 3.21: Multi-dimensional array access

that a is a four byte pointer to a character, this leads to the access path $(a, (0, 4), (3, 1))$. In the example given above the first three expressions are semantically equivalent, whereas the last one differs from the others because the value resulting from the pointer dereference operation is used as an operand of an arithmetic operation. Nevertheless the access path for this expression is the same as for the other ones. To distinguish between these different kinds of expressions we call expressions like the first three in our example minimal expressions. As already mentioned before, minimal base expressions which have the same access path always only differ in their syntax, but are semantically equivalent.

Definition 34 (minimal expression)

An expression is said to be minimal, if it either consists of a single variable, or if the outermost operator is one of the following: '$*_u$', '[ ]', ' . ', '$\Leftrightarrow\!>$', '$\&_u$'.

Minimal expressions are always closely related to the objects used in the program. In any case they either specify an addressable object or the address of an object. Non-minimal expressions specify values that are computed by the program, but are not related to objects. Examples are the sum or the product of two values, which are neither addresses nor addressable values in general.

Up to here, only the representation of base expressions has been discussed, and hence the next thing to be done is to think about a way to represent a simple expressions as well. As stated earlier, access paths have been defined in such a way that they can easily be used to represent base expressions. Simple expressions, however, cannot be represented by a single access path in general (if they are not base expressions). Nevertheless it is possible to use a set of access paths to represent simple (non-base) expressions as well.

[22]As stated in [Ame89a] these expressions access the same memory location but differ in their type if $m < n - 1$. In this case the original expression has a pointer to array type whereas the transformed one has the type pointer to *type*.

```
type₁ a[10], b;
type₂ x, y, z;
int n;
⋮
b = a[n];
x = y*z;
```

Figure 3.22: Program code containing simple non-base expressions

The difference between simple and base expressions is that simple expressions can contain multiple variables. The code shown in figure 3.22 contains two simple non-base expressions: a[n] and y∗z. Simple non-base expressions lead to the problem that there is no single access path that can be used to represent these expressions. In the case of y∗z there are two values that are accessed, and hence two access paths and two minimal expressions exist which correspond to these values. The corresponding simple expressions are y and z, the access paths belonging to these expressions are $(y, (0, \textbf{sizeof}(type_2)))$ and $(z, (0, \textbf{sizeof}(type_2)))$ respectively. However, arithmetic expressions like y∗z are almost always of no interest for the alias analysis since they are rarely used to specify memory locations. Expressions that are not used to specify memory locations will not produce or carry the information needed to produce new aliases. Therefore one might consider ignoring expressions like these. This will prove fatal when expressions using the operators '$+_b$' or '$\Leftrightarrow_b$' are taken into consideration as well, since they are used to manipulate pointers, and hence the expressions using one of these operators can be used to specify memory locations. In this case ignoring such expressions would make the whole analysis worthless since pointer manipulation is frequently used in C programs. In the case of a[n], which is equivalent to ∗(a + n), the sum of a and n is computed and then dereferenced. When a pointer and an integral typed value are added, the pointer arithmetic of the C programming language computes a new pointer of the same type as the input pointer which has a value specified by the following expression.

$$\texttt{a} + \texttt{n} \;\Leftrightarrow\; (type_1 *)\;(((\textbf{char}\;*)\;\texttt{a}) + (\texttt{n} \cdot \textbf{sizeof}(type_1)))$$

If the value of n were constant, and hence known at compile time, the corresponding access path would be $(a, (0, 4), (n \cdot \textbf{sizeof}(type_1), \textbf{sizeof}(type_1)))$. Since access paths which have offsets depending on the values of variables are not allowed, they cannot be used to represent the expression. However, floating access paths can be used in such cases: $(a, (0, 4), (\text{'?'}, \textbf{sizeof}(type_1)))$. Now, one might think that simple expressions could as well be represented by a single access path if unknown offset values are used. Unfortunately this is not true in general, since it is not always clear which value is the pointer value and which value is the offset, when two values are added. Since the C programming language allows type casts between pointers and integral types, one cannot even rely on the type of the expressions that are added.

The example program shown in figure 3.23 shows that integral types can hold pointer values as well. In this program the two integer values p1 and p2 hold 'pointers' to the two arrays a1 and a2. Later, either the address of a1[n] or a2[n] is assigned to p, depending on the value of c. Finally, the value of x is assigned to a1[n] or a2[n]. This surely is not the usual way a programmer would choose to gain this effect. However, there might be reasons to do things like this (e.g. an optimised user-defined memory access management). As can be seen from this example one does not know in general which value holds the address and which value holds the offset when values are added. As well it becomes clear that even

```
int a1[10], a2[10], x;
long int c, p, p1, p2, n;
p1 = (long int) &(a1[0]);
p2 = (long int) &(a2[0]);
if (...) {
   ⋮
   c = 1;
} else {
   ⋮
   c = 0;
}
p = (1 − c)*p1 + c*p2 + n*sizeof(int);
*((int *) p) = x;
```

Figure 3.23: Exchanging the role of pointers and offsets

other arithmetic operands can be (mis-)used to handle pointers. In our case `c` is used as an alternative to conditional statements. In the given context `(1 ⇔c)*p1 + c*p2` is equivalent to `((c == 0) ? p1 : p2)`.

Things like these are not explicitly allowed by the ANSI standard ([Ame89a]). Although the ANSI standard allows type casts between pointer and integral types, the exact effects of such type casts are not specified in detail. If the size of an integral type is big enough it is assumed that a pointer casted into an integral type that is later cast back into a pointer is still the same. Without this assumption the example given above will not lead to the expected results and will hence have undefined effects.

Altogether this example shows that, given any arithmetic expression, one cannot be sure that no pointer is hidden inside this expression. Therefore it has to be assumed that any value is a hidden pointer. Even if pointers are added to integral typed values one will have to take into consideration that the 'real' pointer value might be the value which has the integral type and vice versa due to preceding type casts. Of course the assumption that every integral type could somewhere be used as a pointer produces many more aliases than if this assumption would not have been made. Therefore a more precise algorithm, which does not make this assumption is also given in section 3.2.3.4. Altogether this leads to the fact that simple expressions cannot be represented by a single access path. However, it is still possible to represent simple expressions by a set of access paths instead.

Now, after it has been shown how simple expressions can be represented by access paths as well, an extended version of the function *acc_path* is defined. The function *acc_path_set* returns the set of access paths representing a simple expression. The function $acc\_path\_set_{fix}$ returns the same set, besides the fact that all floating access paths have been replaced by the set of fixed access paths covered by the corresponding floating access path.

Definition 35 (access paths representing simple expressions)
Given a simple expression the function

$$acc\_path\_set \quad : \quad EXP_{simple} \Leftrightarrow\!\!\rightarrow \mathbb{P}(AP)$$

returns the set of access paths corresponding to this expression. The access paths representing a simple expression can be found by ignoring all but one of

```
int i, *iptr;
long int l, *lptr;
char *cptr;
```

type	size [bytes]
character	1
integer	2
long integer	4
pointer	4

exp	*acc_path_set*(*exp*)
`cptr[i]`	$\{\ (cptr, (0,4), ('?',1))\ ,\ \ (i, (0,2), ('?',1))\ \}$
`cptr[l]`	$\{\ (cptr, (0,4), ('?',1))\ ,\ \ (l, (0,4), ('?',1))\ \}$
`*(lptr + l)`	$\{\ (lptr, (0,4), ('?',4))\ ,\ \ (l, (0,4), ('?',4))\ \}$
`iptr + l`	$\{\ (iptr, (0,4))\ ,\ \ (l, (0,4))\ \}$

Figure 3.24: Simple expressions and their corresponding access paths

the expressions variables and then computing the access path representing this base expression. Furthermore, all offsets depending on the values of one of the ignored variables have to be set to unknown.

The corresponding set of fixed access paths is returned by the function $acc\_path\_set_{fix}$.

$$\begin{aligned} acc\_path\_set_{fix} &: EXP_{simple} \leftrightarrow\!\!\!\!\rightarrow \mathbb{P}(AP_{fix}) \\ acc\_path\_set_{fix}(exp) &= \{ap \mid \exists(ap' \in acc\_path\_set(exp)) : ap \in cover(ap')\} \end{aligned}$$

To make things clearer, some example expressions and their corresponding sets of access paths are shown in figure 3.24. Note, that although the integer value `i` is not big enough to hold a pointer value the corresponding access path has as well been added to the set of access paths representing the expression.

The sets of fixed access paths have not been added to the table since they are either equivalent to the shown sets or infinitely large in the other case. This shows one of the major advantages of using floating access paths:

Fact 36

Given a simple expression *exp* then the set of access paths corresponding to this expression ($acc\_path\_set(exp)$) is always finite. However, the same does not hold for the set of fixed access paths representing the expression ($acc\_path\_set_{fix}(exp)$).

Now, that (possibly infinite sets of) fixed access paths can be used to represent base as well as other simple expressions, we can attach a value and a location to a fixed access path. The location represented by a fixed access path is the storage location that can be accessed by the corresponding minimal base expressions, whereas the value represented by a fixed access path is the value stored at this particular location. In general, neither the value nor the location remain unchanged during the execution of the program, since the values of variables or the values that can be referenced through variables may change. This implies that the values and locations that are represented by a certain access path can change as well while the program is being executed. The function *loc* returns the storage location that can be accessed by a fixed access path, and *val* returns the value that is stored at this location in a certain program state.

Definition 37 (locations and values belonging to a fixed access path)

$$loc \;:\; AP_{\mathit{fix}} \times STATE \nLeftrightarrow LOC \;\;,\;\; val \;:\; AP_{\mathit{fix}} \times STATE \nLeftrightarrow VAL$$

$$\text{with} \quad STATE \subseteq (LOC \rightarrow VAL) \times POS$$

$$\begin{array}{lcl} loc(ap, st) = l & \Leftrightarrow & l \text{ is the location specified by } ap \text{ in state } st \\ val(ap, st) = v & \Leftrightarrow & v \text{ is the value stored at the location specified by } ap \text{ in state } st \end{array}$$

As can be seen from the above definition the two functions are always related to a certain state of the program (st) since their values are not necessarily stable during the program execution.

Fact 38

If there are two different fixed access paths which lead to the same location, then the locations reachable by adding the same fixed dereference operations to both access paths are the same as well.

$$\begin{gathered} \exists((v, do_1, \ldots, do_n), (v', do'_1, \ldots, do'_{n'}) \in AP_{\mathit{fix}}, st \in STATE) \;\;: \\ loc((v, do_1, \ldots, do_n), st) = loc((v', do'_1, \ldots, do'_{n'}), st) \\ \Rightarrow \\ \forall(do''_1, \ldots, do''_{n''} \in DEROP_{\mathit{fix}}) \;\;: \\ loc((v, do_1, \ldots, do_n, do''_1, \ldots, do''_{n''}), st) = \\ loc((v', do'_1, \ldots, do'_{n'}, do''_1, \ldots, do''_{n''}), st) \end{gathered}$$

Fact 39

Given a simple expression *exp*, a program state *st* and the set of fixed access paths representing the expression ($acc\_path\_set_{\mathit{fix}}(exp)$), then at least one access path $ap \in acc\_path\_set_{\mathit{fix}}(exp)$ exists for every value $val \in VAL$ that can be accessed by *exp*, so that this value can be accessed by *ap* as well.

$$\begin{gathered} \text{The expression } exp \text{ can access the value } val \\ \Rightarrow \\ \exists(ap \in acc\_path\_set_{\mathit{fix}}(exp)) \;\;:\;\; val(ap, st) = val \end{gathered}$$

This is quite obvious since the access paths which represent a simple expression have been chosen in such a way to make sure that every value that can be accessed by a given expression can be accessed by at least one of the corresponding access paths as well.

Definition 40 (alias)

Two fixed access paths *ap* and *ap'* are aliases (*alias*(*ap*,*ap'*) = **true**), if they refer to the same storage location in a certain program state.

$$\begin{array}{rcl} & alias \;:\; AP_{\mathit{fix}} \times AP_{\mathit{fix}} \nLeftrightarrow \mathbb{B} & \\ alias(ap, ap') & = & \exists(st \in STATE) : loc(ap, st) = loc(ap', st) \end{array}$$

```
(1)  if (a > b) {                      if (a) b = f;
(2)    if (a == b) a = 1;              else c = f;
(3)    else b = 2;                     if (a) d = f;
(4)  }                                 else e = f;
```

Figure 3.25: Programs with non-realizable execution paths

Since aliases are always related to a certain program point, they can be divided into two groups: those aliases that occur every time when that certain program point is reached and those that occur only in some cases. The members of the first group are called must aliases, whereas the members of the second group are called may aliases. The function *must_alias* determines if a given alias is only a may alias or if it is a must alias.

Definition 41 (may and must alias)

$$\begin{aligned} \mathit{must\_alias} &: AP_{\mathit{fix}} \times AP_{\mathit{fix}} \Leftrightarrow \mathbb{B} \\ \mathit{must\_alias}(ap, ap') &= \exists (p \in POS) : \forall (st \in STATE) : \\ &\quad (pos(st) = p) \Rightarrow loc(ap, st) = loc(ap', st) \end{aligned}$$

Definition 42 (e-alias)
A variable v is an e-alias of a variable v', if there exist two aliased fixed access paths *ap* and *ap*' using these variables.

$$\begin{aligned} \mathit{e\_alias} &: VAR_{SSA} \times VAR_{SSA} \Leftrightarrow \mathbb{B} \\ \mathit{e\_alias}(v, v') &= \exists (ap, ap' \in AP_{\mathit{fix}}) : (alias(ap, ap') = \mathbf{true}) \wedge \\ &\quad (ap = (v, do_1, \ldots, do_n)) \wedge \\ &\quad (ap' = (v', do'_1, \ldots, do'_{n'})) \end{aligned}$$

If two variables are e-aliases, this means that at least one storage location exists that can be accessed from either variable by the use of base expressions.

Definition 43 (execution path)
Every path through a functions CFG graph that starts at the *START* node and ends with the *END* node is called an execution path.

Definition 44 (realizable / non-realizable execution path)
An execution path is called realizable if it corresponds to a possible execution of the function, and non-realizable otherwise[23].

[23]The term 'realizable execution path' has previously been used in [LR91] in relation with interprocedural control flow graphs (ICFG) to make sure the aliases introduced on different paths through the call graph are treated differently (call path sensitive analysis). Here it is used only for ordinary intraprocedural control flow graphs (CFG) to distinguish between paths through a function that really can occur and those that cannot.

Figure 3.25 shows two example programs where non-realizable execution paths occur. In the first program `a` is always greater than `b` when the **if** statement in the second line is reached. Therefore the assignment `a` = `1` in this line is never executed. The second program has two non-realizable execution paths. These execution paths are non-realizable because the two **if** statements in line 1 and 3 have the same condition. Therefore it is not possible that the assignment `d` = `f` will be executed after the assignment `c` = `f` was executed, or that the assignment `e` = `f` will be executed after the assignment `b` = `f` was executed. If the knowledge about these non-realizable execution paths is taken into account one could securely assume that `a` will not be modified after the execution of the first program, because the only statement that might modify `a`'s value does not appear on a realizable execution path. For the second program this additional information leads to the fact that neither `b` and `e` nor `c` and `d` may be pointers to the same location as long as they have not already been before the execution of the code.

3.2.1.2 Aliases computed by the algorithm

Since a precise alias analysis cannot be done within reasonable time, some restrictions have to be made. Which restrictions have been made and why they were chosen will be explained in the following. The following facts outline the kind of alias analysis that is performed.

When a function is being analysed

- it is assumed that every execution path is realizable
- track it is not kept of the conditions leading to a certain alias
- only may aliases are collected

As can be seen from the examples given previously, there may be non-realizable execution paths. Nevertheless, it is assumed that every execution path is realizable because the loss of precision that is caused by this assumption does not justify the additional costs resulting from detecting all non-realizable paths. If one would like to detect all non-realizable execution paths, it would be necessary to find out under which circumstances the expressions used as the conditions of statements controlling the control flow[24] become true or false. Since the alias calculation would then depend on the calculation of these conditions as well as the calculation of these conditions would depend on the aliases, it would be necessary to repeatedly compute both of them until stabilisation is reached. This would then lead to a considerable increase in costs of the algorithm.

To get even more precise results one would have to deal with conditional aliases instead of may aliases. This means that all the found aliases are related to a condition under which the alias may occur, instead of simply keeping track of those aliases that may occur. Besides the fact that every execution path is assumed to be realizable, track it is not kept of such conditions or of the execution paths on which the aliases depend. This means that the alias analysis is not execution path sensitive. In case there are two may aliases, each of them introduced by a different branch of an **if** statement, it is nevertheless (falsely) assumed that both of them can occur at the same time after the **if** statement was executed. The

[24]E.g. expressions used as the conditions of **if** or **while** statements

computation of conditional aliases would significantly increase the costs of the algorithm and has hence been omitted.

Since the computation of must aliases depends on the calculation of conditional aliases it has to be omitted as well. Strictly speaking a must alias analysis can only be performed if the analysis is execution path sensitive. Therefore only may aliases will be computed.

As stated in [Hor97], it is not possible to compute the smallest possible set of flow-insensitive may aliases with reasonable time and space costs anyway, and therefore some losses of precision in favour of reasonable costs are unavoidable. Furthermore, it was shown in [LR91] that in the presence of two level pointers the problem of determining precise intraprocedural may alias sets is $\mathcal{NP}$-hard.

There are other alias analysis algorithms that do not need the restrictions made here. E.g. in [CBC93] and [Deu94] execution path sensitive algorithms are presented. Here the aliases are propagated through the CFG graph. The restrictions that have been made here are strongly related to the fact that our analysis is based on a static memory model representation instead of a collection of aliased access paths or similar approaches.

The major advantage of our algorithm (that there are only very few restrictions to the C programming language) results in extreme increases in costs if the above-mentioned restrictions are not made. However, this is not very astonishing since there always is an interaction between precision, costs and restrictions of an algorithm and it is not possible to get along without some compromises.

3.2.2 Function interface graphs

Basically, there are two different concepts used by alias analysis algorithms: the first one is based on sets of aliased expressions that are propagated along the edges of the control flow graph, whereas the second one is based on some kind of memory representation. Function interface graphs belong to the second group. They can be used to find the aliases that are generated by a certain function. Once a function interface graph has been built it summarises all the effects to the memory that might occur during the function execution.

In many cases the function interface graph contains more information than necessary for the following interprocedural analysis parts. E.g. aliases between non-static local variables that may be introduced during the function execution are only of interest for the function itself but not for the rest of the program. This is because they are not able to affect the aliases between other variables of the calling function(s). In such cases it is sometimes possible to remove superfluous parts of the function interface graphs before they are combined during the interprocedural analysis parts. Nevertheless this has to be done with great care since storage locations that can be accessed by one of the non-static local variables could be affected by other variables as well. Therefore it is necessary that every possible modification of data accessible by global variables or by function parameters is still represented in the reduced function interface graph. Additionally, one has to represent the information about the parameters that are passed to those functions as well as the actual values of all global variables, since all data accessible through these parameters / variables could be modified by the called functions. This is necessary since the effects of other functions that are called by the actually processed function cannot yet be considered in general since it might be that these functions have not been analysed yet.

Our representation of a functions effects is based on values and references between those values. This is quite similar to the representation with storage locations like those used in [CWZ90, HPR89, LH88], but there are some advantages in representing values instead of locations. Anyway, there are only slight differences between values and locations in C

```
long int *iptr, i1, i2;
float f;
i1 = 5;
iptr = &i1;
f = (float) ((long int) iptr);
i2 = *((long int *) ((long int) f));
```

Figure 3.26: Float value used as a 'pointer'

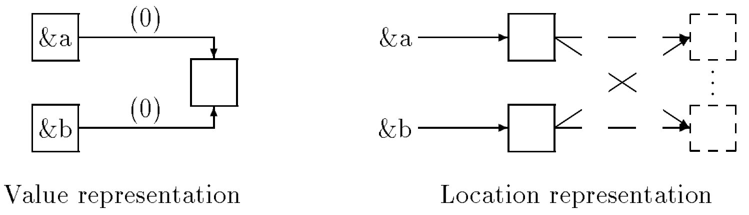


Figure 3.27: Representation using storage locations / values

since values are in many cases used as the addresses of locations (pointers). Since the use of pointer variables and cast operations is hardly restricted[25], every value must be supposed to be used as a pointer sometime. Therefore it is not possible to rely on the type of an expression, if one wants to determine whether it is a pointer or not. Even if a variable is an integer or a float, it may be transformed into a pointer value and be dereferenced then. Figure 3.26 shows a small example where a float is used to store the address of a storage location[26].

Figure 3.27 shows the differences between the representation using values and the one using locations for a simple assignment like a = b. If the value representation is used, there are two nodes for the addresses of the two variables and another one which holds the value of the variables itself. This ensures that whenever one of the two variables is accessed (e.g. by *a) the used value is the same as if the other variable was chosen. The annotations on top of the edges are the corresponding dereference operations. Here the last component (the size of the value) has been omitted. This is done to keep the example as simple as possible and hence easier to survey. Several of the following examples are reduced in a similar manner whenever parts of the information are not relevant.

If the location representation is used it becomes more complicated to represent the effects of the assignment statement. Since the locations where the values of a and b are kept do not change, the corresponding nodes remain unchanged as well. The only effect of the assignment is that the same data can be accessed by those variables after it has been executed. Therefore it would be necessary to ensure that every location reachable by b, will now be reachable by a with exactly the same sequence of dereference operations. Remember that if the locations reachable by a single dereference step are the same all the other locations reachable by multiple dereference steps are the same as well. Since it is not

[25]Many other programming languages (e.g. Pascal) are much more restrictive concerning their pointer manipulation constructs. Even if pointer manipulation is possible it is less frequently used in such languages because it is much more complicated. Unlike these languages, the C programming language encourages the user to use pointers extensively, and in many cases it is not even possible to avoid pointer usage.

[26]This is based on the assumption that the size of long integers is at least equal to the size of pointers and that type casts from integral types to pointers and back do not modify the pointer value.

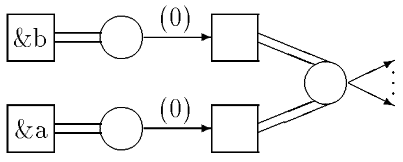


Figure 3.28: Representation of statement $*(\mathtt{a}+\mathtt{b})$ using value and reference nodes

known in advance which dereference operations will be used, it is comparatively complicated (and less efficient) to make sure that the accesses to other locations through a and b have the same effects. Another disadvantage of the second representation is that the addresses of variables neither are locations nor are they stored at any location. Hence they do not fit as naturally into the representation as they do when using the value representation.

Even if the value representation is used, there are some cases where it is necessary to make sure that the same values can be reached with the same sequence of dereference operations starting at two different values. This is quite similar to the problems with the location representation described above. E.g. if a statement like $*(\mathtt{a}+\mathtt{b}) = \mathtt{c}$ occurs one does not know exactly which variable plays the role of the pointer and which one is the offset value. Therefore both variables have to be treated alike, which means that both variables are treated as if they were the pointer value. Here pointer value means the value holding the address of a memory location. As long as the user does not ensure that only variables with pointer type hold such addresses, it is not possible to benefit from the variables type information. Only in cases where the user guarantees that no memory addresses are kept in non-pointer type variables, the variable with integer type can be safely ignored. In order to efficiently handle cases where such a promise is not given, two different kinds of nodes are used: value nodes and reference nodes. Value nodes represent the values itself whereas the reference nodes collect all the references between different value nodes.

If the case described above arises, and we have to make sure that two value nodes share the same references, this can be done by sharing the same reference node. Figure 3.28 shows an example where the values reachable by the variables a and b are exactly the same even though the values of the variables itself are not equal. Here and in the following examples boxes are used to represent value nodes, whereas circles are used for reference nodes. Since a and b share the same reference node, every future change to the values referenced by either variable automatically affects the values referenced by the other variable as well.

Definition 45 (function interface graph)

A function interface graph $fig \in FIG$ is an annotated directed multi-graph $fig = (V, E)$ where the functions

$$src \;:\; E \nrightarrow V \;\;,\;\; dst \;:\; E \nrightarrow V$$

return the source and destination node of an edge respectively. The node set

$$V \;=\; VN \cup RN \quad \text{with} \quad VN \cap RN = \varnothing$$

contains the two different types of nodes: value nodes (VN) and reference nodes (RN). The set of edges

$$\begin{aligned} E \;=\; & VRE \cup RVE \quad \text{with} \\ & \forall(e \in VRE) \;:\; src(e) \in VN \;\wedge\; dst(e) \in RN \quad \text{and} \\ & \forall(e \in RVE) \;:\; src(e) \in RN \;\wedge\; dst(e) \in VN \end{aligned}$$

contains two different types of edges: links (VRE) and reference edges (RVE). Furthermore, there are functions returning the annotations

$$\begin{array}{rcl} vars & : & VN \nrightarrow \mathbb{P}(VAR_{SSA}) \\ derop & : & RVE \nrightarrow DEROP \\ off & : & RVE \nrightarrow OFF \\ gen & : & RVE \nrightarrow GEN \ \text{, where } GEN = \{\,'i','a'\,\} \end{array}$$

and the following additional conditions hold

$$\begin{array}{lcl} \forall (vn \in VN) & : & \left| \{ e \mid e \in E \wedge src(e) = vn \} \right| \ \leq \ 1 \\ \forall (e, e' \in E, e \neq e') & : & src(e) \neq src(e') \ \vee \ dst(e) \neq dst(e') \ \vee \\ & & off(e) \neq off(e') \ \vee \ derop(e) \neq derop(e') \ \vee \\ & & gen(e) \neq gen(e') \qquad . \end{array}$$

Unlike usual graphs, multi-graphs do not allow identification of an edge by its source and destination node, since there may be several different edges connecting the same node. Hence it is necessary to have two functions which return the source and destination nodes for each edge (*src* and *dst*). As can be seen from the definition there are four functions used to access the different annotations of the function interface graph. The first of these functions (*vars*) accesses the annotations of the value nodes, whereas the following three functions, namely *derop*, *off* and *gen*, access the annotations of the reference edges. The meaning of these annotations will be explained in detail soon.

As can be seen from the definition, the function interface graph has two different kinds of nodes: value nodes (VN) and reference nodes (RN). As well there are two different kinds of edges: those pointing from a reference node to a value node and those pointing from a value node to a reference node. The edges pointing from reference nodes to value nodes are called reference edges (RVE), whereas the other ones are called links (VRE). There are no edges pointing from reference nodes to reference nodes or from value nodes to value nodes.

The last part of the definition shows two additional properties of function interface graphs: the first ensures that there is never more than one link connected to a value node, whereas the second ensures that even though there may be multiple reference edges connecting the same nodes, there will not be two such edges which have exactly the same annotations.

The reference nodes are only used to make it easier (and more efficient) to let value nodes share the same references. Logically they build a unit together with each value node that has a link connected to them, since they contain the references belonging to those value nodes. Therefore only one link always exists which is connected to a value node. Unlike reference edges the links do not represent dereference operations.

As already mentioned before, there are four functions which access the different annotations of the value nodes and reference edges. The first of these functions (*vars*) returns a set of variables annotated to each value node. This set relates the value nodes used to represent the values of variables to these variables. A variable is annotated to a value node if the value node is used to represent the address of that variable. Many value nodes do not represent a variables address, and hence the corresponding set is empty in this case.

Besides the annotations of the value nodes, the function interface graph holds some additional information which is annotated to the reference edges. This information can be accessed using one of the functions described above. First, there is the kind of dereference operation that belongs to the edge, which can be accessed using the *derop* function, second

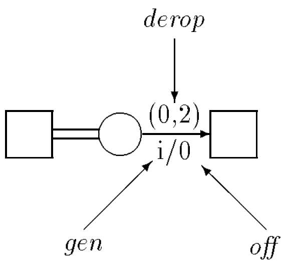


Figure 3.29: Information annotated to reference edges

```
int *a, *b;
a  =  b + 3;
```

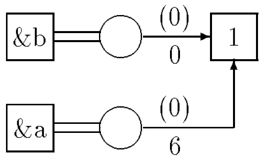


Figure 3.30: Assignment producing a non-zero offset value

there is an additional offset accessible by the *off* function, and finally there is the generation mode annotation which is returned by the *gen* function.

As a convention, the dereference operation is always written above the reference edges, whereas the value of the generation mode annotation and the additional offsets are written below the edges. As long as they are not really necessary, some of the annotations are omitted in the following examples. Only those annotations that are essential for the understanding are included to prevent the examples from getting unnecessarily complex. Figure 3.29 shows an example edge with all its annotations.

3.2.2.1 Offset annotations

The offset attached to the reference edges is useful when the sum of a value and a constant appears on the right hand side of an assignment. The example in figure 3.30 shows the graph that is generated for the assignment statement a = b + 3. Since the value assigned to a is not the value of b itself, but there is a constant (3) added to this value, this has to be taken into account as well[27]. This constant value is stored in the offset annotation of the edge pointing to the value of b (value node 1). By doing so, it is now possible to distinguish between an access via a and b. Even though the access to these variables leads to the same value node (1) in either case, they differ in the annotations along the paths that have been used. If a will be used later we can take advantage of the stored information.

The example from figure 3.31 shows what happens if an assignment using a (like *a = c) comes next. Since a was used instead of b, the newly generated reference edge does not have a dereference operation annotation (0), as it would have had for b, but (6) instead. This reflects the fact that both a[0] and b[3] access exactly the same memory location.

3.2.2.2 Generation mode annotations

During the building process of the function interface graph there are two different situations under which a new edge is created and added to the graph. In the first case a value was

[27]Like in the previous examples we assume an integer size of two bytes, which leads to an overall offset of $3 \cdot 2 = 6$ bytes.

```
int *a, *b, c;
a  =  b + 3;
*a  =  c;
```

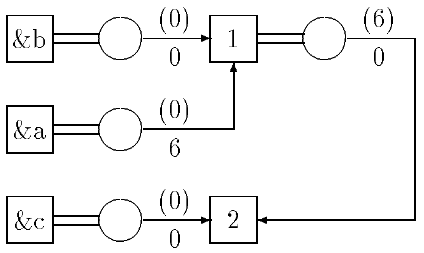


Figure 3.31: Assignment using the non-zero offset value

```
*a = b
```

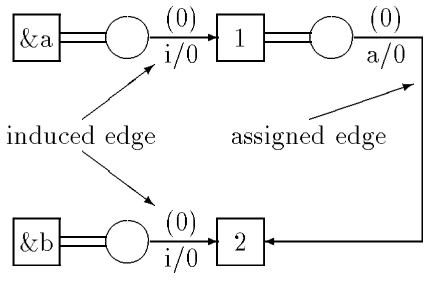


Figure 3.32: Induced and assigned reference edges

dereferenced, and there was no suitable value node and edge representing that particular dereference operation available. In this case a new value node is generated and connected to the value node that is to be dereferenced via a new reference edge. Edges like this newly generated one are then called *induced* edges. The second case are those edges that are generated when the effect of an assignment has to be represented. Assignment statements assign a new value to exactly one location[28]. This is represented by adding a new reference edge from the value node corresponding to the address of this location to the value node corresponding to the newly assigned value. Such an edge is then called *assigned* edge.

Fact 46

An assignment statement can cause the creation of arbitrary many induced edges, but there will always be only one assigned edge that is created.

The example shown in figure 3.32 demonstrates which kinds of edges are created for an assignment statement like $*\mathtt{a} = \mathtt{b}$, under the precondition that no information on both variables is available yet[29]. In this assignment the values of a and b are read, whereas the value of $*\mathtt{a}$ is modified (set to the value of b). As can be seen from the example, assigned edges are marked with an 'a', whereas induced edges are marked with an 'i'. Since no previous information on a and b was available, two induced edges had to be generated to make the values of a and b (value nodes 1 and 2) accessible. After a is accessible, the effect of the assignment, namely changing the value of $*\mathtt{a}$, can be handled by adding a reference edge from value node 1 to value node 2. In fact there is not really a reference edge between

[28]The C programming language allows multi assignment statements like $\mathtt{a} = \mathtt{b} = \mathtt{c}$, but those statements are transformed into multiple separate statements by the SUIF compiler. Therefore it is not necessary to take such effects into account. Under this condition it can be safely assumed that every statement contains at most one assignment operator, and accordingly there is only one value changed then. Note that statements like $\mathtt{a}++$ are assignment statements as well.

[29]This means that there are no nodes representing the values of either a or b yet.

these value nodes. Instead the reference edge connects the reference node belonging to value node number 1 with the value node number 2. For reasons of simplicity, we nevertheless speak of value nodes being connected by reference edges in such situations.

3.2.3 Building the function interface graph for statement sequences belonging to the same CFG node

When a function is analysed this is done within two major passes. The first pass analyses the code sequences belonging to a single CFG node and builds the corresponding function interface graphs. During the second pass, which will be described in section 3.2.4, the function interface graphs of the different CFG nodes are merged.

When the process of analysing a sequence of statements belonging to the same CFG node starts, the corresponding function interface graph is empty. Even though there might be some information about the used variables available from other previously analysed CFG nodes, this information is not taken into account at this stage. Instead this is left to the second pass where the function interface graphs are merged. Whenever a previously unused value is used, new nodes, links and edges are added to the graph if necessary.

Since assignment statements are the only kind of statements at this stage that can introduce new aliases, it is sufficient to analyse only these statements. Of course, function calls can introduce new aliases as well, but those aliases are handled during the interprocedural part of the analysis. This leaves the assignment statements as the only ones of interest for the moment. Other statements like jump and branch instructions do not have to be taken into account, because their effects are already summarised by the CFG graph.

3.2.3.1 The attribute grammar generating the function interface graphs

The algorithm used to compute the function interface graphs for a statement sequence will now be defined. This is done using the previously described extension of attribute grammars. As already mentioned before, attribute grammars do not specify the order in which the computations of the attributes have to be performed. A corresponding order is given right after the definition of the attribute grammar.

The grammar does not take operator precedence, l-value or type information into account, since it is only used to describe the algorithm that is used after the syntax tree has already been built. All the necessary type and l-value checks have been performed at that stage and the operator precedence is implicitly given by the tree structure. This prevents the grammar from getting unnecessarily complex. Only such matters are handled by the grammar that are directly related to the alias analysis, and all the standard problems like l-value and type checks can be safely ignored here.

The language that is described by the grammar consists of sequences of simple assignment statements. Not all operators that are used in the C programming language occur in the grammar definition. This is done because

- the operator is only a syntactical shortcut for a sequence of other operators ('++', '⇔⇔', '+=', '⇔=', '*=', '/=', '%=', '<<=', '>>=', '&=', '|=', '^=', '[ ]', '⇔>')

or

- the operator produces side-effects, respectively, takes influence on the control flow ('( )', '? :')[30].

[30]Here '( )' stands for the function call operator. This should not be mistaken with the cast operator which is written as '(...)'.

Expressions containing the first mentioned kind of operators can be treated correctly by simply transforming them into equivalent expressions not using these operators. In the second case, the statements have already been normalised (see chapter 2) and can therefore be ignored here. Besides that, the **sizeof** operator does not occur in the grammar, since it is assumed that it has already been replaced by the corresponding constant value. Further the cast operator '(...)' is treated as an unary operator. The second parameter (the type to which the expression will be converted) is left out since it will not influence the analysis. Since the type of all (sub-)expressions is already determined before the algorithm starts, it is not necessary to analyse the part of the type cast that specifies the destination type.

There is only one thing left that needs exceptional handling by the grammar: dynamic memory allocation. There are three functions that can be used to allocate a new block of memory: `malloc`, `calloc` and `realloc`. Unlike usual function calls that will be handled during the interprocedural part of the analysis, these functions have to be taken into consideration by the grammar as well. The first two of them return a new and uninitialised block of memory, whereas the third one returns a new block of memory that has been initialised by copying the memory from the old block. The grammar treats `malloc` and `calloc` as unary operators (*ALLOC*) where the argument specifies the size of the memory block to be allocated[31]. Therefore `calloc` has not to be treated explicitly, and it is sufficient to have one operator representing both functions. `realloc` is treated as a binary operator (*REALLOC*) where the first argument is the old memory block and the second argument is the new size.

$$
\begin{array}{rcll}
G & = & (T, N, SEQ, P) \text{ with} & \\
T & = & \{ & CONST,\ VAR,\ COMP,\ ALLOC,\ REALLOC,\ \text{'\~{}'},\ \text{'}+_u\text{'},\ \text{'}\Leftrightarrow_u\text{'},\ \text{'!'}, \\
 & & & \text{'}*_u\text{'},\ \text{'}\&_u\text{'},\ \text{'.'},\ \text{'(...)'},\ \text{'}+_b\text{'},\ \text{'}\Leftrightarrow_b\text{'},\ \text{'}*_b\text{'},\ \text{'/'},\ \text{'\%'},\ \text{'<<'},\ \text{'>>'},\ \text{'}\&_b\text{'}, \\
 & & & \text{'|'},\ \text{'\^{}'},\ \text{'>'},\ \text{'<'},\ \text{'>='},\ \text{'<='},\ \text{'=='},\ \text{'!='},\ \text{'\&\&'},\ \text{'||'},\ \text{'='}\ \} \\
N & = & \{ & SEQ,\ STMT,\ VAL,\ UNOP,\ BINOP\ \} \\
P & = & \{ & SEQ \ \Leftrightarrow\!\rightarrow\ STMT \mid STMT\ SEQ\ , \\
 & & & STMT \ \Leftrightarrow\!\rightarrow\ VAL\ \text{'='}\ VAL\ , \\
 & & & VAL \ \Leftrightarrow\!\rightarrow\ VAR \mid CONST \mid UNOP\ VAL \mid \text{'}\&_u\text{'}\ VAL \mid \\
 & & & \qquad \text{'}*_u\text{'}\ VAL \mid ALLOC\ VAL \mid VAL\ \text{'.'}\ COMP \mid \\
 & & & \qquad VAL\ BINOP\ VAL \mid REALLOC\ VAL\ VAL\ , \\
 & & & UNOP \ \Leftrightarrow\!\rightarrow\ \text{'\~{}'} \mid \text{'}+_u\text{'} \mid \text{'}\Leftrightarrow_u\text{'} \mid \text{'!'} \mid \text{'(...)'}\ , \\
 & & & BINOP \ \Leftrightarrow\!\rightarrow\ \text{'}+_b\text{'} \mid \text{'}\Leftrightarrow_b\text{'} \mid \text{'}*_b\text{'} \mid \text{'/'} \mid \text{'\%'} \mid \text{'<<'} \mid \text{'>>'} \mid \text{'}\&_b\text{'} \mid \\
 & & & \qquad \text{'|'} \mid \text{'\^{}'} \mid \text{'>'} \mid \text{'<'} \mid \text{'>='} \mid \text{'<='} \mid \text{'=='} \mid \text{'!='} \mid \\
 & & & \qquad \text{'\&\&'} \mid \text{'||'}\ \}
\end{array}
$$

The sets A_{inh}, A_{syn} and A_{int} containing the inherited, synthesised and intrinsic attributes will be defined next.

$$A = A_{inh} \cup A_{syn} \cup A_{int}$$

$$
\begin{array}{rcll}
A_{inh} & = & \{ & VAL.deref,\ VAL.str\_off,\ VAL.csize\ \} \\
A_{syn} & = & \{ & VAL.addr,\ VAL.val,\ VAL.add\_off,\ VAL.derop,\ VAL.cval\ \} \\
A_{int} & = & \{ & CONST.cval,\ CONST.strid,\ UNOP.ufunc,\ BINOP.bfunc,\ VAL.type, \\
 & & & COMP.str\_off,\ VAR.id\ \}
\end{array}
$$

[31]`calloc(n,t)` can in any case be transformed into `malloc(n` $\cdot$ **sizeof**`(t))`. The only remaining difference between these two function calls is that `calloc` initialises the memory by setting each byte to zero. However, since this does not affect the analysis this effect can be ignored.

The purposes of these attributes are shortly described below. Further the range of the values is defined for every attribute.

- $\mathit{VAL.addr} \in (\mathit{VN} \cup \{\mathit{nil}\})$: value node belonging to the address of the sub-expression (if the sub-expression holds a value having an address)
- $\mathit{VAL.val} \in (\mathit{VN} \cup \{\mathit{nil}\})$: value node belonging to the value of the sub-expression
- $\mathit{VAL.add\_off} \in \mathit{OFF}$: the constant value added to or subtracted from a value
- $\mathit{VAL.derop} \in (\mathit{DEROP} \cup \{\mathit{nil}\})$: last applied dereference operation of a sub-expression (if one exists)
- $\mathit{VAL.cval}$, $\mathit{CONST.cval} \in (\mathbb{R} \cup \{\mathit{nil}\})$: constant value[32] (if the sub-expression is an integer or float constant)
- $\mathit{CONST.strid} \in \mathbb{N}$: unique number identifying the n'th string constant in a program, value is zero for integer and floating point constants
- $\mathit{VAL.deref} \in \mathbb{B}$: indicates if the sub-expression has to be dereferenced (this has to be done if it is used as the argument of a '$*_u$' operator or occurs on the right hand side of an assignment)
- $\mathit{VAL.str\_off}$, $\mathit{COMP.str\_off} \in \mathbb{N}_0$: the offset used to get to the position of a component inside a structure or union[33] measured in bytes
- $\mathit{VAL.csize} \in \mathbb{N}$: the size of the sub-component if a component of a structure or union is processed, the size of the actual sub-expression's type otherwise
- $\mathit{UNOP.ufunc} \in (\mathbb{R} \rightarrow \mathbb{R})$: the unary function belonging to the unary operator
- $\mathit{BINOP.bfunc} \in (\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R})$: the binary function belonging to the binary operator
- $\mathit{VAL.type} \in \mathit{TYPE}$: the type of a sub-expression where TYPE is the set of possible types used by the C programming language
- $\mathit{VAR.id} \in \mathit{ID}$: a reference to an identifier contained in a symbol table[34]

Now the attribute computations as well as the plain computations will be described. In the following only those productions are listed that have a non-empty *AC* or *PC* set. All other productions do not have any computations related to them. Furthermore all *AC* and *PC* sets that are not explicitly mentioned here are assumed to be empty. Symbols which occur more than once in a production are marked with subscripts to distinguish between them.

[32]Here $\mathbb{R}$ is used as the set of floating point values that can be represented by a computer, which is in fact only a finite and hence enumerable subset of the 'real' $\mathbb{R}$.

[33]Since the components of unions overlap and use the same storage the position is always zero in this case.

[34]Since the same identifier can be used for more than one variable if they are defined in different scopes, a reference to the symbol table is used to identify the variables. This ensures that even those variables with the same name can be clearly identified by this attribute.

The attribute computations modify the function interface graph that represents the effects of the corresponding statements. It is assumed that *fig* is the function interface graph currently being processed. Hence all nodes and edges generated by the grammar rules are added to this graph.

A closer look at the following attribute computations shows that there are dependencies between synthesised attributes of the same symbol. However, they do not cause any problems because they do not lead to circular dependencies. In fact they could be avoided if necessary, but it makes the attribute computations smaller and easier to understand if they are used nevertheless. They do not influence the order in which the rules are applied because all these dependencies are non-circular and local to the attribute computations, which means that they do not lead to new dependencies between the different rules. There are several possible ways to compute these attributes. In any case they can be computed in order of appearance.

$$
\begin{array}{lll}
\multicolumn{3}{l}{AC(STMT \longrightarrow VAL_1 \text{ '=' } VAL_2) \; = \; \{} \\
\quad VAL_1.deref & = & \mathbf{false} \\
\quad VAL_2.deref & = & \mathbf{true} \\
\quad VAL_1.str\_off & = & 0 \\
\quad VAL_2.str\_off & = & 0 \\
\quad VAL_1.csize & = & size(VAL_1.type) \\
\quad VAL_2.csize & = & size(VAL_2.type) \\
\} & & \qquad\qquad \boxed{\text{I}}
\end{array}
$$

$$
\begin{array}{l}
PC(STMT \longrightarrow VAL_1 \text{ '=' } VAL_2) \; = \; \{ \\
\quad assign(VAL_1.addr, VAL_1.derop, VAL_2.add\_off, VAL_2.val) \\
\} \qquad\qquad \boxed{\text{II}}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{AC(VAL \longrightarrow VAR) \; = \; \{} \\
\quad VAL.addr & = & var\_node(VAR.id, fig) \\
\quad VAL.val & = & \mathbf{if}\ (VAL.deref)\ \ deref(VAL.addr, (VAL.str\_off, VAL.csize)) \\
 & & \mathbf{else}\ \ nil \\
\quad VAL.add\_off & = & \mathbf{if}\ (VAL.val \neq nil)\ \ get\_off(VAL.addr, (VAL.str\_off, VAL.csize)) \\
 & & \mathbf{else}\ \ VAL.str\_off \\
\quad VAL.derop & = & (VAL.str\_off, VAL.csize) \\
\quad VAL.cval & = & nil \\
\} & & \qquad\qquad \boxed{\text{III}}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{AC(VAL \longrightarrow CONST) \; = \; \{} \\
\quad VAL.addr & = & nil \\
\quad VAL.val & = & get\_str(CONST.strid) \\
\quad VAL.add\_off & = & \text{'?'} \\
\quad VAL.derop & = & nil \\
\quad VAL.cval & = & CONST.cval \\
\} & & \qquad\qquad \boxed{\text{IV}}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{AC(VAL_1 \longrightarrow UNOP\ \ VAL_2) \; = \; \{} \\
\quad VAL_1.addr & = & nil \\
\quad VAL_1.val & = & VAL_2.val \\
\quad VAL_1.add\_off & = & \mathbf{if}\ (UNOP.ufunc = (\ldots))\ \ VAL_2.add\_off\ \ \mathbf{else}\ \ \text{'?'} \\
\quad VAL_1.derop & = & nil \\
\quad VAL_1.cval & = & \mathbf{if}\ (VAL_2.cval \neq nil)\ \ UNOP.ufunc(VAL_2.cval)\ \ \mathbf{else}\ \ nil \\
\quad VAL_2.deref & = & VAL_1.deref \\
\quad VAL_2.str\_off & = & 0 \\
\quad VAL_2.csize & = & size(VAL_2.type) \\
\} & & \qquad\qquad \boxed{\text{V}}
\end{array}
$$

```
$AC(VAL_1 \longrightarrow \text{'}*_u\text{'}\ \ VAL_2) = \{$
    $VAL_1.addr$      $=$  $VAL_2.val$
    $VAL_1.val$       $=$  $\textbf{if}\ (VAL_1.deref)$
                             $deref(VAL_2.val, (VAL_2.add\_off + VAL_1.str\_off, VAL_1.csize))$
                           $\textbf{else}\ nil$
    $VAL_1.add\_off$  $=$  $\textbf{if}\ (VAL_1.val \neq nil)$
                             $get\_off(VAL_2.val, (VAL_2.add\_off + VAL_1.str\_off, VAL_1.csize))$
                           $\textbf{else}\ \text{'?'}$
    $VAL_1.derop$     $=$  $(VAL_2.add\_off + VAL_1.str\_off, VAL_1.csize)$
    $VAL_1.cval$      $=$  $nil$
    $VAL_2.deref$     $=$  $\textbf{true}$
    $VAL_2.str\_off$  $=$  $0$
    $VAL_2.csize$     $=$  $size(VAL_2.type)$
$\}$                                                                                VI
```

```
$AC(VAL_1 \longrightarrow \text{'}\&_u\text{'}\ \ VAL_2) = \{$
    $VAL_1.addr$      $=$  $nil$
    $VAL_1.val$       $=$  $VAL_2.addr$
    $VAL_1.add\_off$  $=$  $VAL_2.add\_off$
    $VAL_1.derop$     $=$  $nil$
    $VAL_1.cval$      $=$  $nil$
    $VAL_2.deref$     $=$  $\textbf{false}$
    $VAL_2.str\_off$  $=$  $0$
    $VAL_2.csize$     $=$  $size(VAL_2.type)$
$\}$                                                                                VII
```

```
$AC(VAL_1 \longrightarrow ALLOC\ \ VAL_2) = \{$
    $VAL_1.addr$      $=$  $nil$
    $VAL_1.val$       $=$  $new\_value\_node(fig)$
    $VAL_1.add\_off$  $=$  $0$
    $VAL_1.derop$     $=$  $nil$
    $VAL_1.cval$      $=$  $nil$
    $VAL_2.deref$     $=$  $\textbf{true}$
    $VAL_2.str\_off$  $=$  $0$
    $VAL_2.csize$     $=$  $size(VAL_2.type)$
$\}$                                                                                VIII
```

```
$AC(VAL_1 \longrightarrow VAL_2\ \ \text{'.'}\ \ COMP) = \{$
    $VAL_1.addr$      $=$  $VAL_2.addr$
    $VAL_1.val$       $=$  $VAL_2.val$
    $VAL_1.add\_off$  $=$  $\textbf{if}\ (VAL_1.deref)\ 0\ \textbf{else}\ VAL_2.add\_off + COMP.str\_off$
    $VAL_1.derop$     $=$  $VAL_2.derop$
    $VAL_1.cval$      $=$  $nil$
    $VAL_2.deref$     $=$  $VAL_1.deref$
    $VAL_2.str\_off$  $=$  $COMP.str\_off + VAL_1.str\_off$
    $VAL_2.csize$     $=$  $size(VAL_1.type)$
$\}$                                                                                IX
```

```
$AC(VAL_1 \longrightarrow VAL_2\ \ BINOP\ \ VAL_3)\ =\ \{$
    $VAL_1.addr$      $=$  $nil$
    $VAL_1.val$       $=$  if $((VAL_2.cval = nil)$ and $(VAL_3.cval = nil))$
                             $bin\_join(VAL_2.val,\ VAL_3.val)$
                           else if $(VAL_2.cval = nil)$ $VAL_2.val$
                           else if $(VAL_3.cval = nil)$ $VAL_3.val$
                           else $nil$
    $VAL_1.add\_off$  $=$  if $(BINOP.bfunc \in \{+,-\})$
                             if $((VAL_2.cval = nil)$ and $(VAL_3.cval \neq nil)$ and
                                 $(VAL_2.add\_off \neq\ '?'))$
                               if $(is\_pointer(VAL_2.type))$
                                 $BINOP.bfunc(VAL_2.add\_off, ptr\_size(VAL_2.type) \cdot VAL_3.cval)$
                               else $BINOP.bfunc(VAL_2.add\_off,\ VAL_3.cval)$
                             else if $((VAL_2.cval \neq nil)$ and $(VAL_3.cval = nil)$ and
                                       $(VAL_3.add\_off \neq\ '?'))$
                               if $(is\_pointer(VAL_3.type))$
                                 $BINOP.bfunc(VAL_3.add\_off, ptr\_size(VAL_3.type) \cdot VAL_2.cval)$
                               else $BINOP.bfunc(VAL_3.add\_off,\ VAL_2.cval)$
                             else $'?'$
                           else $'?'$
    $VAL_1.derop$     $=$  $nil$
    $VAL_1.cval$      $=$  if $((VAL_2.cval \neq nil)$ and $(VAL_3.cval \neq nil))$
                             $BINOP.bfunc(VAL_2.cval,\ VAL_3.cval)$
                           else $nil$
    $VAL_2.deref$     $=$  $VAL_1.deref$
    $VAL_3.deref$     $=$  $VAL_1.deref$
    $VAL_2.str\_off$  $=$  $0$
    $VAL_3.str\_off$  $=$  $0$
    $VAL_2.csize$     $=$  $size(VAL_2.type)$
    $VAL_3.csize$     $=$  $size(VAL_3.type)$
$\}$                                                                     $\boxed{\text{X}}$
```

```
$AC(VAL_1 \longrightarrow REALLOC\ \ VAL_2\ \ VAL_3)\ =\ \{$
    $VAL_1.addr$      $=$  $nil$
    $VAL_1.val$       $=$  $realloc\_val\_node(VAL_2.val)$
    $VAL_1.add\_off$  $=$  $'?'$
    $VAL_1.derop$     $=$  $nil$
    $VAL_1.cval$      $=$  $nil$
    $VAL_2.deref$     $=$  true
    $VAL_2.str\_off$  $=$  $0$
    $VAL_2.csize$     $=$  $size(VAL_2.type)$
    $VAL_3.deref$     $=$  true
    $VAL_3.str\_off$  $=$  $0$
    $VAL_3.csize$     $=$  $size(VAL_3.type)$
$\}$                                                                     $\boxed{\text{XI}}$
```

Dynamic memory allocation / deallocation

As already mentioned before, the functions that are used to allocate or deallocate memory need exceptional treatment differing from the treatment of other functions. The effects of the functions `malloc`, `calloc` and `realloc` are handled by the grammar rules. On the contrary, the effects of the function `free` do not have to be treated by the grammar. This is because the `free` function does not affect the function interface graph, and hence calls of `free` can be ignored completely.

The two rules related to dynamic memory allocation (VIII and XI) are rather simple. In case of memory allocation (rule VIII) only a new value node has to be generated to represent the values contained in the newly allocated memory area. The rule handling `realloc` statements (XI) is only slightly more complicated. Since it is possible that values stored in the previously allocated memory block are used as pointers pointing to other locations, it could be that these values will again be used to access these memory locations after they have been copied to the new memory block. Therefore it is necessary that the value nodes representing the new and the old memory block share the same reference node after the `realloc` statement has been processed. The definition and explanation of the *realloc_val_node* function that will follow soon show the further details.

As can be seen there is no big effort necessary to integrate the concept of dynamic memory allocation. Dynamic memory allocation causes that few problems because it fits quite naturally into the concept of function interface graphs.

Exemplary grammar rule explanation

Since the different purposes of the various attributes have already been explained, not every single attribute grammar rule is explained explicitly. Nevertheless one rule is picked out as an example and explained in detail in the following, just to make things clearer.

One of the more interesting rules is rule VII which handles the address operation. The expression related to this rule (VAL_1) applies the address operation to a sub-expression (VAL_2). The address operator returns the address of the memory location that holds the value of the sub-expression. This is reflected by the fact that the *val* attribute of the expression receives the value of the *addr* attribute of the sub-expression. The value returned by the address operator is not an addressable value itself. Therefore the *addr* attribute of the expression is set to *nil*. Since the address operation cannot only be applied to simple variables, but as well to structure and union components, the additional offset resulting from non-null component offsets of structure components has to be taken into account here. This is done by copying the *add_off* attribute of the expression to the attribute of its sub-expression. The *derop* attribute as well as the *cval* attribute are set to *nil* since the last operation neither was a dereference operation nor was the expressions value an integer or float constant. Further the *deref* attribute is set to **false** because the sub-expression does not has to be dereferenced. Since the sub-expression does not describe the component of a structure or union, its *str_off* attribute can be set to 0. Finally, the size of the sub-expressions type is assigned to the *csize* attribute of the sub-expression.

Now the definition of the attribute grammar is almost complete, and the only things still missing are the definitions of the functions that have been used by the grammar computations. These function definitions will be given and explained in the following.

Functions used by the different grammar computations

There are various functions used by the grammar computations that have not been explained yet. Besides the functions used by the grammar computations there are two more functions that have to be defined since they are used by the other functions. These functions are the *unite* and the *get_ref_node* function.

Besides some rather complex functions that will be defined and explained in detail soon, there are as well some very simple functions that will not be defined. For the latter only a short description of their purposes will be given. The following list gives an overview of the purposes of these simple functions.

size(t)	:	returns the size of type t
ptr_size(t)	:	returns the size of the type pointed to by the pointer type t
is_ordinal(t)	:	checks whether type t is an ordinal type (character, integer or float) or not
is_pointer(t)	:	checks whether type t is a pointer or non-pointer type[35]
get_str(n)	:	returns the value node belonging to a string constant identified by the number n, returns *nil* for non-strings
new_value_node(fig)	:	adds a new value node to *fig* and returns it
new_reference_node(fig)	:	adds a new reference node to *fig* and returns it
new_link(fig)	:	adds a new link to *fig* and returns it
new_reference_edge(fig)	:	adds a new reference edge to *fig* and returns it
delete_value_node(fig, n)	:	deletes the value node n from *fig*
delete_reference_node(fig, n)	:	deletes the reference node n from *fig*
delete_link(fig, l)	:	deletes the link l from *fig*
delete_reference_edge(fig, e)	:	deletes the reference edge e from *fig*

The analysis always starts with an empty function interface graph. It then generates nodes and edges for the different statements. The functions *new_value_node*, *new_reference_node*, *new_link* and *new_reference_edge* are used to add new nodes and edges to the function interface graph. The edges (links and reference edges) are not connected to the nodes when they are newly generated, and their source and destination nodes as well as their annotations have to be specified by statements like '*edge.src = node*'. If a node or edge has to be removed from the function interface graph this can be done by using the functions *delete_value_node*, *delete_reference_node*, *delete_link* and *delete_reference_edge*.

Addition and multiplication operations within *OFF*

The first two functions that are defined explicitly here are used in the attribute computations of the rule handling the pointer dereference operations ('$*_u$'). They are the extended versions of the addition and multiplication operation. Since the additional offset annotation (*add_off*) can have any value contained in the set *OFF*, the two functions have to deal with the value representing unknown offsets (*'?'*) as well. They are defined as follows:

Definition 47 (addition and multiplication within *OFF*)

$$+ \;:\; OFF \times OFF \nrightarrow OFF \qquad\qquad \cdot \;:\; OFF \times OFF \nrightarrow OFF$$

$$a + b \;=\; \begin{cases} a+b & \text{if } a,b \in \mathbb{Z} \\ \text{'?'} & \text{otherwise} \end{cases} \qquad\qquad a \cdot b \;=\; \begin{cases} a \cdot b & \text{if } a,b \in \mathbb{Z} \\ \text{'?'} & \text{otherwise} \end{cases}$$

Value nodes belonging to a variable

Another function that has been used by the grammar rules is the *var_node* function. This function is shown in figure 3.33. It takes a SSA variable and a function interface graph as its input and returns the value node belonging to the variables address, if such a node exists

[35]Note that array types are treated as pointers, and hence the *is_pointer* function returns true for arrays as well.

```
(1) function $var\_node(var, fig)$
(2) {
(3)    if $(\nexists(vn \in fig.VN) : (var \in vn.vars))$ {
(4)       $vn = new\_value\_node(fig)$;
(5)       $vn.vars = \{var\}$;
(6)    }
(7)    return $vn$;
(8) }
```

Figure 3.33: Function returning the value node representing the address of a variable

(7). In case no such node exists a new node will have to be created first, and marked as the node belonging to the address of the given variable afterwards (4,5). After that this node will be returned.

The *unite* function

In some cases it becomes necessary to unite two or more nodes, which is done by the *unite* function shown in figure 3.34. In case the nodes to be united are value nodes, the corresponding reference nodes have to be united as well (4,5). This ensures that there is always only one reference node belonging to a value node in any function interface graph. In either case a new node will be generated that will be used as the result of the union function (6,8). In the case of value nodes the newly generated node will represent all the variables addresses that have formerly been represented by one of the nodes that have to be united (7). In any case, all edges connected to one of the nodes to be united have to be updated and connected with the new node (9-11). Afterwards, the now superfluous nodes will be removed from the function interface graph (13-16). Since it is possible that due to the union of the nodes there are now edges having exactly the same annotations, the duplicate edges have to be removed (18-23). Finally, the newly generated united node will be returned (25).

The *get_ref_node* function

The *get_ref_node* function, which is shown in figure 3.35, is used when it becomes necessary to access the reference node that is connected to a certain value node (e.g. if a new reference edge has to be added). If a link already exists from the given value node (*src_node*) to a reference node, this node is returned (4). Otherwise, a new reference node as well as a new link have to be generated and connected to the value node (6-9). Afterwards either the existing reference node found or the newly generated reference node can be returned.

The *realloc_val_node* function

This really simple function is used to represent the effects of a `realloc` function call. Its only parameter is the value node representing the previously allocated memory. First a new value node representing the newly allocated memory is generated (3). Then, if there is a value node representing the previously allocated memory (4), the newly generated value node is connected to the reference node belonging to the other value node (5-7). Finally, the newly generated value node representing the memory returned by the `realloc` function is returned. All this has to be done to make sure that the value nodes representing other locations which can be accessed from the previously allocated memory, can later be accessed from the value node representing the newly allocated memory as well. Exactly this can be achieved by letting the two value nodes share the same reference node.

```
function $unite(node\_set)$
{
  if $(node\_set \subseteq fig.VN)$ {
    $ref\_node\_set = \{n \in fig.RN \mid \exists(e \in fig.VRE) : (e.src \in node\_set) \wedge (e.dst = n)\}$;
    $unite(ref\_node\_set)$;
    $unite\_node = new\_value\_node(fig)$;
    $unite\_node.vars = \cup_{n \in node\_set} n.vars$;
  } else if $(node\_set \subseteq fig.RN)$ $unite\_node = new\_reference\_node(fig)$;
  for all $(e \in fig.E)$ {
    if $(e.src \in node\_set)$ $e.src = unite\_node$;
    if $(e.dst \in node\_set)$ $e.dst = unite\_node$;
  }
  if $(node\_set \subseteq fig.VN)$ {
    for all $(n \in node\_set)$ $delete\_value\_node(fig, n)$;
  } else if $(node\_set \subseteq fig.RN)$ {
    for all $(n \in node\_set)$ $delete\_reference\_node(fig, n)$;
  }
  while $(\exists(e, e' \in fig.VRE) : (e.src = e'.src) \wedge (e.dst = e'.dst))$ {
    $delete\_link(fig, e')$;
  }
  while $(\exists(e, e' \in fig.RVE) : (e.src = e'.src) \wedge (e.dst = e'.dst) \wedge (e.derop = e'.derop) \wedge$
                        $(e.off = e'.off) \wedge (e.gen = e'.gen))$ {
    $delete\_reference\_edge(fig, e')$;
  }
  return $unite\_node$;
}
```

Figure 3.34: Function used to unite the nodes of a function interface graph

```
function $get\_ref\_node(src\_node)$
{
  if $(\exists(e \in fig.E) : e.src = src\_node)$ {
    $rn = e.dst$;
  } else {
    $rn = new\_reference\_node(fig)$;
    $l = new\_link(fig)$;
    $l.src = src\_node$;
    $l.dst = rn$;
  }
  return $rn$;
}
```

Figure 3.35: Function returning the reference node belonging to a value node

```
(1)  function $realloc\_val\_node(old\_node)$
(2)  {
(3)     $new\_node = new\_value\_node(fig)$;
(4)     if $(old\_node \neq nil)$
(5)        $e = new\_link(fig)$;
(6)        $e.src = new\_node$;
(7)        $e.dst = get\_ref\_node(old\_node)$;
(8)     }
(9)     return $new\_node$;
(10) }
```

Figure 3.36: Function returning the value node for `realloc` function calls

```
(1)  function $assign(src\_node, derop, off, dst\_node)$
(2)  {
(3)     if $((src\_node \neq nil)$ and $(dst\_node \neq nil))$ {
(4)        $rn = get\_ref\_node(src\_node)$;
(5)        if $(\nexists(e \in fig.RVE) : (e.src = rn) \wedge (e.dst = dst\_node) \wedge (e.derop = derop) \wedge$
(6)                              $(e.off = off) \wedge (e.gen = 'a'))$ {
(7)           $e = new\_reference\_edge(fig)$;
(8)           $e.src = rn$;
(9)           $e.dst = dst\_node$;
(10)          $e.derop = derop$;
(11)          $e.off = off$;
(12)          $e.gen = 'a'$;
(13)       }
(14)    }
(15) }
```

Figure 3.37: Function computing the effects of assignments

The *assign* function

The *assign* function shown in figure 3.37 adds a new reference edge to the function interface graph to represent the effects of an assignment statement. However, this is done only if the values on either side of the assignment have corresponding value nodes (3). Since the reference edge that is to be generated has to be connected to the reference node belonging to the given value node (*src_node*), this reference node has to be generated if it does not exist yet. Otherwise the already existing reference node can be used (4). To represent the effect of the assignment this reference node has to be connected with the value node which represents the right hand side of the assignment (*dst_node*). Furthermore, the values of the *derop* and *off* annotations of the connecting edge have to correspond to the values passed to the function. Besides which, this edge has to be marked as an assigned edge, since it was created to represent the effects of an assignment. If such an edge already exists, the function interface graph is not changed. Otherwise a new edge is generated, connected to the nodes and the edge's annotations are set to the corresponding values (5-12).

The *deref* function

The *deref* function (figure 3.39) is used when a dereference operation has to be computed. Dereference operations do not modify any value (unlike assignments), they are only used

```
int *i;
char *c;
*((int **) c) = i + 3;
*((int **) (c + 1)) = i + 3;
... = *((int **) c);
```

Figure 3.38: Assignments with non-equal intersecting access ranges

to read values. Hence, a dereference operation never causes an assigned edge to be created. Instead, induced edges are generated to make it clear that the corresponding value has been read and not modified. Nevertheless dereference operations can affect assigned edges as well since the access ranges of induced and assigned edges are adapted under certain circumstances.

At the beginning of the *deref* function, the reference node belonging to the given value node (*src_node*) has to be generated if it does not exist yet (4). The range of values that may be accessed by the new dereference operation (*derop*) is computed and compared with the ranges of the outgoing edges of the reference node. If they intersect this means that there may be a value that can be accessed by either dereference operation (9-10), and hence the corresponding value nodes have to be united (11,21). Besides uniting the value nodes, the edges connecting the reference node to one of the nodes to be united, will be united as well. Therefore all these edges are deleted (18), and the smallest possible access range including all the access ranges of the deleted edges is computed (12,13). Since induced and assigned edges are never united this has to be done for either kind of edge separately. Further the combined offset of the assigned edges has to be computed. The combined offset becomes unknown (*'?'*) if two different offsets have to be matched, if the access ranges of the corresponding edges are not equal or if the access range of one of the corresponding edges is unknown (14-17). The second is necessary since the access ranges may overlap and therefore affect each other as in the example shown in figure 3.38. Here, the second assignment partially overwrites the pointer value stored by the first assignment, and so this pointer may point to absolutely any storage location. Therefore it cannot be assumed that `*((int **) c)` will point to `i + 3` when the third assignment is processed, which is reflected by the use of the unknown offset.

After all reference edges have been processed and deleted, the value nodes that have been connected to these edges can be united (21). In case there have not been any edges, and respectively no value nodes connected to them, a new value node has to be generated (22). Now the new reference edges can be generated and connected to the reference node. If there are assigned reference edges connected to one of the united nodes (23) a new assigned reference edge is generated. This edge receives a dereference operation corresponding to the united access range, and also receives the combined offset that has been calculated previously (24-29). In any case an induced reference edge is generated and equipped with the corresponding annotations (31-36). This ensures that an induced edge exists in any case after the function has been executed, even if there has not been an induced edge matching the given dereference operation before.

Fact 48

The offsets of induced reference edges are always zero since offset annotations only make sense for edges generated to represent the effects of assignment statements.

Now the processing of the dereference operation is completed, and the target value node can be returned (38).

The *get_off* function

The *get_off* function shown in figure 3.40 is closely related to the *deref* function. After the *deref* function has united the nodes and edges matching the given dereference operation there are at most two resulting matching edges left (one assigned and one induced edge). If there is an assigned edge, the offset of this edge is returned (6). This can be safely done since the analysed code fragment contains only a sequence of assignments and no control flow. Therefore a value once assigned will stay valid as long as it is not re-assigned by another statement of the sequence. If a value is modified the second time and used afterwards, the combined offset value has already been computed by the *deref* function. In case the value has not been assigned before there exists only an induced edge having zero as its offset, and hence the returned value is zero in this case (7).

The *bin_join* function

The *bin_join* function is used to let two value nodes (vn_1 and vn_2) share the same reference node. This has to be done if a binary operator is processed, since one does not know which (if any) of the operators holds the 'pointer' value[36]. In cases where the resulting value is dereferenced later, the corresponding reference edge is reachable through the value nodes belonging to both operands of the binary operation. This ensures that both of the operands are now treated as if they were the pointer value.

The first thing that has to be checked is if the operands of the binary operator are constant values. The only really interesting case is if both operands are non-constant values (3). In case there is no link connecting the corresponding value nodes with a reference node, those links have to be generated (7-9, 14-16). Otherwise the existing reference nodes can be used and modified later if necessary (5,12). Should none of the value nodes be connected to a reference node, a new reference node has to be generated and connected to both value nodes (18-21). If there is only one value node which has a corresponding reference node, the other value node is simply connected to the already existing reference node (22-25). Otherwise, if both value nodes have a link to a reference node, these reference nodes have to be united (26). In either case the value node belonging to the sub-expression is determined to be the value node belonging to the first operand (27). The second operand could have been chosen as well as the first one since both nodes are now sharing the same reference node. If only one of the operands is a non-constant value there is not much to be done. In this case the value node belonging to the sub-expression is the same as the value node belonging to the non-constant value (28,29). Finally, in cases where both operands are constants ($vn_1 = vn_2 = nil$) there are no nodes that could be accessed, and so there exists no value node belonging to this sub-expression (30).

3.2.3.2 Order of attribute computations

The order in which the grammar's computations are to be carried out is not specified by the grammar itself. The grammar's dependencies are not very complex, and it is quite easy to prove that this grammar is l-attributed[37]. This leads to the fact that the computations can be carried out within a single depth-first-left-to-right pass over the syntax-tree.

[36]As already mentioned before, it is not possible to rely on the type of the operands.

[37]In fact the proof consists of a trivial inspection: all that has to be done is to check for the desired properties of l-attributed grammars for every single rule.

```
function $deref(src\_node, derop)$
{
  if $(src\_node \neq nil)$ {
    $rn = get\_ref\_node(src\_node);$
    $unite\_set = \emptyset;$
    $unite\_off = nil;$
    $ass\_range = \emptyset;$
    $ind\_range = \emptyset;$
    for all $(n \in fig.VN, e \in fig.E : (e.src = rn) \wedge (e.dst = n))$ {
      if $(acc\_rng(e.derop) \cap acc\_rng(derop) \neq \emptyset)$ {
        $unite\_set = unite\_set \cup \{n\};$
        if $(e.gen = 'a')$ $ass\_range = ass\_range \cup acc\_rng(e.derop);$
        else if $(e.gen = 'i')$ $ind\_range = ind\_range \cup acc\_rng(e.derop);$
        if $(unite\_off = nil)$ $unite\_off = e.off;$
        else if $(unite\_off \neq e.off)$ $unite\_off = '?';$
        else if $(acc\_rng(e.derop) \neq acc\_rng(derop))$ $unite\_off = '?';$
        else if $(acc\_rng(e.derop) = \mathbb{Z})$ $unite\_off = '?';$
        $delete\_reference\_edge(fig, e);$
      }
    }
    if $(|unite\_set| \geq 1)$ $un = unite(unite\_set);$
    else $un = new\_value\_node(fig);$
    if $(ass\_range \neq \emptyset)$ {
      $e = new\_reference\_edge(fig);$
      $e.src = rn;$
      $e.dst = un;$
      $e.derop = deref\_op(ass\_range \cup acc\_rng(derop));$
      $e.off = unite\_off;$
      $e.gen = 'a';$
    }
    $e = new\_reference\_edge(fig);$
    $e.src = rn;$
    $e.dst = un;$
    $e.derop = deref\_op(ind\_range \cup acc\_rng(derop));$
    $e.off = 0;$
    $e.gen = 'i';$
  } else $un = nil;$
  return $un;$
}
```

Figure 3.39: Function computing dereference operations

```
function $get\_off(src\_node, derop)$
{
   $rn = get\_ref\_node(src\_node)$;
   if $(\exists(e \in fig.RVE) : (e.src = rn) \wedge (e.gen = 'a') \wedge$
                    $(acc\_rng(derop) \cap acc\_rng(e.derop) \neq \emptyset))$ {
      $result = e.off$;
   } else $result = 0$;
   return $result$;
}
```

Figure 3.40: Function returning the offset annotation

```
function $bin\_join(vn_1, vn_2)$
{
   if $((vn_1 \neq nil)$ and $(vn_2 \neq nil))$ {
      if $(\exists(e \in fig.E) : e.src = vn_1)$ {
         $rn_1 = e.dst$;
      } else {
         $l_1 = new\_link(fig)$;
         $l_1.src = vn_1$;
         $rn_1 = nil$;
      }
      if $(\exists(e \in fig.E) : e.src = vn_2)$ {
         $rn_2 = e.dst$;
      } else {
         $l_2 = new\_link(fig)$;
         $l_2.src = vn_2$;
         $rn_2 = nil$;
      }
      if $((rn_1 = nil)$ and $(rn_2 = nil))$ {
         $rn = new\_reference\_node(fig)$;
         $l_1.dst = rn$;
         $l_2.dst = rn$;
      } else if $(rn_1 = nil)$ {
         $l_1.dst = rn_2$;
      } else if $(rn_2 = nil)$ {
         $l_2.dst = rn_1$;
      } else $unite(\{rn_1, rn_2\})$;
      $result = vn_1$;
   } else if $(vn_1 \neq nil)$ $result = vn_1$;
   else if $(vn_2 \neq nil)$ $result = vn_2$;
   else $result = nil$;
   return $result$;
}
```

Figure 3.41: Function joining the reference nodes of the arguments of binary operators

```
(1) function depth_first_left_to_right(node)
(2) {
(3)    compute(node.$AC_{inh}$);
(4)    for ($i = 1$ to node.number_sons) {
(5)       depth_first_left_to_right(node.son[$i$]);
(6)    }
(7)    compute(node.$AC_{syn}$);
(8)    compute(node.$PC$);
(9) }
```

Figure 3.42: Function specifying the order of the grammar's computations

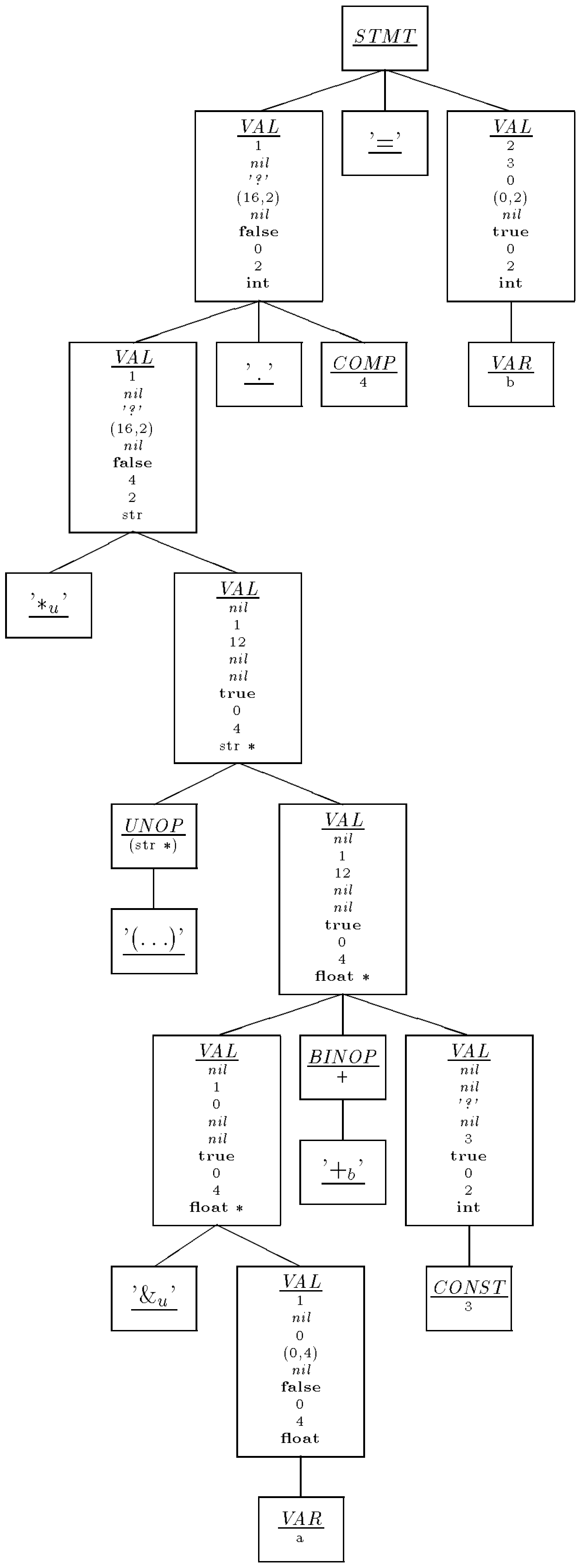
The function shown in figure 3.42 specifies this order exactly. First the inherited attributes of a node (AC_{inh}) are computed (3), then the node's children (if any) are visited (5). After that, the synthesised attributes (AC_{syn}) are computed (7), and finally the remaining plain computation (PC) of the node can be carried out (8).

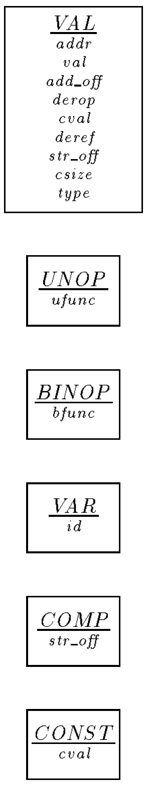
3.2.3.3 An example

The example shown in figure 3.43 demonstrates how the algorithm works, and how the attributes are used. Again it is assumed that the size of an integer is two bytes, whereas the size of float and pointer values is four bytes. Besides that, it is assumed that the size of the used structure (*str*) is six bytes, which means that no unused bytes are added due to alignment rules. Of course code like the one contained in this example is not very likely to appear in real programs. Nevertheless this example has been chosen since it demonstrates some of the most interesting operations of the attributed grammar quite well. Moreover, it can be seen that although the C programming language allows a lot of very complicated, and sometimes even senseless operations they are all properly handled by the grammar rules without any restrictions to the language.

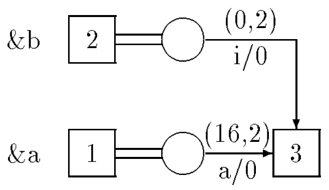
3.2.3.4 A more precise algorithm

The major disadvantage of the algorithm that has been described is the loss of precision caused by the fact that every variable has to be treated like a pointer variable. Although this cannot be avoided in general, it is possible to significantly improve the algorithm by taking the type information into account. If the user ensures that no aliases are introduced by the use of non-pointer values one can safely ignore all variables which have ordinal type (character, integer, float) as well as structures, unions and arrays which do not contain pointer values. E.g. programs that do not contain type casts from pointer to non-pointer values or casts from non-pointer to pointer values fulfil this condition. However, even programs which contain both types of cast operations (pointer to non-pointer and non-pointer to pointer) can fulfil this condition. Note that casts between pointer and non-pointer values can implicitly occur if unions containing pointer and non-pointer values are used in a certain way (if a pointer value is written first and a non-pointer value is read afterwards and vice versa). Besides which, it is possible that cast operations convert a pointer to a structure (or union) into a pointer to another structure (or union), and that one of the structures (or unions) contains a pointer value and the other one a non-pointer value at the same offset position. If the structure (or union) pointers are used in a way so that first a pointer value is written using one of the structure (or union) pointers and a non-pointer is


STMT
VAL 1 nil '?' (16,2) nil false 0 2 int
'='
VAL 2 3 0 (0,2) nil true 0 2 int
VAL 1 nil '?' (16,2) nil false 4 2 str
'.'
COMP 4
VAR b
'*u'
VAL nil 1 12 nil nil true 0 4 str *
UNOP (str *)
'(...)'
VAL nil 1 12 nil nil true 0 4 float *
VAL nil 1 0 nil nil true 0 4 float *
BINOP +
'+b'
VAL nil nil '?' nil 3 true 0 2 int
'&u'
VAL 1 nil 0 (0,4) nil false 0 4 float
CONST 3
VAR a



Node types and their attributes



Function interface graph

```
typedef struct {int x,y,z;} str;
float a;
int b;

(*((str*) ((&a) + 3))).z = b;
```

Source code

Figure 3.43: Attributed parse tree

read from the same storage location using the other structure (or union) pointer afterwards (or vice versa), a situation similar to the one described above arises. In either case an implicit cast operation has been carried out. This makes it almost impossible to statically detect if a program contains cast operations between pointer and non-pointer values as long as it contains unions or casts between structure or union pointers. Since the implicit cast operations described above are commonly used in C programs, only the most simple programs can be analysed with the improved algorithm without user interaction. However, if the user guarantees that the program has the desired properties he can still take advantage of the more precise results computed by the alternative algorithm.

The above-mentioned more precise results can be achieved by slightly changing the attribute grammar computations. To do this only two of the rules have to be altered. The first thing to be done is to make sure that there will be no value nodes generated for ordinal values. Additionally, the binary operators do not have to unite the nodes belonging to their operands since the 'pointer' operand can now be identified by its type.

The modified attribute computations that are necessary to make the algorithm more precise are listed in the following. Note that only those attribute computations that have to be altered will be listed, whereas all other attribute computations can remain unchanged.

```
AC(VAL ⟶ VAR) = {
    VAL.val  =  if (VAL.deref and (not is_ordinal(VAL.type)))
                  deref(VAL.addr, (VAL.str_off, VAL.csize))
                else nil
}                                                              III
```

```
AC(VAL₁ ⟶ VAL₂ BINOP VAL₃) = {
    VAL₁.val  =  if (is_pointer(VAL₂.type) and (not is_pointer(VAL₃.type))) VAL₂.val
                 else if (is_pointer(VAL₃.type) and (not is_pointer(VAL₂.type))) VAL₃.val
                 else nil
}                                                              X
```

3.2.3.5 Handling ϕ-functions

As stated before, the effects of the SSA-form have not yet been taken into consideration in the preceding part of this section, so this is the next thing to do. In fact, programs that have been transformed into SSA-form do not need many additional rules when the function interface graphs have to be generated. Note that variables having different SSA-values have been treated as different variables by the previously described algorithms. The only difference between analysing 'normal' code and code that has been transformed into SSA-form is that the effects of the ϕ-functions have to be taken into account as well. The *phi_func_fig* function shown in figure 3.44 handles the ϕ-functions that may occur at a CFG node. It has to be called for every ϕ-function belonging to a node before the true generation process of the function interface graph starts. This results in a small initial function interface graph that is then completed by the previously described algorithm.

ϕ-functions occur only at CFG nodes with more than one incoming edge. They are always the first statements belonging to these CFG nodes. A ϕ-function of the form $a_x = \phi(a_{y_1}, \ldots, a_{y_n})$ reflects the fact that different values (a_{y_1} to a_{y_n}) have been assigned to a on different incoming control paths, and that a_x may be any of these values. All that has to be done in this case is to make sure that the variables a_x and a_{y_1} to a_{y_n} share the same reference node when the following code will be analysed. Since the function interface graph for the analysed CFG node is empty at the beginning, the value nodes belonging

```
(1) function $phi\_func\_fig(a_x = \phi(a_{y_1}, \ldots, a_{y_n}))$
(2) {
(3)    $node\_set = \emptyset$;
(4)    for all $(v \in \{a_x, a_{y_1}, \ldots, a_{y_n}\})$ {
(5)       $vn = var\_node(v, fig)$;
(6)       $node\_set = node\_set \cup get\_ref\_node(vn)$;
(7)    }
(8)    $unite(node\_set)$;
(9) }
```

Figure 3.44: Function processing the ϕ-functions belonging to a CFG node

to the variables, as well as the shared reference node and the corresponding links have to be generated from scratch. To do this the function collects (and if necessary creates) the reference nodes corresponding to the variables occurring in the ϕ-function (4-6) and unites them afterwards (8).

3.2.3.6 CFG nodes representing function calls

Although the effects of function calls are handled by the interprocedural analysis parts, there are a few things that are done for function calls during the intraprocedural analysis part as well. The only thing that has to be done is to add a few value nodes representing the functions parameters and the value to which the functions result is assigned to the otherwise empty function interface graph. In case the function is called using a pointer to a function, a further node has to be generated for the variable holding the function pointer. The function processing the function calls is shown in figure 3.45. It takes the function interface graph belonging to the function call, the variable to which the functions result is assigned, the variable holding the function pointer and a set containing the functions parameters as its input. This information is then used to produce the corresponding function interface graph. In case the function does not return a value, or the returned value is ignored and hence not assigned to a variable, this is indicated by passing the value *nil* instead of the corresponding variable to the function (parameter *resvar*). The same applies to function calls which do not use function pointers and the corresponding parameter (*callvar*).

If the functions return value is assigned to a variable, this has to be handled like any other assignment besides the fact that there is no value node available for the value returned by the function. Hence a new value node will be generated and connected to the value node representing the address of the variable to which the functions result is assigned (3-15). After this has been done, it is time to deal with the functions parameters as well. This is performed by simply dereferencing the value nodes which represent the addresses of the functions parameters (17-18). This indicates that the corresponding values have been used. Although these newly generated nodes might be of no further use during the intraprocedural analysis they are not at all worthless. They will be used by the interprocedural analysis as a matching interface and are hence in any case necessary. In case of function calls using function pointers the variable holding the function pointer has to be handled like the parameters (20-21). This ensures that the possible aliases of this value can be computed later to find out which functions can be called by the function call. As an example, figure 3.46 shows two function calls and the corresponding function interface graphs.

```
**function** $func\_call\_fig(fig, resvar, callvar, params)$
{
   **if** $(resvar \neq nil)$ {
      $vn = var\_node(resvar, fig);$
      $l = new\_link(fig);$
      $rn = new\_reference\_node(fig);$
      $re = new\_reference\_edge(fig);$
      $vn' = new\_value\_node(fig);$
      $l.src = vn;$
      $l.dst = rn;$
      $re.src = rn;$
      $re.dst = vn'$
      $re.deref = (0, \textbf{sizeof}(resvar));$
      $re.off = 0;$
      $re.gen = 'a';$
   }
   **for all** $(v \in params)$ {
      $deref(var\_node(v, fig), (0, \textbf{sizeof}(v)));$
   }
   **if** $(callvar \neq nil)$ {
      $deref(var\_node(callvar, fig), (0, \textbf{sizeof}(callvar)));$
   }
}
```

Figure 3.45: Function processing function calls

```
int f(int *x, int y);                void (*f)(char x, int y);
int r;

r = f(x,y);                          (*f)(x,y);
```

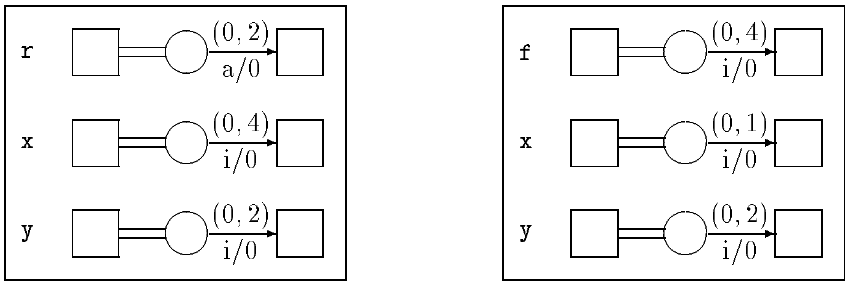


Figure 3.46: Two function calls and their corresponding function interface graphs

```
(1)  function $cfg\_fig\_match(fig_{pre}, fig_{post})$
(2)  {
(3)    $fig_{res} = new\_fig()$;
(4)    $unite\_nodes = \{(vn_{pre}, vn_{post}) \mid (vn_{pre} \in fig_{pre}.VN) \wedge (vn_{post} \in fig_{post}.VN) \wedge$
(5)      $((vn_{pre}.vars \cap vn_{post}.vars \neq \emptyset) \vee$
(6)        $(\exists (v_c, v'_c \in VAR_{SSA}) \mid (v_c \in vn_{pre}.vars) \wedge (v'_c \in vn_{post}.vars) \wedge$
(7)          $(base\_var(v_c) = base\_var(v'_c)) \wedge ind\_mod\_var(v_c) \wedge ind\_mod\_var(v'_c)))\}$;
(8)    $gen\_nodes\_and\_links(fig_{pre}, fig_{post}, fig_{res}, unite\_nodes)$;
(9)    $gen\_ref\_edges(fig_{pre}, fig_{post}, fig_{res})$;
(10)   return $fig_{res}$;
(11) }
```

Figure 3.47: Function matching the function interface graphs of two CFG nodes

3.2.4 Merging the function interface graphs of different CFG nodes

Now, after the code sequences of all CFG nodes have been processed the resulting function interface graphs have to be merged. This is done by processing the CFG edges one by one and combining the function interface graphs of the CFG nodes connected to these edges. After all these edges have been processed, we receive the function interface graph belonging to the analysed function. This graph does not take the effects of called functions into account yet, but it already contains all necessary intraprocedural information.

The matching of the function interface graphs can be done by using a simple worklist algorithm processing all CFG edges. For each edge there are two (maybe empty) function interface graphs that have to be matched. The algorithm shown in figure 3.47 takes the function interface graphs belonging to the two nodes connected by a CFG edge, and produces a new graph (fig_{res}) by combining the two input graphs. Since CFG edges are directed edges and their direction is important for the matching process as well, it is necessary to distinguish between the two nodes connected by a CFG edge. The function interface graph belonging to the source node of the CFG edge is fig_{pre}, whereas the one belonging to the destination node is fig_{post}. The major difference between these nodes is that the assigned edges of the first node have to be matched with the edges of the second node, whereas the assigned edges of the second node are not matched at all. This is obvious since the code belonging to the second node is executed after the code of the first node. Hence assignments made in the code belonging to the first node can influence the code belonging to the second node, but not vice versa.

The matching process for the two function interface graphs starts by producing a new empty function interface graph (3). The nodes that have to be matched are those nodes representing the same variable in the two function interface graphs fig_{pre} and fig_{post} as well as all nodes that can be reached from these nodes using matching access paths. Therefore the initial set of node pairs to be matched is initialised with the node pairs representing the same variable in the two function interface graphs (4-5). However, there are a few more nodes that have to be matched. If the address of a variable has been taken, this variable can be modified indirectly by following statements without accessing the variable itself. Therefore the nodes representing the corresponding SSA variables have to be merged as well if they can be modified indirectly (6-7). These nodes are computed with the help of the function *ind_mod_var* which checks whether the SSA value of a variable is valid since

```
(1) function $equalize(n_1, n_2)$
(2) {
(3)     if $(root(n_1) \neq root(n_2))$ {
(4)         if $(n_1.eq = nil)$ $n_1.eq = root(n_2)$;
(5)         else if $(n_2.eq = nil)$ $n_2.eq = root(n_1)$;
(6)         else $root(n_1).eq = root(n_2)$;
(7)     }
(8) }
```

Figure 3.48: Function uniting two trees based on the *eq*-relation

the address of the corresponding variable in the original program has been taken[38].

Now that the initial nodes to be merged have been computed, the value and reference nodes as well as the corresponding links of the new graph are generated (8), and finally the missing reference edges are generated and connected to the newly generated value and reference nodes (9).

When the value nodes of the resulting function interface graph have to be generated, all assigned and induced edges of the first graph (fig_{pre}) have to be matched with the induced edges of the second graph (fig_{post}). Therefore all nodes of the first graph reachable by a certain path have to be united with those nodes of the second graph reachable by a matching path using only induced edges. To compute these nodes the algorithm walks through the two graphs step by step collecting all nodes that have to be united.

The computation of the nodes is done by the functions *gen_nodes_and_links* and *compute_representatives* which are shown in figures 3.50 and 3.51. The function *gen_nodes_and_links* uses the function *compute_representatives* to compute the sets of nodes that have to be matched before the new nodes will be generated.

Computing the set of nodes to be matched

When the function *compute_representatives* is called the two function interface graphs that have to be matched, as well as the set containing pairs of value nodes representing the same variable within the different function interface graphs, are passed to the function. Based on these initial nodes to be matched the algorithm starts to compute representatives for all nodes. Later those nodes with the same representative are united.

To handle this problem efficiently, a version of the union search algorithm described in [Sed92] is used. This algorithm uses a reference between two nodes to identify that they will later be represented by the same node (*eq*). These references together with the nodes produce a forest, where every tree contains all nodes that are later represented by the same node. Every time the algorithm identifies two nodes that will later be represented by the same node, the function *equalize* shown in figure 3.48 is called. This function then unites the two trees if they have not been united yet (3). In case one of these nodes has not been united with another node before, this nodes reference is still unused, and hence this reference can simply be set to refer to the root of the other nodes tree (4,5). Otherwise, the root of one tree can be connected to the root of the other tree (6).

The function that is used to find the root of a tree is shown in figure 3.49. To increase the efficiency of this function, the found root node will be stored after it has been computed

[38]Note that if the address of a SSA variable `vn` is taken somewhere in the program this may lead to the fact that the variable `vm` (which is valid after `vn`) may be modified indirectly by following statements even if the address of `vm` itself has not been taken.

```
(1) function root(n)
(2) {
(3)    if (n.eq ≠ nil) {
(4)       r = root(n.eq);
(5)       n.eq = r;
(6)    } else r = n;
(7)    return r;
(8) }
```

Figure 3.49: Function returning the root of the tree based on the *eq*-relation

once (5). This means that the tree is being rebuilt permanently and that it is not necessary to traverse several nodes on the way to the root more than once. Altogether this leads to a very efficient method to unite two trees and to compute a trees root.

Before the two functions given above can be used, the *eq* references have to be initialised first. This is done by the *compute_representatives* function (figure 3.50) that clears all these references at the beginning (3,4). The algorithm computing the representatives uses a set which keeps the pairs of nodes that have to be matched and a boolean value (*no_off*). This boolean value indicates if the two nodes have been reached by edges with different or unknown additional offset values. In case this value is **true** the dereference operations of the outgoing edges of this node have to get unknown offsets because these offsets depend on the additional offsets of the edges leading to the node that are unknown now. To mark such nodes the flag *no_off* is used. All these flags are set to **false** at the beginning of the algorithm (5). After this has been done, the set containing the nodes to be merged, which has been passed to the function as a parameter (*unite_nodes*), is used to initialise the set of unprocessed nodes. Since the elements of this set do not only contain the two nodes but a flag (*no_off*) as well, this flag has to be initialised here (7). The flags value is always **false** since the two nodes represent addresses of variables and hence these values are constant and nothing could have been added to them.

Since the graphs may contain circles it is necessary to keep track of the nodes that have already been processed. This is done by the set *processed* which is empty when the algorithm starts (8). Unfortunately, the algorithm walks through the two graphs at the same time, and therefore it is not possible to stop whenever a previously visited node is reached, since it might be necessary to visit nodes more than once. Every pair of nodes can be processed at most twice. A node pair has to be revisited if the access ranges of the outgoing edges have been taken into consideration (*no_off* = **false**) when the node pair has been visited for the first time, and it was found later that the access ranges have to be ignored due to unknown or unequal additional offset values of the incoming edges (*no_off* = **true**). Therefore the pairs of nodes visited and the corresponding flags have to be stored. If a certain node pair has been visited once with the *no_off* flag set, it is not necessary to revisit this node, since there will be no further nodes found to be matched in this case. This is because taking the access ranges into consideration only prevents some nodes from being matched that would have been matched otherwise.

After all the initialisations have been done, the algorithm starts by processing the nodes contained in the *unprocessed* set one by one. The first thing to be done is to remove the corresponding triple from the set containing the unprocessed triples and to add it to the set of triples that have already been processed (10,11). As described above, the triple containing the same nodes but a *no_off* flag that is set to **false** can be added to the set of processed

```
function $compute\_representatives(fig_{pre}, fig_{post}, unite\_nodes)$
{
  for all $(n \in (fig_{pre}.N \cup fig_{post}.N))$ {
    $n.eq = nil$;
    $n.no\_off = \mathbf{false}$;
  }
  $unprocessed = \{(vn_{pre}, vn_{post}, \mathbf{false}) \mid (vn_{pre}, vn_{post}) \in unite\_nodes\}$;
  $processed = \emptyset$;
  while $(\exists((vn_{pre}, vn_{post}, no\_off) \in unprocessed))$ {
    $unprocessed = unprocessed - \{(vn_{pre}, vn_{post}, no\_off)\}$;
    $processed = processed \cup \{(vn_{pre}, vn_{post}, no\_off)\}$;
    if $(no\_off)$ {
      $processed = processed \cup \{(vn_{pre}, vn_{post}, \mathbf{false})\}$;
      $vn_{pre}.no\_off = \mathbf{true}$;
    }
    $equalize(vn_{pre}, vn_{post})$;
    if $(\exists(e \in fig_{pre}.E) : e.src = vn_{pre})$ $rn_{pre} = e.dst$;
    else $rn_{pre} = nil$;
    if $(\exists(e \in fig_{post}.E) : e.src = vn_{post})$ $rn_{post} = e.dst$;
    else $rn_{post} = nil$;
    if $((rn_{pre} \neq nil)$ and $(rn_{post} \neq nil))$ {
      $equalize(rn_{pre}, rn_{post})$;
    }
    $unprocessed = unprocessed \cup \{(vn'_{pre}, vn'_{post}, no\_off') \mid$
      $\exists(e_{pre} \in fig_{pre}.E, e_{post} \in fig_{post}.E) :$
        $(e_{pre}.src = rn_{pre}) \wedge (e_{pre}.dst = vn'_{pre}) \wedge$
        $(e_{post}.src = rn_{post}) \wedge (e_{post}.dst = vn'_{post}) \wedge$
        $((acc\_rng(e_{pre}.derop) \cap acc\_rng(e_{post}.derop) \neq \emptyset) \vee no\_off) \wedge$
        $(no\_off' = ((e_{pre}.off \neq e_{post}.off) \vee (e_{pre}.off = '?') \vee (e_{post}.off = '?'))) \wedge$
        $((vn'_{pre}, vn'_{post}, no\_off') \notin processed) \wedge (e_{post}.gen = 'i')\}$;
  }
}
```

Figure 3.50: Function computing representatives for the nodes that have to be united

triples as well in case the *no_off* flag has been set, because it does not make sense to revisit this node pair then (12,13). Additionally, one of the nodes is marked to have no outgoing edges with dereference operations having non-unknown offsets (14). Here it does not matter which node is marked since both nodes will later be united anyway. Finally, the trees of the nodes are joined to ensure that the nodes will be united later (16).

After the value nodes have been processed, the reference nodes belonging to the value nodes have to be computed (17-20). In case each of the value nodes has a corresponding reference node, these nodes have to be marked to be united later as well (21-22). Now, new triples are added to the set containing the unprocessed triples, if there are nodes reachable from the reference nodes by matching edges. As described above, the access ranges are only taken into account if the *no_off* flag is not set. The flag of the new triples (no_off') is set depending on the additional offsets of the edges reaching the nodes. Only those triples are added that have not yet been marked as being processed before (24-30). The computation terminates if there are no triples left to be processed. Since no triple can be inserted more than once it is clear that this algorithm cannot run into an endless loop, and hence it will terminate in any case.

Generating the nodes and links

Now, after the representatives have been computed, the new nodes and links are generated by the function *gen_nodes_and_links*. Before the new nodes are generated the *no_off* flags for these nodes are computed. This is done by going through the nodes and setting the flag of the representative if the flag is set for any of these nodes (4-5). This means that the flag of the representative is set whenever one of the flags of the represented nodes is set.

After this has been done, the generation of the new value nodes can begin. Since all nodes that are represented by the same node in the resulting graph are now components of the same tree, a new node will be generated for each of these trees. Besides that, the newly generated node becomes the new root of the tree (7-9). This allows the algorithm to compute the value node of the resulting graph representing one of the nodes of the input graphs by using the *root* function. Furthermore, the *no_off* flag of the former root of the tree is copied to the newly generated root node (10). In case one of the value nodes that is to be replaced by the newly generated value node has represented a variables address, the newly generated node will represent this variables address in future. Hence the variables annotated to any of the nodes to be replaced have to be annotated to the new node within the next step (13-15).

After all new value nodes have been generated, the new reference nodes are generated in much the same way (17-20). The only difference here is that all *no_off* flags can be initially set to **false** since the *no_off* flags of the reference nodes have not been set before (20).

As the last step, the connecting links still missing have to be generated for the resulting function interface graph. To do this, all links of the two input graphs are processed and in case the representatives of the links source and destination node are not yet connected by a corresponding link in the resulting function interface graph, such a link is generated (23-29). Finally, the *no_off* flag of the reference node has to be set in case the flag of the value node connected to the reference node is set (30). This means that the *no_off* flag of the reference node is set whenever at least one of the *no_off* flags of the corresponding values nodes is set.

Generating the reference edges

Now, after all nodes and links have been generated, the processing of the reference edges can

```
function $gen\_nodes\_and\_links(fig_{pre}, fig_{post}, fig_{res}, unite\_nodes)$
{
   $compute\_representatives(fig_{pre}, fig_{post}, unite\_nodes)$;
   for all $(vn \in (fig_{pre}.VN \cup fig_{post}.VN))$ {
      if $(vn.no\_off)$ $root(vn).no\_off =$ true;
   }
   for all $(vn \in (fig_{pre}.VN \cup fig_{post}.VN))$ {
      if $(vn.eq = nil)$ {
         $vn.eq = new\_value\_node(fig_{res})$;
         $vn.eq.no\_off = vn.no\_off$;
      }
   }
   for all $(vn \in (fig_{pre}.VN \cup fig_{post}.VN))$ {
      $rt = root(vn)$;
      $rt.vars = rt.vars \cup vn.vars$;
   }
   for all $(rn \in (fig_{pre}.RN \cup fig_{post}.RN))$ {
      if $(rn.eq = nil)$ {
         $rn.eq = new\_reference\_node(fig_{res})$;
         $rn.eq.no\_off =$ false;
      }
   }
   for all $(e \in (fig_{pre}.VRE \cup fig_{post}.VRE)$ {
      $vn = e.src$;
      $rn = e.dst$;
      if $(\nexists(e' \in fig_{res}.VRE) \ : \ (root(vn) = e'.src) \wedge (root(rn) = e'.dst))$ {
         $e' = new\_link(fig_{res})$;
         $e'.src = root(vn)$;
         $e'.dst = root(rn)$;
         $e'.dst.no\_off = e'.dst.no\_off$ or $e'.src.no\_off$;
      }
   }
}
```

Figure 3.51: Function generating the nodes and links for two function interface graphs that have to be matched

start, which is done by the *gen_ref_edges* function shown in figure 3.52. The set of edges that have to be processed is initialised and contains all reference edges of the input graphs (3). Although induced and assigned edges are never united they have to be matched here. This has to be done since values that are assigned in the first graph produce assigned edges that have to be matched with the edges that are produced when the previously assigned values are used in the second graph. When the processing of the edges starts, an arbitrary edge is chosen, and all matching edges having the same generation mode attribute are removed from the set of edges that have to be processed (15). This avoids multiple edges having the same annotations. Two edges can be matched if they have intersecting access ranges or if the access ranges are ignored due to the *no_off* flag (14), if their source and destination nodes have the same corresponding node in the resulting graph (9) and if their generation mode attributes fulfil the condition displayed in the lines 10 to 13. This condition ensures that two edges are not matched if one edge is an assigned edge of the second graph (fig_{post}) and the other one is an induced edge of the first graph. Every time when a set of edges that will be united has been removed from the set of edges to be processed, a new edge representing all the removed edges is added to the resulting graph (22-27). The dereference operation of the merged edge is computed by uniting the access ranges of the edges that are merged (6,16). However, if the *no_off* flag of the source node has been set the dereference operation will in any case receive an unknown offset (5). The additional offset of the resulting edge is kept only if all merged edges have the same additional offset and access range. Otherwise the additional offset is unknown (7,17,18).

Note, that the algorithm explained here can be sped up by avoiding to copy the function interface graphs and merging them directly. Especially, if there are comparatively few nodes that have to be merged the overhead caused by copying the graphs results in an unnecessary waste of time. However, the algorithm as it was described here is easier to understand because there are no side-effects that have to be taken into account. Such side-effects occur when changes to one of the graphs are made and the second graph is indirectly modified as well, since the graphs have already been partially connected during the merging process. Therefore it was left to the implementation to deal with such details, and the much simpler algorithm has been described here.

3.2.4.1 An example

The program shown in figure 3.53 serves as an example on how the merging process for function interface graphs is done. The program is shown in its initial form as well as in SSA form. The numbers at the end of the source code lines are the numbers of the corresponding CFG nodes. The CFG graph and the function interface graphs belonging to the CFG nodes are shown in figure 3.54, and the resulting function interface graph is shown in figure 3.55. As can be seen, the ϕ-functions belong to the *DO* node (3) since this node has two incoming edges with different reaching definitions. Since there is no further code belonging to this node the resulting function interface graph is comparably simple. The only thing that has been done is that shared reference nodes as well as links have been generated for the variables occurring in the ϕ-functions. The assignments occurring in the program belong to the two sequential nodes (2 and 4), and so the function interface graphs belonging to these nodes are used to represent the effects of these assignments. The other CFG nodes have empty function interface graphs since there is no code belonging to them.

What can be seen from the resulting function interface graph is that `ele` and `first` both point to a storage location holding a pointer back to the same storage location. Although

```
**function** $gen\_ref\_edges(fig_{pre}, fig_{post}, fig_{res})$
{
  $edges = fig_{pre}.RVE \cup fig_{post}.RVE$;
  **while** $(\exists(e \in edges))$ {
    **if** $(root(e.src).no\_off)$ $unite\_range = \mathbb{Z}$;
    **else** $unite\_range = acc\_rng(e.derop)$;
    $unite\_off = e.off$;
    **for all** $(e' \in fig_{pre}.RVE \cup fig_{post}.RVE)$ {
      **if** $((root(e.src) = root(e'.src))$ **and** $(root(e.dst) = root(e'.dst))$ **and**
          (**not** $((e.gen = 'a')$ **and** $(e \in fig_{post}.RVE)$ **and**
                $(e'.gen = 'i')$ **and** $(e' \in fig_{pre}.RVE)))$ **and**
          (**not** $((e'.gen = 'a')$ **and** $(e' \in fig_{post}.RVE)$ **and**
                $(e.gen = 'i')$ **and** $(e \in fig_{pre}.RVE)))$ **and**
          $((acc\_rng(e.derop) \cap acc\_rng(e'.derop) \neq \emptyset)$ **or** $root(e.src).no\_off)$ {
        **if** $(e.gen = e'.gen)$ $edges = edges - \{e'\}$;
        $unite\_range = unite\_range \cup acc\_rng(e'.derop)$;
        **if** $((acc\_rng(e.derop) \neq acc\_rng(e'.derop)) \vee (unite\_off \neq e'.off))$ {
          $unite\_off =$ '?';
        }
      }
    }
    $e'' = new\_reference\_edge(fig_{res})$;
    $e''.src = root(e.src)$;
    $e''.dst = root(e.dst)$;
    $e''.derop = deref\_op(unite\_range)$;
    $e''.off = unite\_off$;
    $e''.gen = e.gen$;
  }
}
```

Figure 3.52: Function generating the edges for two function interface graphs that have to be matched

```
typedef struct s {                typedef struct s {
  int e;                            int e;
  struct s *next;                   struct s *next;
} int_list;                       } int_list;
int_list *ele, *first;            int_list *ele_1, *ele_2, *ele_3, *first_0;
int sum;                          int sum_1, sum_2, sum_3;
:                                 :
ele = first;                      ele_1 = first_0;                        (2)
sum = 0;                          sum_1 = 0;                              (2)
do {                              do {
                                    ele_3 = φ(ele_1, ele_2);              (3)
                                    sum_3 = φ(sum_1, sum_2);              (3)
  sum = sum + (*ele).e;             sum_2 = sum_3 + (*ele_3).e;           (4)
  ele = (*ele).next;                ele_2 = (*ele_3).next;                (4)
} while (...);                    } while (...);
```

Figure 3.53: Example program

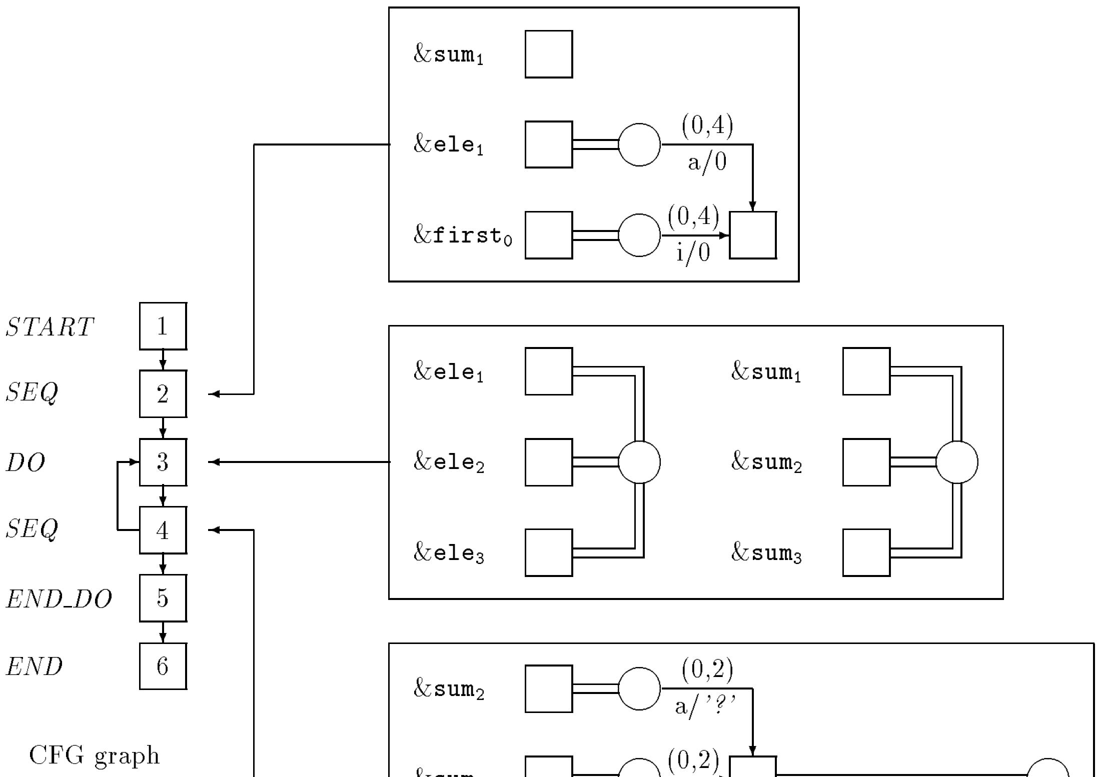


Figure 3.54: Function interface graphs of different CFG nodes

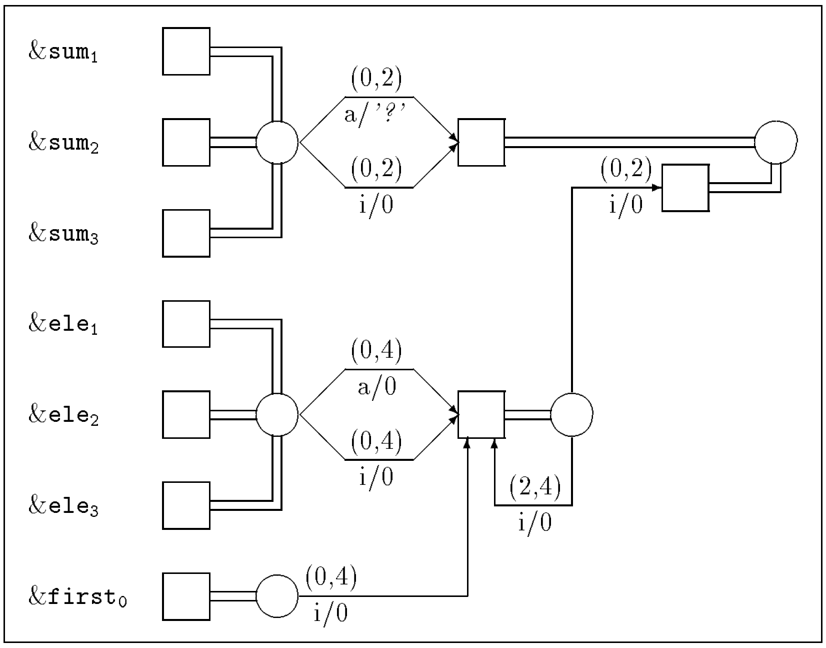


Figure 3.55: Merged function interface graph

this is not true in general, it is at least possible. Nevertheless, it might be that `first` is a pointer to an element of a non-cyclic list, and that there is no pointer back to the preceding elements. However, this is an acceptable source of imprecision since it is not possible to represent the effects of linked lists exactly, if a finite representation is chosen. Many of the other approaches dealing with alias analysis (e.g. [HPR89, CWZ90, LH88, LR92]) deal with this problem in a similar manner to keep the access paths or graphs as small as possible without losing too much precision.

3.2.5 Using the function interface graph to find may aliases

The function interface graph can be used to find out if two expressions might be aliased or not by computing the set of value nodes reachable by the access paths corresponding to the given expressions. These value nodes can be computed by the help of the two functions *acc_vals* and *acc_step* that are defined in the following.

Definition 49 (value nodes belonging to an access path)
Given a function interface graph *fig* and an access path $(v, do_1, \ldots, do_n) \in AP$ then the functions

$$\begin{aligned} acc\_vals &: FIG \times AP \nrightarrow \mathbb{P}(VN) \\ acc\_step &: FIG \times (VN \times OFF) \times DEROP^* \nrightarrow \mathbb{P}((VN \times OFF)) \end{aligned}$$

can be used to determine all nodes representing the values that might be accessed by expressions corresponding to the given access path. *acc_vals* is defined as

$$acc\_vals(fig, (v, do_1, \ldots, do_n)) = \{vn \in VN_{fig} \mid \exists (o \in OFF, vn_v \in VN_{fig}) : (v \in vars_{fig}(vn_v)) \wedge ((vn, o) \in acc\_step(fig, (vn_v, 0), do_1 \ldots do_n))\}$$

where vn_v is the value node belonging to the address of the variable v. The function *acc_step* that has been used by *acc_vals* is defined as follows.

$$acc\_step(fig, (vn, o), do_1 \ldots do_n) \;=\; \begin{cases} \{\ (vn'', o'') \in (VN \times OFF) \mid \exists(vn' \in VN_{fig},\ l \in VRE_{fig}, \\ \quad rn \in RN_{fig},\ e \in RVE_{fig},\ p \in \mathbb{Z}) \ : \\ \qquad (l.src = vn) \ \wedge \ (l.dst = rn) \ \wedge \\ \qquad (e.src = rn) \ \wedge \ (e.dst = vn') \ \wedge \\ \qquad (p \in acc\_rng(do_1)) \ \wedge \\ \qquad ((p + o) \in acc\_rng(e.derop)) \ \wedge \\ \qquad (vn'', o'') \in acc\_step(fig, (vn', e.off), do_2 \ldots do_n)\ \} & \text{if } (n > 0) \\ (vn, o) & \text{otherwise} \end{cases}$$

The major part of the computation of the value nodes reachable by a certain access path is done by the *acc_step* function, which searches step by step for a path through the function interface graph. This function takes a function interface graph (*fig*), a value node (vn) and the additional offset (o) that occurred on the path to this value node as its input. Besides these a sequence of dereference operations ($do_1 \ldots do_n$) that have to be applied are passed to the function as well. The function recursively computes a set of pairs containing value nodes and offset values ((vn'', o'')). The computed value nodes (vn'') are exactly those value nodes that can be reached by applying the given sequence of dereference operations ($do_1 \ldots do_n$) to the given value node (vn). The additional offset (o) that appeared on the path to this node is taken into consideration by the function as well. The computation is done as follows: as long as there are dereference operations left in the sequence ($n > 0$), the function computes the pairs of value nodes and used offsets ($vn', e.off$) for the first dereference operation of the sequence. It then recursively calls itself with these pairs and the remaining dereference operations ($do_2 \ldots do_n$). Otherwise, if the sequence of dereference operations is empty, there is nothing left to do but to return the found pair.

Now, there is not much left to do for the *acc_vals* function. All this function has to do is to pass the value node representing the address of the variable contained in the access path as well as the sequence of dereference operations to the *acc_step* function. After that the value nodes have to be extracted out of the pairs that will be returned by the *acc_step* function.

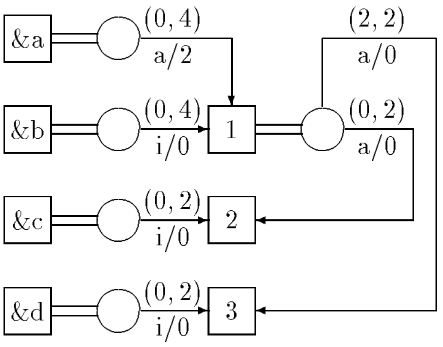
Now, after the value nodes related to a certain access path have been specified, it is quite easy to define the aliases represented by the function interface graph.

Definition 50 (aliases represented by the function interface graph)
Two expressions may be aliases (due to a function interface graph) only if two access paths exist corresponding to the expressions and a value node further exists contained in both sets of value nodes belonging to these access paths. In this case *fig_alias* returns **true**, otherwise **false** will be returned.

$$\begin{aligned} fig\_alias \ &: \ FIG \times AP \times AP \nLeftrightarrow \mathbb{B} \\ fig\_alias(fig, ap_1, ap_2) \ &= \ (acc\_vals(fig, ap_1) \cap acc\_vals(fig, ap_2) \neq \emptyset) \end{aligned}$$

To make things clearer, an example is shown in figure 3.56. This figure contains a small sequence of C code and the corresponding function interface graph that is created by the intraprocedural analysis. Looking at the program code it is clear that ∗`b` and `c` as well as ∗`a` and `d` are aliased after the code has been executed. This is reflected by the

```
void fkt()
{
  int *a, *b, c, d;
...
  b[0] = c;
  b[1] = d;
  a  =  b + 1;
}
```



exp	*acc_vals*(*fig*, ap_{exp})
*a	{ 3 }
*b	{ 2 }
c	{ 2 }
d	{ 3 }

fig_alias	*a	*b	c	d
*a	**true**	**false**	**false**	**true**
*b	**false**	**true**	**true**	**false**
c	**false**	**true**	**true**	**false**
d	**true**	**false**	**false**	**true**

Figure 3.56: Function interface graph and its represented aliases

function interface graph as well. The access paths corresponding to *a, *b, c and d are $(a,(0,4),(0,2))$, $(b,(0,4),(0,2))$, $(c,(0,2))$ and $(d,(0,2))$. Applying the *acc_vals* function to these access paths computes the set of value nodes belonging to them. The computed sets of value nodes as well as the result of the *fig_alias* function are shown in the corresponding tables contained in the figure[39].

3.2.6 Correctness

The alias analysis is precise in the sense that every possible alias is detected, whereas the opposite, that every found alias can really occur, does not hold in general. The aliases represented by the function interface graph are determined by the *fig_alias* function, which leads to the following claim to be proved.

Claim:

Given a function, its corresponding function interface graph *fig* and two base expressions[40] $exp, exp' \in EXP_{base}$ that may be aliased due to the functions effects, then they can be identified as possible aliases by the *fig_alias* function.

Proof:

First Step: Since aliases have only been defined for access paths and not for expressions we have to deal with the access paths corresponding to the expressions *exp* and *exp'* here. It is assumed that

$$ap = (v, do_1, \ldots, do_n) \quad \text{and} \quad ap' = (v', do'_1, \ldots, do'_{n'})$$

are the access paths corresponding to *exp* and *exp'*. According to the definition of aliases (Def. 40) there has to be at least one program state *st* so that the memory

[39]Although the functions take access paths instead of expressions as their input, the tables contain the corresponding expressions and not the access paths for reasons of simplicity.

[40]It is not necessary to deal with simple expressions here, since if there are two aliased simple expressions there always are two corresponding base expressions that are aliased as well.

location accessible by the access paths *ap* and *ap'* are equal if the expressions *exp* and *exp'* are aliased. When the program starts the memory locations accessible by different variables are different as well[41]. Only the execution of assignments occurring in the program can change this. To make the two access paths *ap* and *ap'* refer to the same location there has to be a sequence of assignment statements $ass_1, \ldots, ass_t$ which produce the alias. In the following it is assumed that each of these assignments has the form

$$ass_i \quad : \quad exp_i = exp'_i$$

and that $st_{\overline{ass_i}}$ and $st_{\underline{ass_i}}$ denote the program states before and after the assignment ass_i has been executed.

The assignment sequence which produces the alias does not necessarily have to appear in the program code in the same order. It is sufficient if the assignments are executed in the given order. In between the execution of the assignments there may be other non-assignment statements (e.g. loops or jumps) that are executed, but since these statements do not modify any program data they can be ignored here.

$$\forall(1 \le i < t) \quad : \quad mem(st_{\underline{ass_i}}) = mem(st_{\overline{ass_{i+1}}})$$

The definition of the *fig_alias* function (Def. 50) implies that

$$fig\_alias(fig, ap, ap')$$

becomes true only if a value node exists corresponding to both access paths.

Altogether this leads to the following to be shown: If the memory locations accessible by the access paths *ap* and *ap'* become the same due to the execution of the sequence of assignment statements $ass_1, \ldots, ass_t$, then at least one value node exists corresponding to both access paths.

$$\begin{array}{c} (loc(ap, st_{\overline{ass_1}}) \neq loc(ap', st_{\overline{ass_1}})) \ \wedge \ (loc(ap, st_{\underline{ass_t}}) = loc(ap', st_{\underline{ass_t}})) \\ \Rightarrow \\ \exists(vn \in VN_{fig}) \ : \ vn \in (acc\_vals(fig, ap) \cap acc\_vals(fig, ap')) \end{array} \tag{3.1}$$

Second Step: The above given supposition (3.1) will now be proved by induction.

<u>$t = 0$</u>: In this case there are no assignments that are executed and hence no new aliases can occur. This means that the program state will not change when the (empty) sequence of assignments is executed. Therefore the first part of the implication is always false, and hence the implication itself becomes true.

[41]This is only true assuming that initialising assignments have not been executed yet, and that no memory location is accessed directly by the program. The direct access to memory locations as in `a` = (**int** ∗)`100` followed by ∗`a` = ... is not supported by the ANSI standard anyway. However, even the effects of direct memory accesses could be handled correctly by introducing value nodes representing the memory locations accessible by the integer and floating point constants occurring in the program. Since this would worsen the precision of the analysis, it is not done, although it would only need a slight modification of the attribute grammar rule handling constants (rule IV).

$\underline{t \rightarrow t+1}$: The execution of the assignment statement ass_{t+1} modifies the memory state by assigning a new value to the location specified by exp_{t+1}. In case the value of exp_{t+1} is a pointer value or is cast into a pointer value, the locations that are accessible by using this value equal those that can be accessed from the location holding the value of exp'_{t+1}. According to the supposition, ap and ap' have not been aliased before the entire assignment sequence has been executed, but are aliased afterwards. This leads to two cases: in the first case the alias already exists before the $(t+1)$'th assignment will be executed,

$$loc(ap, st_{\overline{ass}_{t+1}}) = loc(ap', st_{\overline{ass}_{t+1}})$$

and in the second case it does not, but it is created by the assignment statement ass_{t+1}.

$$\begin{gathered}(loc(ap, st_{\overline{ass}_{t+1}}) \neq loc(ap', st_{\overline{ass}_{t+1}})) \ \wedge \\ (loc(ap, st_{\underline{ass}_{t+1}}) = loc(ap', st_{\underline{ass}_{t+1}}))\end{gathered}$$

The first case is rather trivial: since ap and ap' are already aliased after the assignment ass_t has been executed there has to exist a value node accessible by both access paths according to the supposition.

In the second case some further inspections are necessary. Since all differences between the memory states of $st_{\overline{ass}_{t+1}}$ and $st_{\underline{ass}_{t+1}}$ are caused by the assignment ass_{t+1} this means that the memory location reachable by ap and ap' after the assignment has been executed, has been reachable by the access path representing exp'_{t+1} before the assignment ass_{t+1} has been executed (3.2) and that the value that has been modified by the assignment is used by either ap or ap' (3.3). Without affecting the validity of the proof it is assumed that ap is the access path using the newly assigned value.

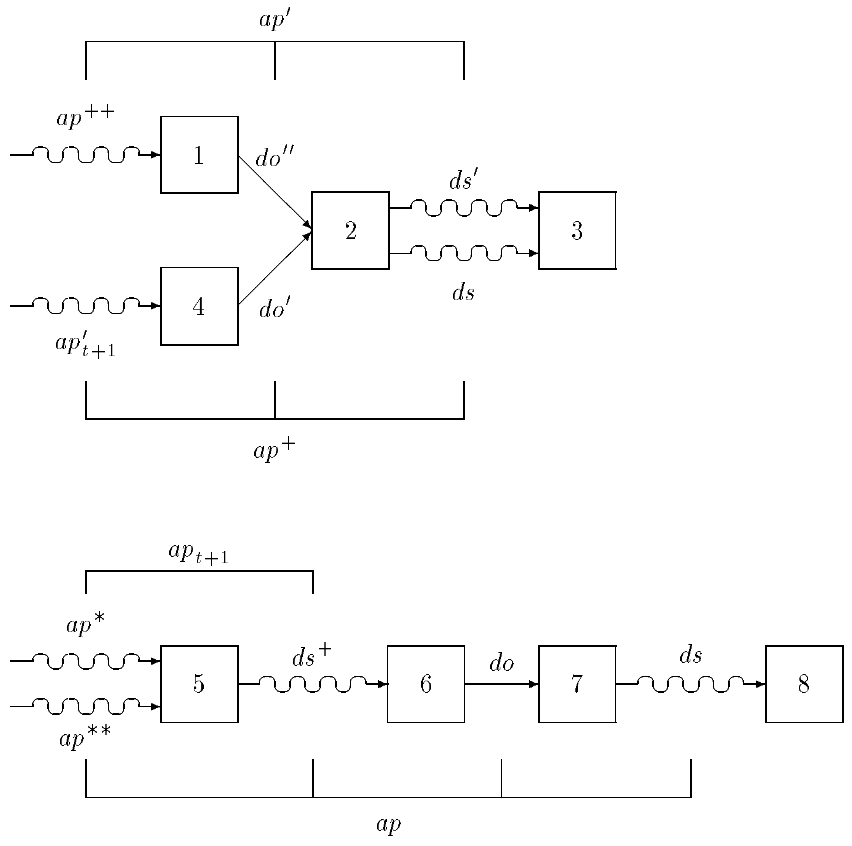
$$\begin{gathered}\exists(ap_{t+1}, ap'_{t+1}, ap^{+}, ap^{++}, ap^{*}, ap^{**} \in AP, \quad do, do', do'' \in DEROP, \\ ds, ds', ds^{+} \in DOSEQ) \ :\end{gathered}$$

$$\begin{gathered}(ap'_{t+1} \in acc\_path\_set_{fix}(exp'_{t+1})) \ \wedge \\ (loc(ap^{+}, st_{\overline{ass}_{t+1}}) = loc(ap', st_{\overline{ass}_{t+1}})) \ \wedge \\ (loc(concat(ap^{++}, (do'')), st_{\overline{ass}_{t+1}}) = loc(concat(ap'_{t+1}, (do')), st_{\overline{ass}_{t+1}})) \ \wedge \\ (ap^{+} = concat(concat(ap'_{t+1}, (do')), ds)) \ \wedge \\ (ap' = concat(concat(ap^{++}, (do'')), ds')) \ \wedge\end{gathered} \tag{3.2}$$

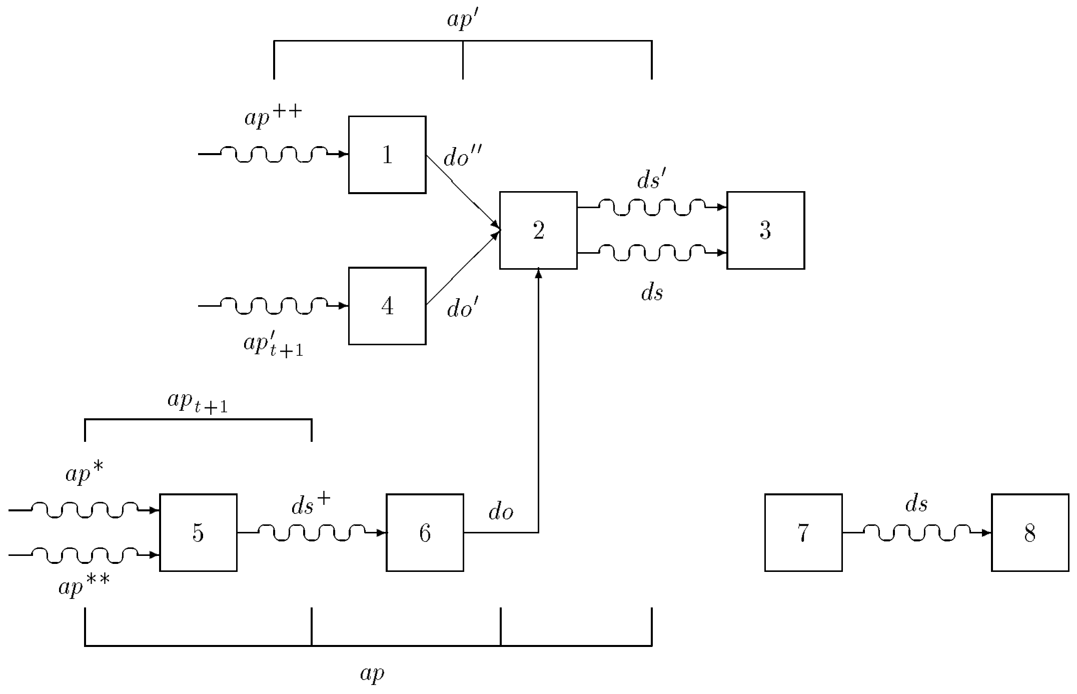
$$\begin{gathered}(ap_{t+1} \in acc\_path\_set_{fix}(exp_{t+1})) \ \wedge \\ (loc(ap^{**}, st_{\overline{ass}_{t+1}}) = loc(ap^{*}, st_{\overline{ass}_{t+1}})) \ \wedge \\ (ap_{t+1} = concat(ap^{*}, ds^{+})) \ \wedge \\ (ap = concat(concat(concat(ap^{**}, ds^{+}), (do)), ds))\end{gathered} \tag{3.3}$$

The memory state before and after the assignment has been executed is shown in figure 3.57. Note that this graph represents the real memory locations and is no function interface graph.

As can be proved by inspecting the grammar rules, the construction of the rules is fashioned in a way that the attribute *val* always holds the value node corresponding to the access path of the current subexpression. Further, the



Memory state before the execution of the assignment ass_{t+1} $(mem(st_{\overline{ass}_{t+1}}))$



Memory state after the execution of the assignment ass_{t+1} $(mem(st_{\underline{ass}_{t+1}}))$

Figure 3.57: Memory state before and after the execution of an assignment

attribute *addr* always holds the value node corresponding to the address of the current subexpression. The rule handling the assignments then connects the value node corresponding to the expression on the right hand side of an assignment to the value node corresponding to the address of the left hand side of the assignment. This means that this value node is contained in the set of value nodes corresponding to the access paths ap_{t+1} and ap'_{t+1} then.

$$\exists(vn \in VN_{fig}) \quad : \quad vn \in (acc\_vals(fig, ap_{t+1}) \cap acc\_vals(fig, ap'_{t+1}))$$

Moreover, there is only one single node vn contained in the above given sets because the grammar rules unite the value nodes in case more than one exist. Since there is an unique value node corresponding to these access paths it can be derived from (3.2) and (3.3) together with the supposition that the value node vn can as well be accessed by two sub-access paths of ap and ap'.

$$\begin{gathered} vn \in acc\_vals(fig, ap^{++}) \\ vn \in acc\_vals(fig, concat(ap^{**}, ds^{+})) \end{gathered}$$

According to the supposition and (3.3) there has to be a value node vn' accessible using either access path ap' or ap^{+}.

$$\exists(vn' \in VN_{fig}) \quad : \quad vn' \in (acc\_vals(fig, ap') \cap acc\_vals(fig, ap^{+}))$$

Since both of these access paths use the value that has been assigned to exp_{t+1}, and is, hence, represented by a single value node (vn), this means that the value node vn' can be reached from the node vn using either the dereference operation sequence ds or ds'. This however means that vn' can as well be reached by using the access path ap, which was to be proved.

$$vn' \in (acc\_vals(fig, ap) \cap acc\_vals(fig, ap'))$$

The only fact that has not been taken into account so far is that the assignments can be represented by different function interface graphs because they belong to different CFG nodes. However, the function merging the function interface graphs is constructed in a way that all nodes that can be reached by the same access paths in the function interface graphs to be merged will later be represented by the same node in the resulting function interface graph. Hence, the above made assumptions will hold even if the merged final function interface graph is inspected.

Now, that it has been shown that if two expressions may be aliased, a value node exists contained in either set of access paths belonging to the two given expressions, it is obvious that the *fig_alias* function will indicate the two expressions to be aliases as it should be, and hence our treatment on the intraprocedural level is correct.

3.2.7 Time and space bounds

Both the time and space bounds are mainly influenced by the merging of the function interface graphs belonging to the different CFG nodes. The generation of the function interface graphs needs time and space proportional to the input size (number of statements and ϕ-functions) of the analysed code ($O(|STMT| + |PHI|)$) since the only thing that has to be done is to generate nodes and edges in the function interface graph that correspond to a given access path. The access paths grow (at most) linearly to the size of the expressions and so the number of nodes and edges that have to be generated is linear to the size of the analysed code as well.

3.2.7.1 Time bounds

The time needed to merge two function interface graphs (fig_1 and fig_2) is proportional to the product of their value node numbers ($O(|fig_1.VN| \cdot |fig_2.VN|)$) in the worst case. This results from the fact that the algorithm walks through the two graphs in parallel and that each pair of nodes is at most visited twice. However, it is not very likely that every pair of nodes will be visited. Even if this should happen it would mean that almost all nodes have to be united to one node which would reduce the size of the resulting graph and respectively the costs of the following merging steps significantly.

In many cases there are nodes in the second graph that are not matched since they are only reachable by assigned edges. Furthermore, not every combination of value nodes is reachable by matching paths which decreases the time needed to process the two graphs as well. Altogether the average time needed to merge two graphs is more likely to grow linearly to the size of the value node sets ($O(|fig_1.VN| + |fig_2.VN|)$).

Assuming this linear time behaviour, the overall time to process a function is proportional to ($O(|E| \cdot (|STMT| + |PHI|))$) where $|E|$ is the number of CFG edges. Even if one or more merging steps become comparatively expensive this will automatically reduce the costs for other (following) merging steps, since the resulting graph gets smaller the longer the merging process takes, because more and more nodes are then united. As can be seen from the results of the tests shown in section 4.4, the average time needed to analyse a statement grows almost proportionally to the function size. Assuming that the sizes of the functions do not grow beyond a certain limit, the overall analysis time will then grow proportionally to the program size.

3.2.7.2 Space bounds

The space needed by the intraprocedural analysis grows linearly to the size of the input code ($O(|STMT| + |PHI|)$). As already mentioned before, the space needed for a function interface graph belonging to a CFG node is proportional to the size of the code belonging to this CFG node. This means that the space needed for all the function interface graphs is proportional to the input code size. When the merging process starts, these function interface graphs are added to the resulting graph one by one. Even if there were no nodes to be united during the merging process the resulting graph would not be bigger than all the single function interface graphs taken together. So the space needed is at most $2 \cdot (|STMT| + |PHI|)$ which obviously is in $O(|STMT| + |PHI|)$.

Now the intraprocedural part of the analysis is complete. It has been explained how the function interface graphs for each function contained in a program can be computed.

These graphs represent all the effects a function can have on any memory location without taking other functions calling this function or called by this function into account. Using these function interface graphs, all local aliases which do not depend on the called or calling functions can be detected.

3.3 Interprocedural analysis

Now that the intraprocedural part of the analysis is completed, it is about time to combine the function interface graphs of the different functions to build a single graph representing the effects of the whole program. To do this, it becomes necessary to check which other functions are called by each function to build a call graph. If the program contains function calls using function pointers, only a first approximation of the call graph can be built. Starting with this graph, the function calls are processed one by one. This could then lead to the detection of new functions being called by function calls using function pointers, in case such function calls occur in the program. This process continues until no more function calls are left to process. After that the final combined function interface graph representing all possible aliases that may occur during the program execution is complete.

3.3.1 Basic definitions

Before going into the details of the call graph computation and explaining how the different function interface graphs are combined, some sets and functions that will be used later have to be defined first. These sets and functions are necessary to associate variables, function calls and functions with each other.

As explained in section 2.3, the functions and function calls have been normalised. Therefore, every function call has either the form

$$\mathtt{v_0} = \mathtt{f}(\mathtt{v_1}, \ldots, \mathtt{v_n});$$

or simply

$$\mathtt{f}(\mathtt{v_1}, \ldots, \mathtt{v_n});$$

where $\mathtt{v_0}, \mathtt{v_1}, \ldots, \mathtt{v_n}$ are variables[42] and `f` is a function typed expression. Further, every statement returning a functions result value has the form

$$\textbf{return}\ \mathtt{v};$$

where `v` is a variable. Hence, every value passed to a function or returned by a function can be identified by the variable holding this value. When functions and function calls are formally defined in the following, their definition can be based upon the variables associated with the variables corresponding to the parameters and return values.

The set *CALL* is the set of function calls occurring in a program. Every function call $c \in CALL$ consists of an unique number[43] identifying the function call, a variable or the value *nil* specifying the value to which the functions result is assigned and a sequence of variables passed to the called function. The sequence of variables contains the variables in order of appearance in the function call.

Definition 51 (functions calls)

$$CALL \subset (\mathbb{N} \times (VAR_{SSA} \cup \{nil\}) \times VAR_{SSA}{}^{*}) \quad \text{where}$$
$$\forall (c, c' \in CALL) \ :$$
$$((c = (i, v_0, v_1 \ldots v_s)) \ \wedge \ (c' = (i', v'_0, v'_1 \ldots v'_{s'})) \ \wedge \ (c \neq c')) \ \Rightarrow \ i \neq i'$$

[42]Here as well as in the following definitions the subscripts used by the variables are not meant to be the SSA values, they are only used to enumerate the variables. If the SSA values are to be processed this is done by using the *ssa_val* function.

[43]The only purpose of this number is to be able to distinguish between function calls that would otherwise be represented by the same tuple.

The set *FUNC* is the set of functions occurring in a program. Every function $f \in FUNC$ consists of an unique number[44] identifying the function, the variable used by the **return** statement of that function[45] or *nil* (if no such variable exists), a sequence of variables specifying the formal parameters of the function and a sequence of function calls contained in the function. The sequence of variables contains the variables in order of appearance in the function header. As well the sequence of function calls contains the function calls in order of their textual appearance in the function.

Definition 52 (functions)

$$\begin{gathered} FUNC \subset (\mathbb{N} \times (VAR_{SSA} \cup \{nil\}) \times {VAR_{SSA}}^{*} \times CALL^{*}) \quad \text{where} \\ \forall (f, f' \in FUNC) \ : \\ (f = (i, v_0, v_1 \ldots v_s, c_1 \ldots c_t)) \wedge (f' = (i', v'_0, v'_1 \ldots v'_{s'}, c'_1 \ldots c'_{t'})) \wedge (f \neq f') \\ \Rightarrow \\ i \neq i' \end{gathered}$$

Since functions can contain more than one function call, it is important to be able to distinguish between the different function calls contained in a particular function. The following functions can be used to identify a certain function call by assigning an unique number to each call contained in a function.

Definition 53 (functions and corresponding function calls)

$$\begin{gathered} nth\_call \ : \ FUNC \times \mathbb{N} \nrightarrow CALL \ , \ call\_no \ : \ CALL \nrightarrow \mathbb{N} \\ nth\_call((i, v_0, v_1 \ldots v_s, c_1 \ldots c_t), n) = c_n \quad \text{if } (1 \leq n \leq t) \\ call\_no(c) = n \quad \text{if } \exists (f \in FUNC) : nth\_call(f, n) = c \end{gathered}$$

The function *nth_call*[46] returns the n-th function call (in textual order) occurring in the function, whereas the function *call_no* returns the number of the function call inside the function containing this call. Using these functions we are now able to distinguish between the different function calls occurring in a certain function.

Besides these two functions there is another useful function, which deals with the functions occurring in the analysed program. The function *params* can be used to determine the number of parameters of a function or a function call.

Definition 54 (number of parameters of functions and function calls)

$$\begin{gathered} params \ : \ (FUNC \cup CALL) \nrightarrow \mathbb{N} \\ params(x) = s \text{ if} \\ (((x \in FUNC) \ \wedge \ (x = (i, v_0, v_1 \ldots v_s, c_1 \ldots c_t))) \ \vee \\ ((x \in CALL) \ \wedge \ (x = (i, v_0, v_1 \ldots v_s)))) \end{gathered}$$

[44]Like the numbers identifying the function calls, the numbers identifying the functions are only needed to make the functions distinguishable.

[45]As stated in section 2.3, every function has at most one **return** statement after it has been normalised.

[46]This function is only partially defined. However, it is not necessary to define a value for $n > t$ since this value will not be used anywhere.

Given a function or function call x, the function *params* returns the number of formal parameters of x in case x is a function, and the number of values passed to the function contained in x in case x is a function call.

The next thing to be done is to associate the variables with the function calls and the functions. This can be done with the help of the next two functions. The first one returns the variables corresponding to the parameters of functions or function calls.

Definition 55 (parameters of functions and function calls)

$$\mathit{param} \;:\; (\mathit{FUNC} \cup \mathit{CALL}) \times \mathbb{N} \nrightarrow \mathit{VAR}_{SSA}$$
$$\mathit{param}(x, n) = v_n \text{ if}$$
$$(((x \in \mathit{FUNC}) \;\wedge\; (x = (i, v_0, v_1 \ldots v_s, c_1 \ldots c_t)) \;\wedge\; (1 \leq n \leq s)) \;\vee$$
$$((x \in \mathit{CALL}) \;\wedge\; (x = (i, v_0, v_1 \ldots v_s)) \;\wedge\; (1 \leq n \leq s)))$$

Given a function or function call x and a positive number n, the function *param*[47] returns the variable corresponding to the n-th formal parameter of x in case x is a function, and the n-th variable passed to the function called by x in case x is a function call.

Now the values passed to a function have been handled, and it remains for the values returned by a function to be handled as well, which is done by the next function.

Definition 56 (return values of functions and function calls)

$$\mathit{result} \;:\; (\mathit{FUNC} \cup \mathit{CALL}) \nrightarrow \mathit{VAR}_{SSA} \cup \{\mathit{nil}\}$$
$$\mathit{result}(x) = \begin{cases} v_0 & \text{if } (((x \in \mathit{FUNC}) \;\wedge\; (x = (i, v_0, v_1 \ldots v_s, c_1 \ldots c_t))) \;\vee \\ & \quad ((x \in \mathit{CALL}) \;\wedge\; (x = (i, v_0, v_1 \ldots v_s)))) \\ \mathit{nil} & \text{otherwise} \end{cases}$$

Given a function or function call x, the function *result* returns the variable that is contained in the **return** statement of x in case x is a function and a value is returned by the function. Should x be a function call, the variable to which the functions result is assigned by x is returned if such a variable exists.

Besides the functions defined up to here there are three more functions that will be needed later. The first of these functions determines if a certain variable v is an input or output variable of a function or a function call. The input variables of functions are the variables corresponding to the formal parameters, whereas the output variables are the variables returned by the **return** statements. Accordingly, the input variables of function calls are those that are passed to the called functions, whereas the output variables are the variables to which the values returned by the called functions are assigned. The input and output variables are exactly those values that are exchanged directly between the calling and the called functions.

Definition 57 (input and output variables)

$$\mathit{is\_io\_var} \;:\; \mathit{VAR}_{SSA} \times (\mathit{FUNC} \cup \mathit{CALL}) \nrightarrow \mathbb{B}$$
$$\mathit{is\_io\_var}(v, x) = (v \in \{v_0, v_1, \ldots, v_s\}) \text{ if}$$
$$(((x \in \mathit{FUNC}) \;\wedge\; (x = (i, v_0, v_1 \ldots v_s, c_1 \ldots c_t))) \;\vee$$
$$((x \in \mathit{CALL}) \;\wedge\; (x = (i, v_0, v_1 \ldots v_s))))$$

[47]Like the function *nth_call* this one is only partially defined.

The second function can be used to find out if a certain variable is a global variable or not. Here global variable means any variable with static storage duration. This includes all variables declared outside of a function (globals), as well as those declared within a function using the **static** attribute. The variables scope is not of any interest here. Since values can be assigned to a static variable in one call, and later the value of this static variable can be assigned to another variable in a second call, static variables may not be treated like other local variables[48].

Definition 58 (global variables)

$$\begin{aligned} is\_global &: VAR_{SSA} \nrightarrow \mathbb{B} \\ is\_global(v) &= \begin{cases} \textbf{true} & v \text{ is a global or static variable} \\ \textbf{false} & \text{otherwise} \end{cases} \end{aligned}$$

Up to here the SSA numbers of the variables used in the program did not influence any of the defined functions. Variables which only differ in their SSA values have been treated as independent variables up to here. However, it is necessary to know which variables are valid when a function is called. Here valid means that the variable has a SSA number corresponding to the program point where the function call is located.

The function *is_valid* can be used to find the variables having the correct SSA numbers for a given function call. In the following it is assumed that n_c is the CFG node corresponding to the function call c and that n_v and $n_{v'}$ are the CFG nodes corresponding to the assignment statements where the values of the variables v and v' are set[49].

Definition 59 (valid variables)

$$\begin{aligned} is\_valid &: VAR_{SSA} \times CALL \nrightarrow \mathbb{B} \\ is\_valid(v,c) &= \exists (p \in path(n_v, n_c)) : \\ &\qquad \forall (v' \in VAR_{SSA}, p' \in path(n_{v'}, n_c)) : \\ &\qquad\quad (base\_var(v) \neq base\_var(v')) \vee \\ &\qquad\quad (length(p) < length(p')) \vee \\ &\qquad\quad ((length(p) = length(p')) \wedge \\ &\qquad\quad\ (ssa\_val(v) > ssa\_val(v'))) \end{aligned}$$

Although this might look quite complicated at first glance, it is not as bad as it seems. The function checks if the current SSA variable is the most recently defined SSA variable among those SSA variables corresponding to the same variable in the original program. Therefore this variable has to be defined in the closest CFG node defining one of the SSA variables corresponding to the same variable in the original program (this is checked using the *length* function). Further, if there is more than one of these SSA variables defined in this CFG node, the least recently defined one has to be chosen. Since the algorithm computing the SSA values increases the SSA value for every assignment to a SSA variable, the last one is as well the one having the largest SSA value, which is checked by the function.

Now that the actual SSA values of variables at all function call locations can be determined, the next function will determine the SSA values valid at the end of a function.

[48]Local variables declared using the **static** attribute can be viewed as global variables with local scope. Anyway a compiler would usually store static variables together with the global variables at the same location.

[49]Since the program is in SSA form, every variable is assigned a new value at most once. In case the variable was assigned by a statement a definite CFG node exists (the node corresponding to the assignment statement). Otherwise, the variable has never been assigned within the function, and the *START* node will be used then.

Definition 60 (final variables)
The function *is_final* determines whether the SSA value of a variable is valid at the end of a function or not.

$$\textit{is\_final} \;:\; \textit{VAR}_{SSA} \times \textit{FUNC} \nrightarrow \mathbb{B}$$

$$\textit{is\_final}(v, f) = \begin{cases} \textbf{true} & \text{if } v\text{'s SSA value is valid at the end of function } f \\ \textbf{false} & \text{otherwise} \end{cases}$$

Of course the valid SSA values at the beginning of the function are as interesting as those at the end of the function. Nevertheless, there is no function necessary to compute these values. This is because all SSA values are zero in this case.

As will be seen later there are two more properties of functions and function calls that are of interest. The first is whether a function contains function calls using function pointers or not. The function *ptr_call* determines whether a given function call is a call using a function pointer or if a function contains such a call.

Definition 61 (function calls using function pointers)

$$\textit{ptr\_call} \;:\; (\textit{FUNC} \cup \textit{CALL}) \nrightarrow \mathbb{B}$$

$$\begin{aligned} \textit{ptr\_call}(x) = (((&x \in \textit{CALL}) \;\wedge\; (x \text{ uses a function pointer})) \;\vee \\ ((&x \in \textit{FUNC}) \;\wedge\; (x = (i, v_0, v_1 \ldots v_s, c_1 \ldots c_t)) \;\wedge \\ (&\textit{ptr\_call}(c_1) \vee \ldots \vee \textit{ptr\_call}(c_t)))) \end{aligned}$$

The second matter of interest is whether the address of a function has been taken somewhere in the program. The function *addr_taken* can be used to detect if the address of a given function has been taken.

Definition 62 (functions of which the address has been taken)

$$\textit{addr\_taken} \;:\; \textit{FUNC} \nrightarrow \mathbb{B}$$

$$\textit{addr\_taken}(f) = \begin{cases} \textbf{true} & \text{if the address of } f \text{ has been taken} \\ \textbf{false} & \text{otherwise} \end{cases}$$

3.3.2 Building the call graph

The interprocedural part of the analysis cannot start without computing the call graph of the program that has to be analysed first. Basically, there are two different types of function calls that are represented by the call graph.

- ordinary function calls like `f(...)` where `f` is a function
- function calls using function pointers like `(*f)(...)` where `f` is a pointer to a function and function calls using function variables like `f(...)` where `f` is a function variable[50]

The call graph computation is quite simple for programs which neither contain function variables nor function pointers. In this case, the exact call graph can be statically computed within a single pass over the program code at compile time. Even if the program contains recursive functions this does not cause any problems. A simple example program using only ordinary function calls and its corresponding call graph is shown in figure 3.58. The

```
void main()
{
  int a, b;
  a = dblsqr(3);
  b = sqrdbl(3);
}

int dblsqr(int x)      int sqrdbl(int x)
{                      {
  int d;                 int d;
  d = sqr(x);            d = dbl(x);
  return dbl(d);         return sqr(d);
}                      }

int dbl(int x)         int sqr(int x)
{                      {
  return x + x;          return x*x;
}                      }
```

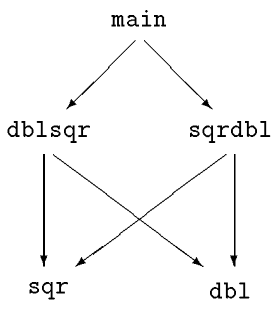

$$\begin{tikzcd}
 & \texttt{main} \arrow[dl] \arrow[dr] & \\
\texttt{dblsqr} \arrow[d] \arrow[drr] & & \texttt{sqrdbl} \arrow[d] \arrow[dll] \\
\texttt{sqr} & & \texttt{dbl}
\end{tikzcd}$$

Figure 3.58: Simple program using only ordinary function calls and its corresponding call graph

program consists of five functions. The two basic functions `dbl` and `sqr` are called by the functions `dblsqr` and `sqrdbl` which themselves are called by the main function `main`.

The major difficulties caused by call graph computation concern the handling of function calls using the second method to call a function. In this case the situation becomes much more complicated, since there is no unique called function belonging to a certain function call any more. Hence it is not known at compile time which functions will be called by a certain function call, and so the call graph cannot be computed exactly in general. In such cases it becomes necessary to approximate the set of functions that may be called for every function call.

The example program shown in figure 3.59 shows a program using function pointers and the corresponding call graph. The program is equivalent to the preceding program, beside the fact that the two quite similar functions `dblsqr` and `sqrdbl` have been replaced by a single function `disp`. Instead of calling the basic functions `dbl` and `sqr` directly they are called by pointers. Which function will be called is specified by the function call contained in the main program `main`. To make the different types of function calls visible in the call graph the edges belonging to function calls using function pointers are marked with a star ($*$).

Since the C programming language hardly restricts the usage of function pointers it becomes almost impossible to statically compute the exact call graph. To do this an execution of the program covering every possible set of inputs which keeps track of the called functions is needed. This would lead to unreasonable costs in time for all but the easiest programs if the program contains function pointers. Therefore a solution has to be found that neither is too expensive, nor makes overly conservative assumptions. Before going into the details of the call graph computation, the call graphs will be defined first.

[50]The C programming language does not support function variables. Nevertheless they could easily be handled by our algorithm as well.

```
void main()
{
  int a, b;
  a = disp(3, &dbl, &sqr);
  b = disp(3, &sqr, &dbl);
}

int disp(int x, int (*f1)(int), int (*f2)(int))
{
  int d;
  d = f2(x);
  return f1(d);
}

int dbl(int x)          int sqr(int x)
{                       {
  return x + x;           return x*x;
}                       }
```

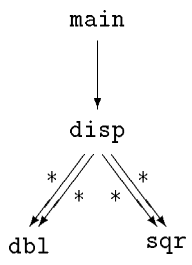


Figure 3.59: Sample program using function pointers and its corresponding call graph

Definition 63 (call graph)
A call graph is an annotated directed multi graph

$$CG = (V, E)$$
$$\text{where } V = \mathit{FUNC} \text{ and } E = (\mathit{FUNC} \times \mathit{FUNC} \times \mathbb{N})$$

are the sets of call graph nodes and edges. Further there are the three functions

$$\begin{array}{rcl} \mathit{calling} & : & E \nrightarrow \mathit{FUNC} \\ \mathit{called} & : & E \nrightarrow \mathit{FUNC} \\ \mathit{no} & : & E \nrightarrow \mathbb{N} \end{array}$$

that take a call graph edge as their input and return one of the edges components. They are defined as follows.

$$\begin{array}{lcl} \mathit{calling}((f_1, f_2, n)) & = & f_1 \\ \mathit{called}((f_1, f_2, n)) & = & f_2 \\ \mathit{no}((f_1, f_2, n)) & = & n \end{array}$$

An edge $(f_1, f_2, n) \in E$ is contained in the call graph if and only if there exists a function call $c \in \mathit{CALL}$ contained in function f_1 where $c = \mathit{nth\_call}(f_1, n)$ and c might call function f_2.

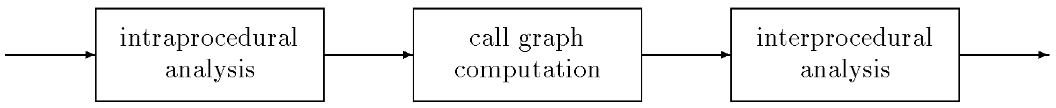
As can be seen from the definition, the call graph edges consist of three components. The first component (f_1) is the calling function, which as well is the source node of the call graph edge, whereas the second component (f_2) is the called function, which as well is the destination node of the call graph edge. The third component (n) contains the unique number identifying which of the calls contained in the calling function belongs to the call graph edge. These components can be accessed using the functions *calling*, *called* and *no*.

The last part of the definition contains an additional condition. This condition makes sure that the call graph contains an edge from every function containing a function call to every function that might be called by this particular call. There are no further edges contained in the call graph. Since functions can contain multiple calls, it is necessary to distinguish between the different calls contained in a function. This is done with the help of the third component of the call graph edges which specifies which function call is meant.

As mentioned earlier the call graph cannot be computed completely before the interprocedural analysis starts. Depending on the analysed program it might be necessary to update the call graph by adding new edges for the function calls using function pointers. This is taken into account by the call graph definition since it depends on which function calls *might* call which functions. This however depends on the information so far collected by the analysing program. Therefore the call graph is not static but depends on the information gathered at a specific stage of the analysis.

As already mentioned before it is quite simple to compute the call graph as long as a program does not use function pointers or function variables, whereas it is almost impossible to compute an exact call graph otherwise. Nevertheless, there have been several approaches to deal with the problems which arise when the call graph for a program using function pointers or function variables has to be computed. For example, the algorithm presented in [HK92] keeps track of the values of function variables. Unfortunately, this algorithm is not able to deal with function calls through function pointers or function variables contained in a structure.

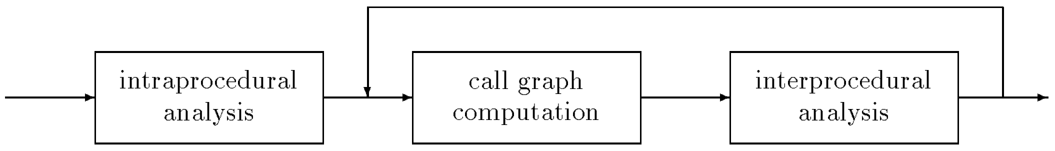
All the algorithms computing call graphs can be separated into two different groups: those restricting the use of function pointers or variables in a certain way (e.g. [HK92]) and those that do not. The first group of algorithms is sometimes able to compute an exact solution depending on the restrictions they make, whereas the second group always computes approximated call graphs only. Although the algorithms of the second group are less precise, the algorithm used here belongs to this group. This is because one of the main aims of the alias analysis presented here is to be able to deal with almost all the properties of the C programming language, which makes the algorithms of the first group completely unsuitable here. If an algorithm of the first group had been chosen this would either lead to some considerable restrictions to the language which are not desired here, or to a call graph having too few call edges which could lead to aliases that may not be found by the alias analysis. Especially the almost unlimited possibilities to manipulate pointers within C programs make it impossible to build a really precise call graph within a reasonable period of time. The major advantage of the algorithms of the first group is that the following analysis steps (the interprocedural parts) can use a call graph that is complete when the analysis starts, and that they can rely on the fact that it will not be modified during the following analysis steps.



Algorithms of the second group either do not have this property or are extremely imprecise. This is because the computation of the sets of functions that may be called can be reduced to the alias problem itself. In this case, the only thing of interest is to find out which function addresses a certain function pointer can hold. This is equivalent to the problem of finding the aliases of the function pointers.

A trivial solution to this problem would be to assume that every pair of variables may be aliases. Of course this solution is unsuitable for the alias analysis itself, but it may be used for the call graph computation. In this case every function call using a function pointer has to be expected to call any of the functions having a matching type if the address of such a function has been taken somewhere in the program[51]. Obviously this will lead to an extremely imprecise call graph in many cases, but it is the only way to build a complete call graph in advance without restricting the language.

If a more precise call graph shall be computed the only thing that can be done is to combine the call graph computation with the interprocedural analysis (especially the alias analysis). By doing this, one can start with an incomplete call graph and let the alias analysis add edges to this call graph every time a new function that may be called by a certain function call is found. The changed call graph may then be used to compute previously undetected aliases and new functions that may be called by one of the function calls respectively. Finally, when stabilisation is reached, the call graph computation as well as the alias computation itself are completed. One of the algorithms using this scheme to update the call graph as well as the results of the interprocedural analysis step repeatedly is described in [Wei80].



In many cases the algorithms which update the call graph step by step are not as fast as the ones which compute a final call graph before the interprocedural analysis starts. This is caused by the repeated updating of the interprocedural results which can sometimes become comparatively costly. As will be seen later, this does not hold for the algorithm presented here, since it is not necessary to update any of the previously computed results when a new call graph edge has been found. The only thing that has to be done is to process the new edge, just like the other edges that have been available right from the start. Hence the algorithm presented here combines the benefits of being as fast as if the complete call graph has been computed before the interprocedural analysis starts and of being as precise as possible without restricting the programming language.

The initial call graph that is used by the interprocedural analysis contains only edges for function calls where the called function can be statically computed (those not using function pointers). Therefore the computation of this call graph is as simple as for those programs which do not contain function pointers, since the calls using function pointers can simply be ignored in this first step.

3.3.3 Combining the function interface graphs produced by the intraprocedural analysis

Now, after an initial call graph has been built, it will be described how the function interface graphs belonging to different functions are combined. Before the matching process itself is described, the next section shows how the function interface graphs can be reduced to make the merging process more efficient.

[51]Note that the effects of function calls using function pointers are undefined if the type of the function pointer does not match the type of the called function ([Ame89a]). Therefore it is sufficient to assume that only calls with matching types occur in a program.

3.3.3.1 Reducing the function interface graph for the interprocedural analysis

The function interface graphs produced by the intraprocedural analysis represent every memory access a function can perform. Since not every memory access contained in a function can influence the behaviour of calling or called functions, the function interface graphs can be reduced before they are combined with other function interface graphs representing calling or called functions. After the function interface graphs have been thinned out, only those effects of a function that may introduce new aliases are represented by these reduced function interface graphs.

Of course some information is lost after the function interface graphs have been reduced, and this can possibly lead to undetected (local) aliases. Therefore it is necessary to keep a copy of the non-reduced function interface graph when the graph is reduced. This keeps the pure local effects of a function still detectable since they can be computed by inspecting the non-reduced function interface graph. Nevertheless the non-local effects can be computed more efficiently since they can take advantage of the smaller reduced function interface graphs. In this context it is helpful to have a proper definition of what is meant by local and non-local effects.

Definition 64 (local and non-local e-aliases)
Given a function f and two variables v and v' from which the same memory location can be accessed (according to definition 42 the variables are e-aliases), then these variables are called local e-aliases if they are e-aliases in every possible program containing the function f, and non-local e-aliases otherwise.

Since it is not possible to compute the effects of all programs containing a given function, local and non-local e-aliases cannot be distinguished by checking if the definition holds. However, local e-aliases are exactly those aliases that can be detected by inspecting a single function without looking at the calling or called functions, which obviously is much easier to check. All local e-aliases are represented by the non-reduced function interface graph. In many cases, when only local variables are involved, the local e-aliases are only of minor interest for other functions. The non-local e-aliases can be detected by inspecting the final function interface graph that is built by combining the reduced function interface graphs of the different functions. To be able to detect all non-local aliases it is essential that the reduced function interface graphs still have enough information to represent every memory access that might lead to non-local e-aliases.

Before the differences between local and non-local aliases and the corresponding function interface graphs are discussed further, it is defined which information, that is which nodes and edges, is removed from the function interface graphs when preparing for the merging process. The definition of the reduced function interface graphs uses the function *reachable* that is defined first.

Definition 65 (reachable nodes)
Given a function interface graph *fig* with $fig = (V, E)$, then the function *reachable* is defined as follows.

$$\mathit{reachable} : V \times V \Leftrightarrow\!\!\!\rightarrow \mathbb{B}$$

$$\mathit{reachable}(n_1, n_2) = \begin{cases} \exists (n \in V, e \in E)) : \mathit{reachable}(n, n_2) \wedge (\mathit{src}(e) = n_1) \wedge (\mathit{dst}(e) = n) & \text{if } n_1 \neq n_2 \\ \textbf{true} & \text{if } n_1 = n_2 \end{cases}$$

This function can be used to find out if a certain node (n_2) of a function interface graph can be reached from another node (n_1) of that graph.

Definition 66 (reduced function interface graph)
Given a function f and its corresponding function interface graph *fig*, the function *reduce* computes the corresponding reduced function interface graph fig_{red}.

$$\mathit{reduce} \;:\; \mathit{FIG} \times \mathit{FUNC} \nrightarrow \mathit{FIG}$$
$$\mathit{reduce}(\mathit{fig}, f) = \mathit{fig}_{red}$$

where

$$\mathit{fig} = (V, E), \quad \mathit{fig}_{red} = (V_{red}, E_{red})$$

and

$$\begin{aligned}
V_{red} \;=\; & \{n_v^* \in V \mid \exists (v \in \mathit{VAR}_{SSA}, n_v \in V) \;: \\
& \quad (v \in \mathit{vars}(n_v)) \;\wedge\; \mathit{reachable}(n_v, n_v^*) \;\wedge \\
& \quad ((\mathit{is\_io\_var}(v, f) \;\vee\; (\exists (c \in \mathit{CALL}, n \in \mathbb{N}) : \\
& \quad\; (c = \mathit{nth\_call}(f, n)) \wedge \mathit{is\_io\_var}(v, c))) \;\vee \\
& \quad (\mathit{is\_global}(v) \;\wedge\; ((\mathit{ssa\_val}(v) = 0) \;\vee \\
& \quad\; (\exists (c \in \mathit{CALL}, n \in \mathbb{N}) : (c = \mathit{nth\_call}(f, n)) \wedge \mathit{is\_valid}(v, c)) \;\vee \\
& \quad\; (\mathit{is\_final}(v, f))))))\} \\
E_{red} \;=\; & \{e \in E \mid (\mathit{src}(e) \in V_{red}) \;\wedge\; (\mathit{dst}(e) \in V_{red})\}
\end{aligned}$$

Only nodes that can be reached from either a node belonging to a global variable or from a node belonging to a function parameter (including return values) are kept in the reduced function interface graph. However, not all nodes belonging to global variables are kept. Only those nodes belonging to global variables whose SSA values are valid at either the beginning or the end of the function, or at places where other functions are called, are chosen. Furthermore, only those edges are kept that connect two of these nodes. All other nodes and edges belong to memory locations that cannot be accessed from the outside of the function, and so they are not of any interest for the interprocedural analysis. Remember that the aliases between local variables can still be detected since a copy of the non-reduced function interface graph is kept.

The example shown in figure 3.60 shows a function in its original form as well as in SSA form. The corresponding function interface graph, as well as the reduced function interface graph are shown in figure 3.61. Like in the previous examples an integer size of two bytes, a pointer size of four bytes and a byte-wise alignment are assumed. As can be seen from this example, the nodes and edges that are only reachable from the nodes corresponding to the local variables that are not used for input or output ($\texttt{ptr}_1, \texttt{ptr}_2, \texttt{ptr}_3, \texttt{ptr}_4, \texttt{c}_1, \texttt{c}_2, \texttt{c}_3, \texttt{c}_4$) have been removed in the reduced function interface graph. Although the variables $\texttt{list}_0, \texttt{n}_0$ and $\texttt{res}_1$ are local variables as well, the nodes reachable from the nodes corresponding to these variables are included in the reduced function interface graph since the first two of them are formal parameters and the last one is the value returned by the function.

3.3.3.2 Merging the function interface graphs for a single function call

Every function call has one or more corresponding edges in the CFG graph. Each of these edges connects the function containing the function call with one of the functions that may be called by this particular call. To represent the effects of such a possible call correctly, the

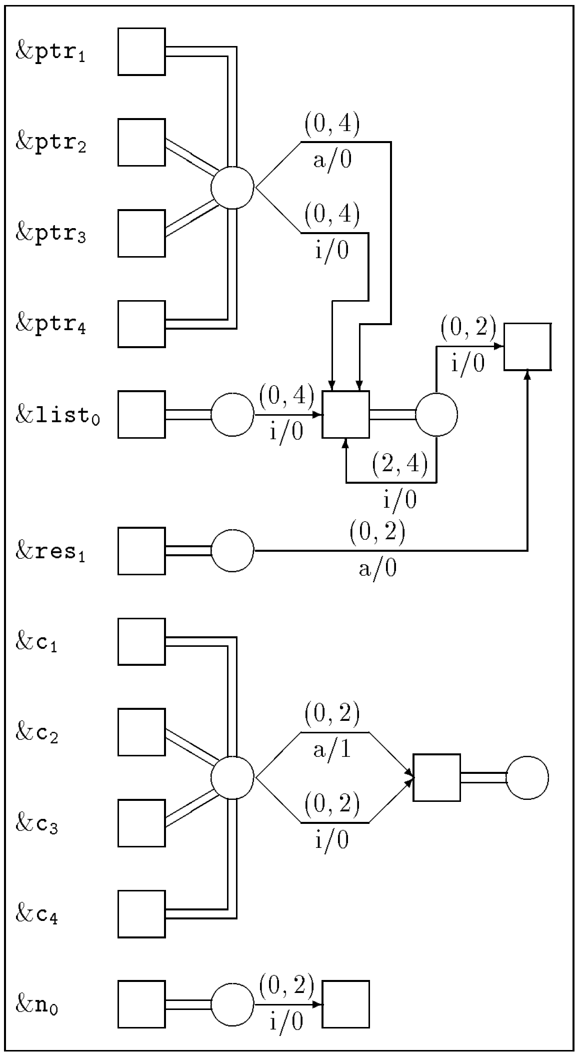
```
typedef struct list {
  int i;
  struct list *next;
} *list_ptr;

list_ptr nth_ele(list_ptr list, int n)
{
  int res, c;
  list_ptr ptr;
  ptr = list;
  c = 1;
  if (n > 1) {
    do {


      ptr = *(ptr.next);
      c = c + 1;
    } while (c < n)
  }


  res = *ptr.i;
  return res;
}
```

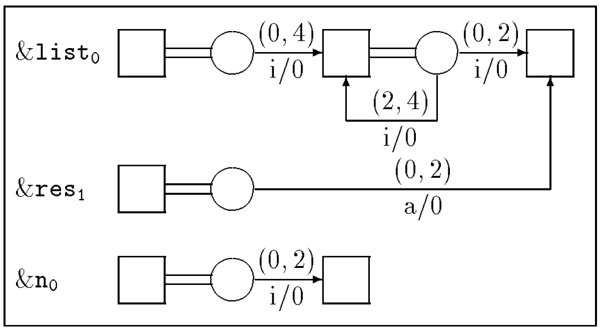
Source code

```
typedef struct list {
  int i;
  struct list *next;
} *list_ptr;

list_ptr nth_ele(list_ptr list$_0$, int n$_0$)
{
  int res$_1$, c$_1$, c$_2$, c$_3$;
  list_ptr ptr$_1$, ptr$_2$, ptr$_3$, ptr$_4$;
  ptr$_1$ = list$_0$;
  c$_1$ = 1;
  if (n$_0$ > 1) {
    do {
      ptr$_3$ = $\phi$(ptr$_1$, ptr$_2$);
      c$_3$ = $\phi$(c$_1$, c$_2$);
      ptr$_2$ = *(ptr$_3$.next);
      c$_2$ = c$_3$ + 1;
    } while (c$_2$ < n$_0$)
  }
  c$_4$ = $\phi$(c$_1$, c$_2$);
  ptr$_4$ = $\phi$(ptr$_1$, ptr$_2$);
  res$_1$ = *ptr$_4$.i;
  return res$_1$;
}
```

Source code in SSA form

Figure 3.60: Sample program and program in SSA form





Non-reduced function interface graph

Reduced function interface graph

Figure 3.61: Non-reduced function interface graph and reduced function interface graph

variables that might be used to transfer data between the calling and the called function have to be matched. Basically, there are only three different ways to transfer data between functions:

Type I: using the same global variables in the calling and the called function

Type II: passing a value from the calling to the called function as a parameter

Type III: returning a value from the called function to the calling function

Every other alternative to transfer data between functions is based at least on one of these concepts, and so they do not have to be treated explicitly.

One might wonder whether functions which communicate by storing data to and later loading data from a certain memory location through pointers have to be treated as a fourth case. However this is not true, since this is only possible if both functions have access to the same pointer value. The precondition for having the same pointer value in different functions is that the pointer value has been passed from one function to the other. This can only be done by using one of the three ways given above or by using a pointer to this pointer. In the last case this would lead back to the initial problem of passing a pointer between functions with the only difference of an increased pointer dereference level.

When the value nodes corresponding to the values that could have been used for interprocedural data transfer are matched, not only the value nodes themselves but every other node reachable from one of these nodes is included into the matching process. Therefore it is sufficient to use the value nodes described above as the basis for the matching process and to follow the paths along the dereference operations as far as possible. As described in section 3.3.3.1, the reduced function interface graph already only contains those nodes that can be reached from a node corresponding to either a global or an input or output variable of a function. These variables are exactly those that can be used by one of the three data transfer methods that have been described above.

The intraprocedural analysis that has been used to generate the function interface graphs annotated to the different CFG nodes did not use shared value and reference nodes to represent global data. This means that multiple value and reference nodes exist which represent the same global variables within different function interface graphs. Furthermore, the value and reference nodes representing the parameters of functions and those representing the values passed to the functions are different as well. Given a certain function call this leads to three different kinds of node pairs that have to be matched.

Type I: value nodes representing the valid global variables in the calling function and the value nodes representing the corresponding globals variables valid when entering and leaving the function[52].

Type II: value nodes representing the formal parameters of the called function and the value nodes representing the corresponding values passed to the function.

Type III: the value nodes representing the return values of the called functions and the value nodes representing the variables to which the functions results are assigned.

These three kinds of node pairs correspond to the previously described three data transfer methods. The initial pairs containing the nodes that have to be matched will now be formally defined.

[52]In case there are no nodes representing the corresponding variables yet, they will have to be created.

Definition 67 (initial value node pairs to be matched)

$$
\begin{array}{l}
\mathit{func\_unite\_nodes} \;:\; \mathit{CALL} \times \mathit{FIG} \times \mathit{FUNC} \times \mathit{FIG} \nrightarrow \mathbb{P}(\mathit{VN} \times \mathit{VN}) \\
\mathit{func\_unite\_nodes}(c, \mathit{fig}_c, f, \mathit{fig}_f) = \{(\mathit{vn}_c, \mathit{vn}_f) \mid \exists (v_c, v_f \in \mathit{VAR}_{SSA}) \;:
\end{array}
$$

$$
\begin{array}{ll}
((v_c \in \mathit{vars}_{\mathit{fig}_c}(\mathit{vn}_c)) \;\wedge\; (v_f \in \mathit{vars}_{\mathit{fig}_f}(\mathit{vn}_f)) \;\wedge & \\
\quad (((\mathit{base\_var}(v_c) = \mathit{base\_var}(v_f)) \;\wedge\; \mathit{is\_valid}(v_c, c) \;\wedge & \\
\qquad ((\mathit{ssa\_val}(v_f) = 0) \;\vee\; \mathit{is\_final}(v_f, f)) \;\wedge\; \mathit{is\_global}(v_f)) \;\vee & \text{(I)} \\
\quad (\exists (n \in \mathbb{N}) \;:\; ((\mathit{param}(f, n) = v_f) \;\wedge\; (\mathit{fig}_c = \mathit{fig}_f) \;\wedge & \\
\qquad\qquad (v_c = v_f))))) \;\vee & \text{(IV)} \\
(\exists (\mathit{vn}'_c, \mathit{vn}'_f \in \mathit{VN}) \;: & \\
\quad (v_c \in \mathit{vars}_{\mathit{fig}_c}(\mathit{vn}'_c)) \;\wedge\; (\mathit{deref}(\mathit{vn}'_c, (0, \mathbf{sizeof}(v_c))) = \mathit{vn}_c) \;\wedge & \\
\quad (v_f \in \mathit{vars}_{\mathit{fig}_f}(\mathit{vn}'_f)) \;\wedge\; (\mathit{deref}(\mathit{vn}'_f, (0, \mathbf{sizeof}(v_f))) = \mathit{vn}_f) \;\wedge & \\
\quad ((\exists (n \in \mathbb{N}) \;:\; ((\mathit{param}(c, n) = v_c) \;\wedge\; (\mathit{param}(f, n) = v_f))) \;\vee & \text{(II)} \\
\quad\; ((v_c \in \mathit{result}(c)) \;\wedge\; (v_f \in \mathit{result}(f)))))) & \text{(III)}
\end{array}
$$

In the definition given above the parts which belong to the different matching node types can be identified by their number. However, there is a fourth component contained in the definition. The node pairs specified by this fourth component have to be matched to handle recursive (direct and indirect) functions correctly as well. When a recursive function occurs, this leads to a matching process where the function interface graph belonging to the called function and the function interface graph belonging to the calling function are the same. In this case, there may be different value nodes which represent the function parameters of the calling and the called function that have to be matched. This has to be done because the algorithm does not distinguish between the different calling instances of recursive functions. Anyway it is not possible to distinguish between all possible calling instances since a recursive function may arbitrarily often call itself. Therefore at most an approximation could be made by computing only a certain number of recursive calling instances. In any case different calling instances remain that are not distinguished by the algorithm. The additional precision gained by distinguishing between some of the calling instances does not justify the costs of computing such an approximation, and hence all calling instances of recursive functions are treated as one.

A closer look at the definition of the nodes to be matched shows that the nodes belonging to the first and fourth group are nodes which represent the address of a variable, whereas the nodes of the second and third group represent the variables values. This difference is caused by the effects that are handled by these groups. The first and fourth group both specify nodes which represent the same variable within different graphs and so they have to be merged completely. The second and third group handle the effects of different variables with different addresses but which hold the same value. In the latter case it is sufficient to merge the value nodes which represent the variables values and leave the value nodes which represent the variables address untouched.

Now, after the nodes that have to be matched have been determined, the algorithm matching the function interface graphs of the calling and the called function can easily be specified. Since the matching process of two function interface graphs belonging to different CFG nodes is quite similar to the matching process for function interface graphs belonging to a calling and a called function, it is not surprising that some of the essential parts of the algorithm merging the function interface graphs for different CFG nodes can be reused. The algorithms used to merge function interface graphs belonging to different CFG nodes have

```
(1) function $call\_fig\_match(c, fig_c, f, fig_f)$
(2) {
(3)     $fig_{res} = new\_fig()$;
(4)     $unite\_nodes = func\_unite\_nodes(c, fig_c, f, fig_f)$;
(5)     $gen\_nodes\_and\_links(fig_c, fig_f, fig_{res}, unite\_nodes)$;
(6)     $gen\_ref\_edges(fig_c, fig_f, fig_{res})$;
(7)     return $fig_{res}$;
(8) }
```

Figure 3.62: Function matching the function interface graphs belonging to a function and a function called by this function

been described in section 3.2.4. Hence the functions that have been reused can be found there and are not shown repeatedly here. In fact the function shown in figure 3.47 is the only function that needs a modification to adapt the algorithm to merge the function interface graphs belonging to the calling and the called function. Figure 3.62 shows the modified version of this function. Besides the fact that some variables have been renamed and that the function has two additional parameters (the function call and the corresponding called function) there is only one line that has been altered significantly (line 4). Now, the nodes that have to be united are computed by the *func_unite_nodes* function (c.f. Def. 67). There are no further changes necessary to the functions merging the function interface graphs.

An example

Figure 3.63 shows how function calls will be handled. It contains a function calling another function, the reduced function interface graphs corresponding to the two functions and the resulting function interface graph representing both functions.

This example contains a new notation that will be used from here on. Whenever it is necessary to distinguish between variables which have the same name, the name of the function containing the variable is given as well. E.g. this means that `f.a` denotes the variable `a` which is used within function `f`[53].

As can be seen, the function `g` modifies the value pointed to by its parameter `b` and returns the value of this parameter afterwards. This results in the fact that `f.c` and `f.b` hold the same pointer value after the function call has been executed. Further, the value of $*$`f.b` and `f.a` is the same due to the assignment contained in `g`. This all is properly represented in the resulting function interface graph since the value nodes representing `f.c` and `f.b` as well as `f.a` and $*$`f.b` are the same.

3.3.3.3 Call path insensitive analysis

Now, that it is clear how the merging of the function interface graphs for a single function call can be handled, it is about time to think about the problem of merging the function interface graphs for all the possible function calls that have been and will be detected. Since there may be function calls that have not yet been detected when the merging process starts one has to deal with the function calls known right from the start as well as with those detected during the merging process.

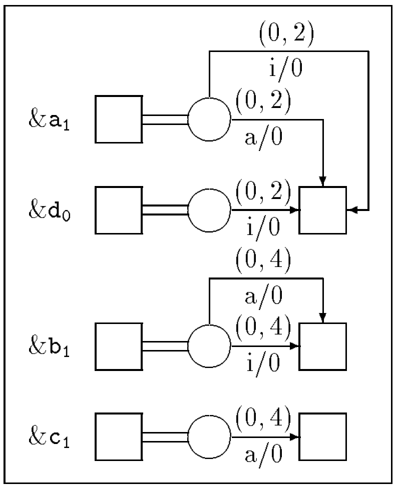
[53]Of course this is not sufficient to identify a variable in any case since there may be variables which have the same name that are defined in the same function within different scopes. However, since such things do not occur in the example programs the variables can properly be identified using this notation.

```
int d;

void f()
{
  int a, *b, *c;
  b = malloc(sizeof(int));
  a = d;
  c = g(a, b);
}

int *g(int a,  int *b)
{
  *b = a;
  return b;
}
```

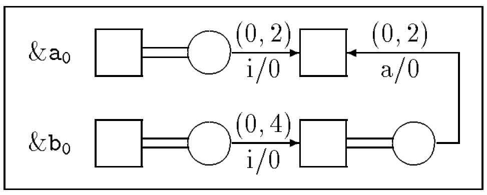
Source code

```
int d_0;

void f()
{
  int a_1, *b_1, *c_1;
  b_1 = malloc(sizeof(int));
  a_1 = d_0;
  c_1 = g(a_1, b_1);
}

int *g(int a_0,  int *b_0)
{
  *b_0 = a_0;
  return b_0;
}
```

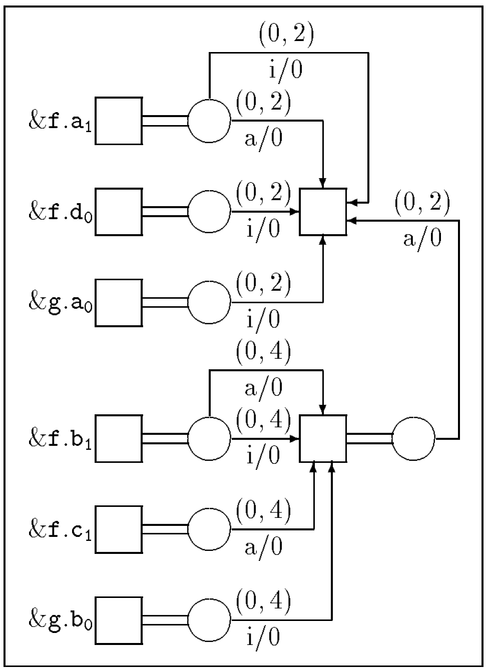
Source code in SSA form



fig$_f$



fig$_g$



fig$_{f\to g}$

Figure 3.63: Sample program, program in SSA form, corresponding (reduced) function interface graphs and resulting function interface graph demonstrating the merging process for function calls

```
(1)  function $update\_refs(cg, old\_fig, new\_fig)$
(2)  {
(3)    for all $(f \in cg.N)$ {
(4)      if $(f.fig = old\_fig)$ $f.fig = f.new\_fig$;
(5)    }
(6)  }

(7)  function $match(cg, unprocessed\_edges)$
(8)  {
(9)    while $(\exists((f_1, f_2, n) \in unprocessed\_edges))$ {
(10)     $fig = call\_fig\_match(nth\_call(f_1, n), f_1.fig, f_2, f_2.fig)$;
(11)     $update\_refs(cg, f_1.fig, fig)$;
(12)     $update\_refs(cg, f_2.fig, fig)$;
(13)     $ptr\_call\_edges = search\_new\_edges(cg, fig)$;
(14)     $add\_edges(cg, ptr\_call\_edges)$;
(15)     $unprocessed\_edges = (unprocessed\_edges \cup ptr\_call\_edges) - \{(f_1, f_2, n)\}$;
(16)   }
(17) }

(18) function $call\_path\_insensitive\_match(cg)$
(19) {
(20)   $match(cg, cg.E)$;
(21) }
```

Figure 3.64: Call path insensitive analysis algorithm

The easiest way to do this is to process the edges of the call graph one by one and to combine the function interface graphs belonging to the calling and the called function until all edges have been processed. This is a very cheap as well as simple strategy. Figure 3.64 shows the corresponding algorithm. The only input of the function *call_path_insensitive_match* is the call graph. It is assumed that the previously computed and reduced function interface graphs have been attached to the nodes of the call graph and can hence be accessed as well. The algorithm uses one set (*unprocessed_edges*) which contains the edges of the call graph that have not yet been processed. At the beginning the set of edges to be processed contains all edges of the initial call graph (20). The merging process starts by choosing an arbitrary call graph edge to be processed as next (9). Then the function interface graphs belonging to the calling and the called function are merged (10). Now, all references to one of the function interface graphs that have been merged have to be updated to refer to the newly created merged function interface graph (3,4,11,12). Due to the merging of the function interface graphs there may be new functions found that could be called by function calls using function pointers. This may lead to new edges (13) which are added to the call graph (14) as well as to the set of edges to be processed (15). These new edges are computed by the function *search_new_edges* which only checks if there are new aliases that have been found for the function pointers used in function calls. Now, the currently processed edge can be removed from the set of edges to be processed (15). After all edges of the call graph have been processed, all call graph nodes hold a reference to the resulting function interface graph[54].

As can be seen from the algorithm there are no requirements on the order in which the

[54]Otherwise the program contains non-called and hence useless functions.

```
                    void g()            void h()
                    {                   {
void f()              int *x1, *x2;       int *y1, *y2;
{                     :                   :
  :                   k(x1, x1, x2);      k(y1, y2, y2);
  g();                :                   :
  :                 }                   }
  h();
  :                 void k(int *p1, int *p2, int *p3)
}                   {
                      :
                    }
```

Source code

$$\begin{tikzcd}
 & \mathtt{f} \arrow[dl] \arrow[dr] & \\
\mathtt{g} \arrow[dr] & & \mathtt{h} \arrow[dl] \\
 & \mathtt{k} &
\end{tikzcd}$$

Call graph

Figure 3.65: Imprecision caused by a call path insensitive analysis

call graph edges have to be processed. Even recursive functions do not need any special treatment. The updating of the references from the call graph nodes to the corresponding function interface graphs is not very efficient. Although it could be improved by using an union search algorithm as described in section 3.2.4 this is not done here. This is done to keep the algorithm as simple as possible and to avoid to have to explain the same algorithm twice[55].

3.3.3.4 Call path sensitive analysis

Although the call path insensitive algorithm works quite well in many situations, there is one significant disadvantage when this algorithm is used: if a function is called by more than a single function call, it is not distinguished between the different calling contexts. This can lead to unnecessary imprecision as shown in the example displayed in figure 3.65. Here the function `k` is called by `g` and `h`. In the first case this leads to an alias between ∗`p1` and ∗`p2`, whereas an alias between ∗`p2` and ∗`p3` will occur in the second case. This means that the values of ∗`p1`, ∗`p2` and ∗`p3` are all represented by the same value node after the corresponding function interface graphs have been merged. However, this leads to a possible alias between ∗`p1` and ∗`p3` that cannot really occur.

To avoid the imprecision caused by the call path insensitive algorithm, a call path sensitive algorithm can be used instead. In this case a different strategy has to be used when the function interface graphs have to be merged. The unnecessary detected possible alias between ∗`p1` and ∗`p3` can be avoided if the function interface graph belonging to `k` is duplicated. When the function calls calling `k` are processed, each call processes a different copy of the function interface graph and hence there are different value nodes representing the values of ∗`p1`, ∗`p2` and ∗`p3` for each function call calling `k`. In this simple case the same effect could have been achieved if the function `k` had been duplicated before the analysis starts and the call path insensitive algorithm had been used afterwards. Figure 3.66 shows the corresponding transformed program. The only difference between duplicating the function interface graph instead of the program code is that the first is more efficient and hence should be preferred. In some cases (e.g. if `k` contains further function calls) the duplication of functions becomes even less efficient because the number of edges contained in the call

[55]Of course the faster algorithm is used by the implementation.

```
                    void g()           void h()
                    {                  {
                      int *x1, *x2;      int *y1, *y2;
                      ⋮                  ⋮
void f()              k_g(x1, x1, x2);   k_h(y1, y2, y2);
{                     ⋮                  ⋮
  ⋮                 }                  }
  g();
  ⋮                 void k_g(int *p1, int *p2, int *p3)
  h();              {
  ⋮                   ⋮
}                   }

                    void k_h(int *p1, int *p2, int *p3)
                    {
                      ⋮
                    }
```

Source code

$$\begin{tikzcd}
 & \mathtt{f} \arrow[dl] \arrow[dr] & \\
\mathtt{g} \arrow[d] & & \mathtt{h} \arrow[d] \\
\mathtt{k_g} & & \mathtt{k_h}
\end{tikzcd}$$

Call graph

Figure 3.66: Call path sensitive analysis

graph and hence the number of merging processes would then increase as well. Since copying functions instead of function interface graphs is obviously significantly less efficient, the function interface graphs instead of the functions are copied when necessary. Nevertheless, it is sometimes useful to keep the alternative method in mind since both methods have some common properties.

Although copying function interface graphs is already much more efficient than copying the functions itself, the additional precision gained by the call path sensitive analysis cannot be achieved without additional costs. These time and space costs are caused by the duplication of the function interface graphs. In our example this does not make a big difference (only the function interface graph belonging to `k` has to be duplicated once), but there may be call graphs which lead to larger amounts of copies to be generated. The call graph shown in figure 3.67 is an example where the number of function interface graphs to be copied increases linearly to the number of functions (or function calls) occurring in the call graph: here the function interface graphs of all functions contained in the call graph (besides f_1) have to be copied since every function is called by two different function calls (besides f_1). However, no matter how the call graph will be, the number of function interface graphs that have to be copied is in any case smaller than the number of edges contained in the call graph.

Beside the call graph discussed above the figure contains a second (expanded) call graph which would have been built if the functions instead of the function interface graphs had been copied. As can be seen, the number of call graph nodes as well as call graph edges increases quadratically in this case, which again shows the significant advantage when function interface graphs instead of the functions themselves are duplicated.

So far, the duplication of the function interface graphs and the corresponding additional time and space costs form no big difference between the call path insensitive algorithm and the call path sensitive algorithm. However, there is an additional problem that has not been addressed up to here: if a function interface graph is duplicated before all the function calls

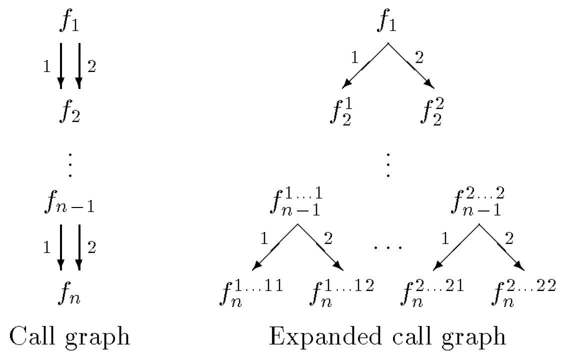


Figure 3.67: Call graph causing many duplication operations

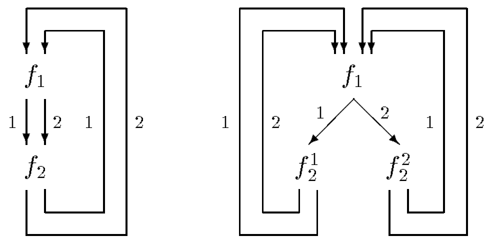


Figure 3.68: Cyclic call graph needing additional matching operations when the function interface graphs are duplicated and corresponding expanded graph

contained in the called function have been processed, additional matching operations might be needed. As stated before, the order in which the different function interface graphs are matched does not play any role for the call path insensitive algorithm. This changes if a call path sensitive algorithm is used instead. Looking at the example from figure 3.67 this means that, if a top down processing order is chosen, the function interface graph belonging to f_2 will then be copied first. Since the function interface graph belonging to f_2 has not yet been matched with the graph belonging to f_3, this matching process would have to be performed for both copies of the graph belonging to f_2. Altogether this would finally lead to as many matching operations as if the algorithm duplicating the functions had been chosen. However, this effect can be avoided if the merging process is performed in a bottom up order. In this case, the function interface graph belonging to f_n is duplicated first, and hence both copies can be merged with the graph belonging to f_{n-1} and so on. Using this strategy will prevent the introduction of additional matching operations.

The precondition to avoid additional matching operations is that there are so far no unprocessed function calls if the function interface graph of a function is to be duplicated. Unfortunately, call graphs do exist where none of the possible processing orders assures that no additional matching operations are necessary. This is either caused by recursive functions or by functions containing function calls using function pointers. In these cases there are two choices: to at least accept some additional matching operations or not to duplicate some of the function interface graphs and hence to accept precision losses.

The following three examples explain which kind of problems may occur. In the first case (figure 3.68) the duplication of one of the function interface graphs requires additional matching operations no matter which order is chosen. In the second case (figure 3.69) it is at least possible that an additional matching operation is necessary. Even if all the function calls of g_1 have been processed it might be that another possible function call exists calling

$$\begin{tikzcd}
f_1 \arrow[dr, "1"'] & & f_2 \arrow[dl, "2"] & & f_1 \arrow[d, "1"'] & & f_2 \arrow[d, "2"] \\
& g_1 \arrow[dl, "1*"'] \arrow[dr, "*n"] & & & g_1^1 \arrow[d, "1*"'] \arrow[drr, "*n"] & & g_1^2 \arrow[dll, "1*"'] \arrow[d, "*n"] \\
h_1 & \cdots & h_n & & h_1 & \cdots & h_n
\end{tikzcd}$$

Figure 3.69: Call graph with functions containing function calls using function pointers

a function h_{n+1} that has not been detected so far. In this case both copies of the duplicated function interface graph belonging to g_1 have to be matched with the function interface graph belonging to h_{n+1} then. The problems shown by the second example may arise if a new function is found that may be called by a given function. However, the contrary can cause problems as well: if a new function is found that calls a given function it might be that the function interface graph belonging to the called function has already been merged with the function interface graph of another calling function before. For this example it is assumed that all possible function calls of g_1 have been processed and the resulting call graph has been duplicated and merged with f_1 and f_2. If there is a new function call calling g_1 found within a function f_3 now, it is not possible to duplicate the function interface graph representing g_1 and the functions called by g_1, since both copies of this graph have already been merged with f_1 and f_2. Of course one could choose one of these duplicates, but this causes the same imprecision that is caused by the call path insensitive algorithm as well.

As can be seen from these examples, the processing order can significantly influence the costs of the call path sensitive algorithm. Choosing an awkward processing order could lead to many unnecessary matching operations. To avoid this and the resulting unnecessary time and space costs, the algorithm has to choose a suitable processing order. Nevertheless this does not solve the problems caused by function calls using function pointers and recursive functions. As long as the program contains neither of these, the problem can simply be solved by processing the call graph in a bottom up order. In this case the call graph is acyclic and no further edges are added to the initial graph.

However, recursive functions and function calls using function pointers have to be dealt with somehow as well. This problem can be addressed by processing the edges of the call graph within two passes. During the first pass only those edges that are not influenced by the problems caused by function calls using function pointers and recursive functions are processed. Within this pass the function interface graphs are duplicated whenever necessary. During the second pass the remaining edges are processed by the call path insensitive algorithm. This avoids additional matching operations in any case. Most of the call graphs of usual C programs are processed completely, or at least almost completely, within the first pass. Only those programs which use many recursive functions or function pointers need the second pass and hence produce slightly less precise results. The algorithm combines the benefits of the call path sensitive algorithm copying the functions and the call path insensitive algorithm. On the one hand, the number of matching processes is the same as if the call path insensitive algorithm had been used, and so there are only the comparably small additional costs of copying the function interface graphs. On the other hand, the precision of the results is equal or comes close to the call path sensitive algorithm copying the functions. The details of the algorithm are described next.

The algorithm performing the call path sensitive analysis is shown in figure 3.70. It uses

```
function $call\_path\_sensitive\_match(cg)$
{
  for all $(f \in cg.N)$ {
    if $(ptr\_call(f))$ $f.out\_no = \infty$;
    else $f.out\_no = 0$;
    if $(addr\_taken(f))$ $f.in\_no = \infty$;
    else $f.in\_no = 0$;
  }
  for all $((f_1, f_2, n) \in cg.E)$ {
    if $(f_1.out\_no \neq \infty)$ $f_1.out\_no = f_1.out\_no + 1$;
    if $(f_2.in\_no \neq \infty)$ $f_2.in\_no = f_2.in\_no + 1$;
  }
  $done =$ false;
  $unprocessed\_edges = cg.E$;
  while (not $done$) {
    if $(\exists (f_1, f_2, n) \in unprocessed\_edges \mid f_2.in\_no = 1)$ {
      $fig = call\_fig\_match(nth\_call(f_1, n), f_1.fig, f_2, f_2.fig)$;
      $update\_refs(cg, f_1.fig, fig)$;
      $update\_refs(cg, f_2.fig, fig)$;
    } else if $(\exists (f_1, f_2, n) \in unprocessed\_edges \mid (f_2.out\_no = 0) \vee$
        $(f_1.fig = f_2.fig))$ {
      $fig = call\_fig\_match(nth\_call(f_1, n), f_1.fig, f_2, f_2.fig)$;
      $update\_refs(cg, f_1.fig, fig)$;
    } else $done =$ true;
    if (not $done$) {
      if $(f_1.out\_no \neq \infty)$ $f_1.out\_no = f_1.out\_no - 1$;
      if $(f_2.in\_no \neq \infty)$ $f_2.in\_no = f_2.in\_no - 1$;
      $ptr\_call\_edges = search\_new\_edges(cg, fig)$;
      $add\_edges(cg, ptr\_call\_edges)$;
      $unprocessed\_edges = (unprocessed\_edges \cup ptr\_call\_edges) - \{(f_1, f_2, n)\}$;
    }
  }
  $match(cg, unprocessed\_edges)$;
}
```

Figure 3.70: Call path sensitive analysis algorithm

two counters which count the unprocessed incoming and outgoing edges of every call graph node. At the beginning these counters have to be initialised (3-7). Since there may be additional edges that are added to the call graph during the computation, such edges have to be taken into account in advance. This is done by setting the outgoing edge counter of functions containing function calls using function pointers to infinity. This shall reflect the fact that there may be an arbitrary amount of outgoing edges that may be connected to this node during the analysis (4). The same is done for the incoming edge counter in case the functions address has been taken somewhere in the program (6). Now, all edges of the initial call graph are processed one by one and the incoming and outgoing edge counters of the calling and the called function are incremented as long as they have not been set to infinity (9-11). Like the call path insensitive algorithm, the call path sensitive algorithm uses a set containing the nodes that have not yet been processed (*unprocessed_edges*) which is initialised to contain all the edges of the initial call graph (14). After this has been done, the algorithm checks if a function call exists, where no other unprocessed function call calls the same function (16). In this case this function call can be securely processed (17), and the references to the corresponding function interface graphs can be replaced by the new merged graph (18-19). If no such function call can be found there are two further types of function calls that can be processed now: those function calls calling a function that does not contain any unprocessed function calls and those where the function interface graphs belonging to the calling and the called function have already been merged before and hence are identical (20). Note that according to the preceding unfulfilled condition at least one further unprocessed call exists which calls the same function in either case. In the first case this means that the function interface graph belonging to the called function has to be duplicated. This is done by computing the merged function interface graph (22) and exchanging only the references to the function interface graph of the calling function (23). In general this does not affect the function interface graph of the called function. Hence this graph can be matched with the function interface graphs belonging to the other functions calling this function as it is. The second case appears if functions are called recursively. The steps to be done then are exactly the same, although the intention is different. In this case the function interface graph of the calling and the called function are identical, and hence the graph is merged with itself. If none of the unprocessed function calls fulfil one of the preceding conditions, the first pass has to be ended (24), and the call path insensitive algorithm will have to deal with the remaining unprocessed function calls (33). Otherwise, some final steps finish the processing of the current function call (25). The most important thing to be done is to update the counters of the unprocessed incoming and outgoing edges as long as they have not been set to infinity (26-27). Finally, the currently processed edge has to be removed from the set containing the unprocessed edges, and it has to be checked whether new edges have been found that have to be processed (28-30). Like in the case of the call path insensitive algorithm every call graph node holds a reference to the final function interface graph when the algorithm terminates.

An example

The example shown in figure 3.71 contains an indirect recursive function call through the functions `nth_ele` and `shift`. This figure contains the source code, the corresponding code in SSA form as well as the call graph of the program. The function interface graphs which belong to these functions are shown in figure 3.72. After the function interface graphs have been merged it can be seen that the value returned by the function `nth_ele` can be any of the three values passed to the function. This is represented by the fact that the corresponding values are all represented by the same node. Although the possible alias between `*p` and `*z` can only occur after the functions have been called recursively twice, this effect can be detected without the need of repeated merging steps. This shows that recursive functions do not need recursive updates or merging processes unlike in many other algorithms (e.g. [WL95]). Hence there is no iteration of the merging process necessary until stabilisation is reached. Everything can be computed without having to check if there are changes requiring a further iteration.

3.3.4 Correctness

The correctness of the algorithm is almost clear since there are only comparably few differences between the merging of function interface graphs belonging to different functions and those belonging to different CFG nodes. The latter have already been shown to be correct in section 3.2.6. Since the only difference between these algorithms is the initial set of nodes to be matched, it is quite clear that as long as these sets are correct, the complete algorithm is correct as well. As discussed in section 3.3.3.2 this set consists exactly of those nodes that belong to the values that can be used to interchange data between functions. This ensures that any value that could be used to transfer data between functions is finally represented by the same value node, and hence all possible aliases can be detected.

3.3.5 Time and space bounds

Both time and space bounds of the algorithm are mainly influenced by two factors: the number of edges contained in the final call graph and the size of the function interface graphs belonging to the different functions. The time and space costs needed to merge two function interface graphs have sufficiently been discussed in section 3.2.7, and so it is not necessary to discuss them again here. The size of the function interface graphs belonging to each function grows at most linearly to the size of the corresponding function. Hence the size of a function interface graph resulting from a merging process of two function interface graphs belonging to two different functions can be at most the sum of the sizes of the two input graphs. In the absolute worst case, a merged function interface graph can reach a size growing linearly to the size of the complete program. However this is not very likely since this would mean that all the functions that have been involved in the merging process neither exchange any data nor access the same data. Assuming that a constant limiting of the number of different dereference operations applied to each variable exists this leads to a size of the function interface graphs growing linearly to the number of involved variables. Since real programs usually use the same variable a couple of times it seems more appropriate to assume a size which grows less fast. In the following it is assumed that E_{CG_p} is the set of edges of the call graph belonging to a program p, that $STMT_p$ is the set of statements contained in the program p and that VAR_{SSA_p} is the set of variables[56] used by the program p.

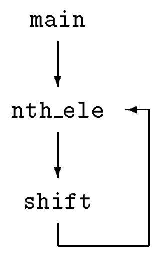
3.3.5.1 Time bounds

Given the assumptions made above, the time needed by the interprocedural analysis parts is in $O(|E_{CG_p}| \cdot |STMT_p|^2)$ in the worst case. Assuming that the size of the merged function interface graphs grows only linearly to the number of the involved variables and not to the size of the program, the worst case becomes $O(|E_{CG_p}| \cdot |VAR_{SSA_p}|^2)$. If the additional

[56]Since the program has been transformed into SSA form it can be assumed that VAR_{SSA_p} contains more variables than the original program had.

```
void main()
{
  int *p, *x, *y, *z, n;
  p = nth_ele(x, y, z, n);
}

int *nth_ele(int *x, int *y, int *z, int n)
{
  int *p, m;
  if (n > 1) {
    m = n - 1;
    p = shift(x, y, z, m);
  } else p = x;
  return p;
}

int *shift(int *x, int *y, int *z, int n)
{
  int *p;
  p = nth_ele(y, z, x, n);
  return p;
}
```

Source code



Call graph

```
void main()
{
  int *$p_1$, *$x_0$, *$y_0$, *$z_0$, $n_0$;
  $p_1$ = nth_ele($x_0$, $y_0$, $z_0$, $n_0$);
}

int *nth_ele(int *$x_0$, int *$y_0$, int *$z_0$, int $n_0$)
{
  int *$p_1$, *$p_2$, *$p_3$, $m_0$, $m_1$, $m_2$;
  if ($n_0$ > 1) {
    $m_1$ = $n_0$ - 1;
    $p_1$ = shift($x_0$, $y_0$, $z_0$, $m_1$);
  } else $p_2$ = $x_0$;
  $p_3$ = $\phi$($p_1$, $p_2$);
  $m_2$ = $\phi$($m_0$, $m_1$);
  return $p_3$;
}

int *shift(int *$x_0$, int *$y_0$, int *$z_0$, int $n_0$)
{
  int *$p_1$;
  $p_1$ = nth_ele($y_0$, $z_0$, $x_0$, $n_0$);
  return $p_1$;
}
```

Source code in SSA form

Figure 3.71: Sample program, corresponding program transformed into SSA form and call graph

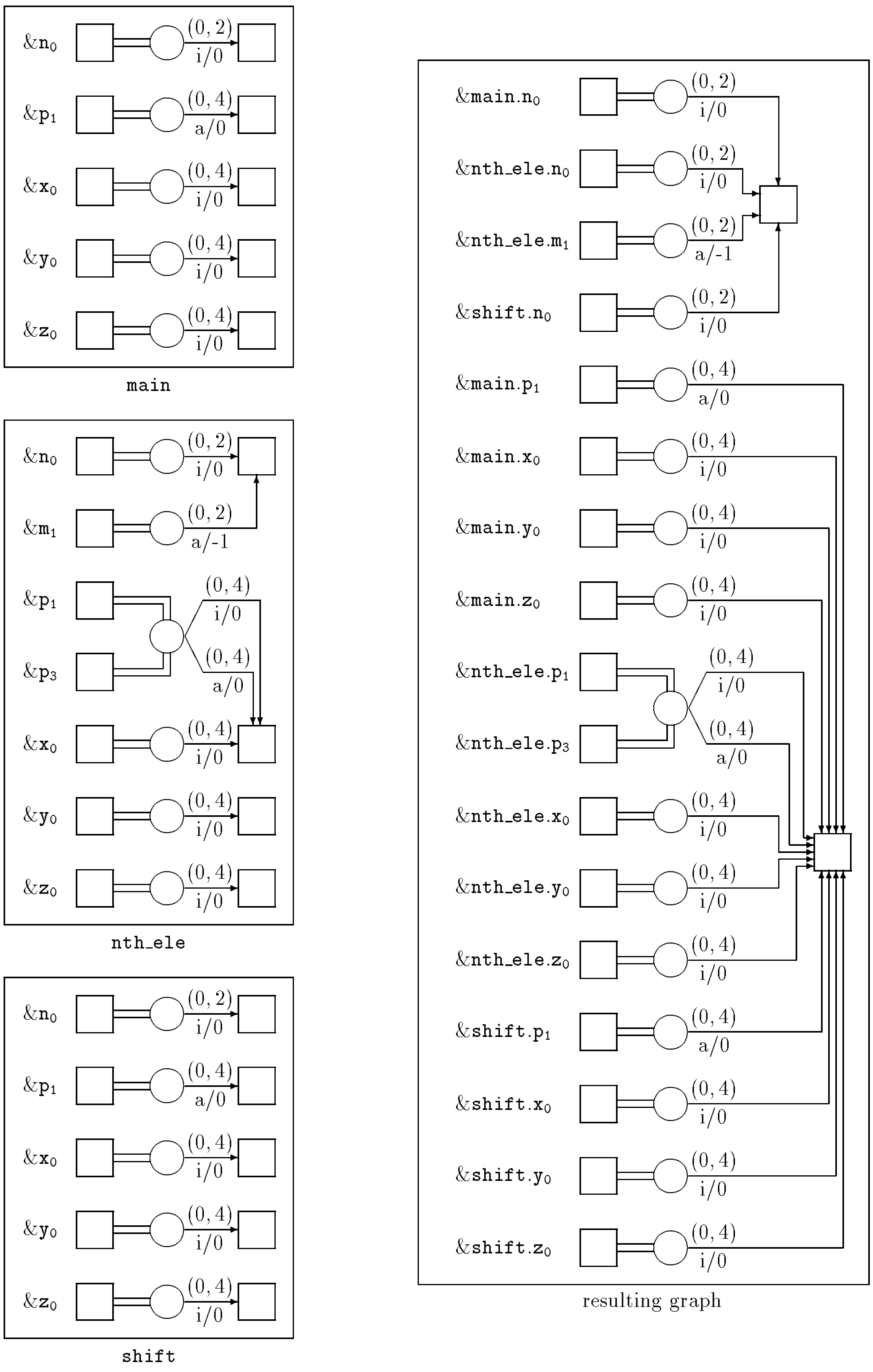


Figure 3.72: Reduced function interface graphs belonging to the different functions and resulting function interface graph

assumption is made that the merging process of two function interface graphs takes time growing linearly to the size of the function interface graphs to be merged, an overall time bound of $O(|E_{CG_p}| \cdot |VAR_{SSA_p}|)$ is reached.

Since the algorithm that has been implemented avoids the unnecessary copying of function interface graphs that have to be merged, the time needed to merge the graphs depends only on the number of nodes that will really be merged. When the function interface graphs belonging to a calling and a called function have to be merged, these nodes are those belonging to the functions parameters as well as those belonging to the global variables. Assuming that the number of value nodes that have to be merged per parameter / global variable is constant, the merging process takes time proportional to the number of involved parameters and global variables. If it is further assumed that the number of parameters is a small value not depending on the program size, the time needed to merge the two graphs grows with the number of global variables. Altogether, this results in an analysis time growing with the number of global variables and the size of the call graph ($O(|E_{CG_p}| \cdot |VAR_{global}|)$). As can be seen from the empirical results shown in section 4.5 this time bound seems to be much more appropriate in the case of real programs.

3.3.5.2 Space bounds

Since two function interface graphs that are merged will never produce a function interface graph bigger than the sum of the two function interface graphs to be merged, there is only one situation where new space is needed during the interprocedural analysis. This happens if a function interface graph has to be duplicated because there are different functions calling the function belonging to the function interface graph. In the worst case this can lead to $O(|E_{CG_p}|)$ duplicates of the function interface graphs that have to be produced. In this case the space needed for all the function interface graphs is in $O(|E_{CG_p}| \cdot |STMT_p|)$ if it is assumed that the size of the function interface graph grows linearly to the size of the program. In most cases significantly less space is needed since the merged function interface graphs are almost always smaller than the two graphs that have been merged. Besides that it is unlikely that many large function interface graphs have to be duplicated because of multiple incoming call graph edges.

The space bound described above is based on the fact that function interface graphs may be duplicated in some cases. If a purely call path insensitive analysis is performed instead, none of the function interface graphs will have to be duplicated, and hence a far better worst case space bound can be achieved. As mentioned above, merging function interface graphs at most reduces the overall size of all function interface graphs taken together. Hence the resulting function interface graph is not larger than the function interface graphs that have been merged taken together. Assuming that the overall size of the function interface graphs grows linearly to the program size this leads to a worst case space bound growing with the program size as well ($O(|STMT_p|)$). As can be seen in chapter 4.5 the space needed by the example programs confirms the above made assumptions.

Now the computation of a single function interface graph which represents all the effects a program can have, has been described. All the possible aliases that may occur during the program execution can be found by means of this graph. The results are precise in the sense that every possible alias can be found by inspecting either the final function interface graph or the non-reduced function interface graph of the function. The analysis is applicable to almost all C programs and not limited to C-subsets, which would make the analysis inappropriate for many real programs.

```
char *strcpy(char *d, char *s)
{
  int u;
  d[u] = s[u];
  return d;
}
```

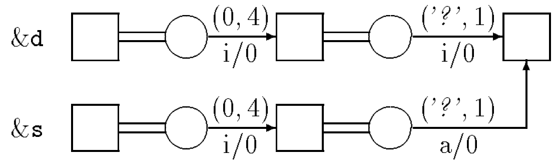


Figure 3.73: Function representing the effects of `strcpy` and the corresponding function interface graph

3.4 Handling C properties needing exceptional treatment

Although almost all the commonly used properties of the C programming language have been taken into account by the algorithm, a few things remain that have not been discussed so far. Since the handling of these properties in the preceding sections would have resulted in a lot of exceptional rules they have been ignored up to here to keep the algorithms simple and easier to understand. Nevertheless it is possible to deal with the effects of these properties as well, and so it will be shortly described in the following how this can be done.

3.4.1 Handling library functions

Library functions are typically used by other programs which do not have the corresponding source code at hand. This results in the problem that no function interface graph can be built for these functions, and hence the functions effects are not taken into consideration properly. This problem can be solved by either trying to get the corresponding source codes and integrating them into the analysis or by providing some functions which have the same effect on the function interface graphs. Since the first possibility is not very efficient, and sometimes even impossible, the second possibility is the one of interest here. Usually the user of a library function knows the functions effects and can hence provide a simple function which has the same effects on the function interface graph. This function interface graph can then be integrated into the analysis instead of the original library function. This means that a function that does not modify any data like `remove` or `abort` can be represented by a function with an empty body[57], whereas a function like `strcpy` has to be replaced by a non-empty function. Here the major difference is that the `strcpy` function modifies some of the values reachable by the parameters passed to the function. Figure 3.73 shows a function which produces a suitable function interface graph. Since the unknown index `u` produces an unknown offset, the resulting function interface graph represents the fact that the data that could previously be accessed by `s[n]` can be accessed by `d[n]` after the function was executed. Of course the `strcpy` function could as well be represented by a function with an empty body if it is assumed that the copied string will never be converted into a pointer, which is in fact not very likely. However, there are other functions (e.g. `memcpy`) where it is much more likely that the copied values contain pointers or are converted into pointers. In such cases it is essential to have a function which represents the effects of the library function (like the one shown above) to keep the analysis precise.

[57]Alternatively, the definitions of these functions could simply be omitted. In this case the algorithm would ignore calls of these functions since there is no corresponding function interface graph available then.

```
void func(float r, int n, ...)
{
  va_list args;
  int x;
  ⋮
  va_start(args, n);
  ⋮
  x = va_arg(args, int);
  ⋮
  va_end(args);
}
```

Figure 3.74: Function having a variable argument list

```
#define va_start(args, fmt) args = tmp_var_args
#define va_arg(args, type) *((type *) &args)
#define va_end(args)
```

Figure 3.75: Macro definitions

3.4.2 Variable parameter lists

Variable parameter lists are used when a function has to deal with a different number of parameters for different calls of the function. Some of the most commonly used functions using variable parameter lists are the functions belonging to the `printf` group. Usually calls to these functions do not influence the possible aliases that can be produced by a program, and so one could think about representing them by a function which has an empty body (as described in section 3.4.1). However, in some cases programs contain user defined functions using variable parameter lists which have to be dealt with correctly. Besides which it is possible to misuse the `sprintf` function to copy values and hence produce aliases.

A typical function with a variable parameter list is shown in figure 3.74. All values that are not passed to the function as fixed parameters are accessed using the function `va_arg`. Since it is not possible to distinguish between the different variable parameters, all the variable parameters passed to the function have to be matched with those variables to which the value of the `va_arg` macro was assigned. Since macros are expanded in one of the earliest processing steps during the analysis they are not represented in the SUIF code. Therefore it is not easy to find out which assignments have been influenced by the macros and which have not. The problem can be solved by redefining the macros as shown in figure 3.75. These macro definitions make sure that every value that is accessed using the `va_arg` function is matched with the variable `tmp_var_args`. Of course, a corresponding declaration of this variable has to be added to every function having a variable parameter list then. The advantage of these macros is that now all variables that are assigned any of the variable arguments are automatically matched with the variable `tmp_var_args`. The only thing that remains to be done now is to match all variable parameters with this variable. This then assures that every variable parameter has been matched with its corresponding variable in the function.

```
void f()
{
  jmp_buf env;
  switch (setjmp(env)) {
    case 0:
/* normal function execution */
      ⋮
      g(&env);
      ⋮
      break;
    case 1:
/* error handling:  escaped from g */
      ⋮
      break;
    case 2:
/* error handling:  escaped from h */
      ⋮
      break;
  }
}
```

```
void g(jmp_buf *env)
{
  ⋮
  h(env);
  ⋮
/* jump to f in case of an error */
  if (...) longjmp(*env, 1);
  ⋮
}


void h(jmp_buf *env)
{
  ⋮
/* jump to f in case of an error */
  if (...) longjmp(*env, 2);
  ⋮
}
```

Figure 3.76: Example program using `longjmp` and `setjmp`

3.4.3 The `longjmp` and `setjmp` functions

Another aspect that has not been taken into account so far, are the effects of the `longjmp` and `setjmp` functions. As described by the ANSI standard ([Ame89a]) the `setjmp` function can be used to mark a certain location inside a function to return to this location with a single jump using the `longjmp` function. The call of the `longjmp` function can be located within any function called directly or indirectly by the function containing the corresponding `setjmp` statement. When the `setjmp` function is executed, it stores the current environment information to a structure and returns null. Later when a `longjmp` statement causes a jump back to the location of the `setjmp` function, this function returns the value that was used at the corresponding call of the `longjmp` function.

The `longjmp` function takes the environment information that has previously been stored by the `setjmp` function and passes a non-null value back to the `setjmp` function. If there are multiple `longjmp` functions contained in a program, each of them passing a different value to the `setjmp` function, these values can be used to identify the `longjmp` function that caused the jump back to the `setjmp` function.

The execution of a `longjmp` function does not affect the values of global or static variables. However, the local variables of a function containing a corresponding `setjmp` function are undefined after a jump back to this function has been performed. Since the functions containing the `longjmp` function could have modified global variables that could then be used by the function containing the `setjmp` function this is another form of communication between functions that cannot be handled by the alias analysis algorithm so far.

To make things clearer there is a simple example program shown in figure 3.76. This is one of the typical situations where these functions could be used. Function `f` calls `g` which then calls `h`. If an error occurs within `g` or `h`, the execution of the function terminates, and the error handling can be done by the function `f`. This is particularly useful if many errors may occur during the execution of `g` and `h` that require that the following code is not executed in this case. E.g. if a file cannot be opened or there is no memory left to allocate, the statements accessing the file or the memory should not be executed. Of course it is possible to have the same effect using nested **if** statements or error variables. This however, produces a huge amount of nested **if** statements in the one case, or a huge amount of **if** statements repeatedly testing the error variable in the other case. Here it depends on the programmer whether he prefers to keep the code as small as possible, which then results in unstructured jumps, or whether he prefers to add some more error handling code to the functions keeping them easier to understand. In any case, there may be programmers using these functions, and so the effects of these functions should be handled somehow.

To handle the effects of such unstructured jumps correctly it is necessary to extend the call graph and to add some special edges representing these jumps. These edges can then represent the possible jumps from a function containing a `longjmp` statement to a function containing a `setjmp` statement. The `longjmp` and `setjmp` function communicate through the `jump_def` structures. If the structure used by a `longjmp` function may be aliased by a structure used by a `setjmp` function, a new edge has to be added to the call graph. This is quite similar to the function pointer concept. However, the handling of the edges representing such jumps has to be different from the handling of usual functions. The value passed from the `longjmp` function to the `setjmp` function can be handled in a similar way to function parameters. However, it has to be taken into account that the parameter passed to the `longjmp` function is returned by the `setjmp` function. Further differences are that the matching of the global variables has to be modified as well. Usually, the global variables used inside a function that are matched with the global variables of the calling function are those valid at the beginning or at the end of a function. Since the `setjmp` function can occur anywhere in the code, the actual SSA values of global variables can be different in this case. Hence this has to be taken into account when such jumps are being processed. This is done by matching the global variables with the SSA values valid at the position of the `setjmp` function instead of those valid at the beginning or at the end of the function.

As was shown in this section even those properties of the C programming language that are rarely used can be handled by our alias analysis algorithm. Even comparably complicated problems like the jumps using the `longjmp` function can be solved without causing too much problems since they can be fitted into the algorithm quite naturally. Altogether this proves that our algorithm should be applicable to almost all real C programs which was one of our primary aims.

Chapter 4

Tests, implementation and empirical results

The alias analysis algorithm presented in chapter 3 has been implemented and tested on an UNIX system. The tests were performed on a two processor Ultra 2 SPARC Creator 3D workstation with 512 MB of RAM. Within this chapter the test suite used to check the algorithm will be presented. Furthermore, the time and space requirements of the algorithms different passes will be discussed and compared with the theoretical time and space bounds found earlier. All reported execution times are user CPU times that have been computed by the C library function `getrusage`. Since the values returned by this function show slight differences for different runs, the mean values of three runs have been computed. As mentioned earlier the different passes have been implemented separately. Therefore every pass has to load its data from a file before its execution, and has to save the modified data to a file afterwards. Since the times needed to load and save these files outrun the times needed by the algorithm itself in many cases, they have not been taken into account for the time measurement. Anyway, this overhead could have been avoided easily by simply combining the different passes to a single program.

4.1 The test suite

The test suite consists of 14 C programs. An overview of these programs and some of their properties is given in the table shown in figure 4.1. The table contains the sizes of the programs source code in bytes as well as the number of source code lines. However, these two values are comparably vague parameters when it comes to judging the programs size. Most of the time and space bounds of our algorithms depend on the number of statements contained in a program or a function. If one only looks at the two values mentioned above, it is not possible to derive the number of statements from them with a tolerable precision. Therefore the number of tree nodes generated by the SUIF compiler is given in the table as well. Unlike the source code size, they yield a much better measure for the number of statements contained in the program.

As can be seen from the table the number of nodes comes close to the number of source code lines in most cases. However, there are at least two programs where a significant difference occurs: the programs 'banner' and 'factor'. Both programs contain a huge array used to store a sequence of integer values. In the first case these values define the bitmaps of the characters to be printed, whereas they contain all primes up to a certain value in the second case. Since the definitions of these arrays consumes comparably large parts of the

program	nodes	size	lines	functions	variables		
					global	local	
						static	non-st.
	n_i	s [$bytes$]	l	f	v_g	v_{ls}	v_l
queens	48	526	39	3	6	0	8
quicksort	59	1 051	65	6	3	0	16
factor	111	49 446	827	3	27	0	14
wf1	113	2 044	101	7	6	0	25
banner	230	74 710	1 166	1	51	0	22
cal	390	12 688	479	12	85	1	102
cmp	564	56 477	2 165	30	63	2	174
compress	1 013	48 594	1 926	19	89	0	191
gnugo	2 215	88 789	3 410	31	163	4	405
patch	3 846	143 431	5 579	118	391	3	729
bc	6 804	207 142	7 836	375	521	1	1 282
flex	8 312	420 525	15 848	165	2 279	14	1 115
bison	8 740	256 609	11 418	158	556	4	1 813
diff	10 114	389 052	13 031	208	363	2	2 369

Figure 4.1: Test suite programs

program code, there are only comparably few statements (and hence SUIF tree nodes) in relation to the number of lines.

Besides the sizes of the different programs, the table contains the number of functions, as well as the number of global and local (static and non-static) variables[1] occurring in the program. Local static variables have only been seldom used, and will hence hardly influence the behaviour of the algorithms.

One program which catches the eye is the lexical analyser generator 'flex' that uses more than two-thousand global variables. As will be seen in the following, this results in significantly larger analysis times compared to other programs of similar sizes which have less global variables. This program surely is a challenge to any kind of static analysis and was included to show that even programs that do not try to encapsulate and modularise its data can be analysed. Of course the analysis takes profit from structured programming and honours the avoidance of global variables by producing better analysis times.

Before going into the details of the test results, a short explanation of the analysed programs will be given in the following list.

queens: Solves the eight queens problem. Prints all possible solutions.

quicksort: Sorts a field of integers using the quicksort algorithm. Shows the value pairs that are exchanged.

factor: Computes the prime factors of a given number using the sieve of Eratosthenes. To make the computation more efficient the program contains an array holding all primes up to 65.537.

[1]Note that the number of variables includes temporary variables generated by the SUIF compiler in some cases.

wf1: Prints a sorted list of all words read from a file and counts how often they occurred in the text.

banner: Prints a given text in huge letters made up from hash signs. This program contains large arrays holding the pixel definitions for each character that can be printed.

cal: Prints the calendar page(s) for either a month or a year.

cmp: Compares two files. Shows the first line and column where the two files differ as well as the differing values.

compress: Compress or decompress one or more files using Lempel-Ziv coding.

gnugo: Plays the Chinese game Go with the user.

patch: Applies changes that have previously been recorded by 'diff' to a file.

bc: An arbitrary precision calculator language.

flex: Fast lexical analyser generator, produces programs for lexical tasks.

bison: GNU project parser generator (similar to 'yacc').

diff: Displays line-by-line differences of text files.

4.2 Normalisation pass

The normal form computation is the first and most simple of the analysis passes. As already mentioned in section 2.3 the code is slightly restructured and enlarged during this pass. The number of tree nodes increases here since complex statements, that have formerly been represented by a single tree node, are now split up into multiple statements that will then be represented by multiple tree nodes. The enlargement rates are in all cases below fifty percent. However, since the enlargement rates are only based on the number of tree nodes (statements) of a program they will surely be smaller if one looks at the program at a more fine grained level. The enlargement rates do not take into account that the statements represented by a certain tree node become smaller if they are split up into two different statements / tree nodes. Nevertheless, the average size of the statements represented by a tree node surely does not vary significantly between different programs, and hence it does not matter if it is equally reduced for all tested programs during the normalisation pass. In any case the number of tree nodes is surely a better measure for the program size than the number of lines or bytes usually used.

The table shown in figure 4.2 contains the number of tree nodes before and after normalisation. Furthermore, the overall analysis time as well as the average analysis time per tree node is given. As can be seen, there are no significant differences between the average analysis times per tree node. Since the times measured for the smaller tests are very close to the minimal time that can be measured, the results become comparably imprecise here. However, the bigger tests show that the overall time grows proportional to the size of the programs as expected.

program	final nodes	initial nodes	increase rate	time	time per node
	n	n_i	$\frac{(n-n_i)\cdot 100}{n_i}$ [%]	t_{nor} $[s]$	$\frac{t_{nor}}{n}$ $[\mu s]$
queens	56	48	16.7	0.003	54
quicksort	83	59	40.7	0.007	84
wf1	155	113	37.2	0.013	84
factor	162	111	45.9	0.019	117
banner	281	230	22.2	0.027	96
cal	522	390	33.8	0.052	100
cmp	792	564	40.4	0.087	110
compress	1 334	1 013	31.7	0.174	130
gnugo	2 674	2 215	20.7	0.249	93
bc	5 144	6 804	< 0.1	1.031	200
patch	5 370	3 846	39.6	0.855	159
bison	9 969	8 740	14.1	0.646	65
flex	10 104	8 312	21.6	1.729	171
diff	11 747	10 114	16.1	1.707	145

Figure 4.2: Normal form computation

4.3 SSA form computation

The next pass following normalisation is the SSA form computation. As stated in section 3.1.2.3 the theoretical worst case time bound of the SSA form computation indicates a time growing quadratically to the program size. As can be seen from the table shown in figure 4.3, the analysed programs do not reflect such a behaviour. The average analysis time per node increases at most slightly, while the sizes of the analysed programs increase rapidly. The highest values are reached by the programs 'flex' and 'bc', which is surely influenced by the fact that these programs contain extremely large functions generated by 'lex'.

Besides the average analysis time per node the table contains the average analysis time per variable and the average number of SSA variables generated to replace the variables of the original program. Like the average analysis time per node, the average analysis time per variable does not show a significant increase tendency. This shows that the SSA form computation can be done in a time growing almost proportional to the size of the analysed program. The average number of SSA variables generated can be taken as a measure of how successful the SSA form computation has been. Its value is in all cases bigger than two, clearly showing that there are enough values assigned directly (not via a pointer) to make it worth transforming the program into SSA form. The bigger this value gets, the better are the chances that the later passes of the analysis will represent different values by different value nodes, which will increase the precision of the analysis results.

A further effect that can be seen in the table, is that the number of variables grows almost proportionally to the size of the program. Hence this value could be used as an alternative measure for the program size. For most of the analysed programs the number of variables is a value close to the fourth of the number of nodes / statements.

program	nodes	variables	avg. SSA numbers	time	time per node	time per variable
	n	v	$\frac{v}{v_{ssa}}$	t_{ssa} [s]	$\frac{t_{ssa}}{n}$ [μs]	$\frac{t_{ssa}}{v}$ [μs]
queens	56	14	2.31	0.011	196	786
quicksort	83	19	2.19	0.019	229	1 000
wf1	155	31	2.68	0.033	213	1 065
factor	162	41	2.31	0.039	241	951
banner	281	73	3.89	0.078	278	1 068
cal	522	188	2.66	0.123	236	654
cmp	792	239	2.50	0.213	269	891
compress	1 334	280	3.59	0.413	310	1 475
gnugo	2 674	572	3.31	0.633	237	1 107
bc	5 144	1 804	2.68	3.503	681	1 942
patch	5 370	1 123	2.88	1.734	323	1 544
bison	9 969	2 373	4.05	2.789	280	1 175
flex	10 104	3 408	2.73	4.990	494	1 464
diff	11 747	2 734	4.57	5.167	440	1 890

Figure 4.3: SSA form computation

4.4 Intraprocedural analysis

After the SSA form computation has been executed, the algorithm proceeds with the intraprocedural analysis. The intraprocedural as well as the interprocedural analysis are surely the most interesting steps of the algorithm, and hence they will be discussed in more detail than the preceding passes.

The table shown in figure 4.4 shows the analysis times of the intraprocedural analysis as well as the average analysis times per node. Since two versions of the intraprocedural analysis algorithm have been discussed, both the analysis time and the average analysis time per node appear twice. The first time (t_{intra}) is the time measured with the conservative algorithm where every value has to be treated as if it were a pointer, whereas the second time (t_{intra}^{nica}) is the time measured with the algorithm assuming that there are no aliases generated by casts from pointers to ordinal types and back. The measured times of both algorithms do not show significant differences, and both grow with the program size. However, the time does not really depend on the overall program size but on the sizes of the analysed functions. As can be seen, the average function size of the larger programs is larger as well, and hence the growth of the average analysis time has to be blamed on the growing function sizes. However, it is not very likely that the function sizes will increase with the overall program size in general. The average function size is only small for the really small programs and does not seem to grow any further after reaching a certain saturation level. Assuming that the sizes of the functions do not grow with the overall size of the program the overall time needed to analyse a program grows linearly to the program size. This is confirmed by the results measured since the average analysis time per node shows no significant increase tendency if one only looks at the larger programs.

Some of the properties of the generated function interface graphs are shown in the table contained in figure 4.5. The table shows the number of variables represented by the function interface graphs (fig_v), the number of value nodes (fig_{vn} / fig_{vn}^{nica}) as well as the

program	nodes	nodes per function	time	time per node	time	time per node
	n	$\frac{n}{f}$	t_{intra} [s]	$\frac{t_{intra}}{n}$ [μs]	t^{nica}_{intra} [s]	$\frac{t^{nica}_{intra}}{n}$ [μs]
queens	56	19	0.057	1 018	0.055	982
quicksort	83	14	0.081	976	0.078	940
wf1	155	22	0.161	1 039	0.160	1 032
factor	162	54	0.209	1 290	0.204	1 259
banner	281	281	0.657	2 338	0.647	2 302
cal	522	44	0.756	1 448	0.736	1 410
cmp	792	26	1.427	1 802	1.437	1 814
compress	1 334	70	3.971	2 977	3.878	2 907
gnugo	2 674	86	5.735	2 145	5.713	2 136
bc	5 144	61	52.274	10 162	52.351	10 177
patch	5 370	46	20.496	3 817	20.241	3 769
bison	9 969	63	28.060	2 815	28.061	2 815
flex	10 104	61	45.308	4 484	45.021	4 456
diff	11 747	56	53.860	4 585	51.903	4 418

Figure 4.4: Intraprocedural analysis times

average number of value nodes that can be reached by a variable ($fig_{vn/v}$ / $fig^{nica}_{vn/v}$). One might wonder why the number of variables represented by the function interface graphs is bigger than the overall number of variables shown in one of the preceeding tables. However, this becomes quite clear if one takes into account that global variables used in different functions have their own independent representation in each function they occur in. Hence they are counted twice or even more often if they are used within multiple functions.

The number of value nodes as well as the average number of reachable value nodes are both given for the two versions of the algorithm. The number of nodes is in both cases more or less proportional to the program size, and indicates that the space needed for the intraprocedural analysis grows linearly to the program size as it was assumed in section 3.2.7.

Moreover, the number of nodes does not significantly differ for the two versions of the algorithm. This is caused by two contrary effects: on the one hand, the number of nodes generated by the more precise algorithm increases since less value nodes are to be merged, on the other hand, less value nodes are generated since it is not always necessary to generate nodes to represent non-pointer values.

The number of reachable nodes per variable shows that the graphs remain comparably flat. This is not very surprising since it is unlikely that extremely long chains of dereference operations occur in real programs. However, this does not imply that there are no long chains of values connected to each other in the memory (e.g. linked lists or trees), but usually such chains are accessed by recursive data structures that do not access their successors by long sequences of dereference operations, but by repeatedly using the same short sequence of dereference operations.

Summing up, one can say that the estimated time and space bounds for the intraprocedural analysis have been confirmed by the results produced by the tests. The estimated linear growth of the space needed by the algorithm corresponds exactly to the measured results. The estimated time bound of $O(|E| \cdot (|STMT| + |PHI|))$ per function depends on the product of the function size and the number of edges of the control flow graph. In the

program	nodes	fig variables	fig nodes	fig nodes per var.	fig nodes	fig nodes per var.
	n	fig_v	fig_{vn}	$fig_{vn/v}$	fig^{nica}_{vn}	$fig^{nica}_{vn/v}$
queens	56	26	72	4.31	76	3.35
quicksort	83	43	88	2.93	92	2.65
wf1	155	66	171	3.62	171	3.48
factor	162	80	168	2.81	172	2.83
banner	281	116	359	5.27	366	4.17
cal	522	283	730	3.58	740	3.22
cmp	792	477	1 062	3.03	1 091	2.94
compress	1 334	633	1 810	3.70	1 861	3.77
gnugo	2 674	1 059	3 116	4.08	3 221	3.71
bc	5 144	2 162	5 068	3.30	5 183	3.30
patch	5 370	2 787	7 205	3.28	7 360	3.21
bison	9 969	4 114	13 171	4.56	13 541	4.27
flex	10 104	4 351	11 012	3.48	11 293	3.33
diff	11 747	4 335	15 914	4.53	16 283	4.59

Figure 4.5: Properties of the function interface graphs generated by the intraprocedural analysis

best case, the size of the control flow graph does not grow when the function grows, and in the worst case it grows linearly to the size of the function. Altogether, this leads to a growth that ranges between linear and quadratical with respect to the function size. This corresponds quite well to the effects detected here, since the average analysis time per node grows more or less proportionally to the average function size.

4.5 Interprocedural analysis

The interprocedural analysis pass is the final step of the analysis. It merges the function interface graphs created to represent the different functions. In the following, two versions of the algorithm will be discussed: the call path sensitive algorithm and the purely call path insensitive algorithm.

The table shown in figure 4.6 shows the number of function calls appearing in the different programs. The calls have been split up into three different groups: calls to functions where the corresponding function definitions are available (c_{def}), calls to functions where only a prototype declaration has been given (c_{decl}) and finally those calls that are based on function pointers (c_{ptr}). As can be seen, function pointers are not used very frequently. In fact only two of the programs contain function calls using function pointers.

The program 'flex', that already has been striking because of its huge number of global variables, shows uncommonly large values again. Although it is not even the biggest program, it contains by far the largest number of function calls. As will be seen, the huge number of global variables together with the huge number of function calls produces extraordinary long analysis times as well.

The call path sensitive analysis processes the function calls one by one as long as suitable function calls exist. After all these calls have been processed, the call path insensitive

program	nodes	functions	function calls		
			defined	declared	pointer
	n	f	c_{def}	c_{decl}	c_{ptr}
queens	56	3	3	2	0
quicksort	83	6	9	4	0
wf1	155	7	15	9	0
factor	162	3	3	26	0
banner	281	1	0	34	0
cal	522	12	25	42	0
cmp	792	30	38	66	0
compress	1 334	19	28	132	0
gnugo	2 674	31	103	215	0
bc	5 144	85	477	296	8
patch	5 370	118	348	432	0
bison	9 969	158	484	646	0
flex	10 104	165	1 057	256	0
diff	11 747	208	318	417	3

Figure 4.6: Number of functions and function calls occurring in the test programs

program	sensitive calls	insensitive calls	indirect calls	overall calls
	c_{sens}	c_{ins}	c_{indir}	$c = c_{sens} + c_{ins}$
queens	3	0	0	3
quicksort	9	0	0	9
wf1	15	0	0	15
factor	3	0	0	3
banner	0	0	0	0
cal	25	0	0	25
cmp	38	0	0	38
compress	28	0	0	28
gnugo	103	0	0	103
bc	396	97	16	493
patch	348	0	0	348
bison	215	269	0	484
flex	940	117	0	1 057
diff	296	34	12	330

Figure 4.7: Number of function calls processed by the call path sensitive / insensitive analysis

analysis processes the remaining function calls, if there are any left. The next table, which is shown in figure 4.7, contains the overall number of function calls that have been processed (c) as well as the number of function calls processed by either phase (c_{sens} / c_{ins}) and the number of functions called indirectly (c_{indir})[2]. In most of the cases the call path sensitive analysis has been able to process all the occurring function calls, and hence it was not necessary to process any of the calls with the call path insensitive algorithm. This shows that complex call graphs that prevent a fully call path sensitive analysis do not appear very often, and that comparably good results can be achieved even if such complex call graphs have to be handled conservatively in some cases.

The overall number of processed function calls processed by the pure call path insensitive analysis is exactly the same as in the case of the call path sensitive analysis, and hence it has not been mentioned separately here[3]. For all but two of the programs, the number of processed function calls (c) equals the number of function calls where the called function has been defined (c_{def})[4]. Only the programs 'diff' and 'bc' contain function calls using function pointers, and hence process some more function calls. All the processed function calls of these programs correspond to function calls that will really be executed when the program is running. Hence no superfluous possible function calls have been detected, and the algorithm produced optimal results in this case.

The table shown in figure 4.8 shows the analysis times as well as the times per global variable and function call. Both the overall time as well as the time per global variable and function call appear twice. The first time was measured during the call path sensitive analysis (t_{inter}), whereas the second time corresponds to a purely call path insensitive analysis (t^{ins}_{inter}). In many cases the analysis times hardly differ from each other, but there are as well cases where some rather large differences occur. Neither the call path sensitive nor the call path insensitive analysis shows a significant advantage compared with the other algorithm if one only looks at the analysis times. This seems to be a little bit surprising since the call path insensitive analysis does not need to make copies of the function interface graphs and should hence be faster. As can be seen, this effect is at least sometimes compensated by the increasing time needed to merge the function interface graphs if no good order was chosen to merge them. Unlike the call path insensitive algorithm, the call path sensitive one prefers to process the call graph in a bottom-up direction, and hence it processes smaller call graphs for a while.

In section 3.3.5, the time bound for the interprocedural analysis was assumed to be in $O(|E_{CG_p}| \cdot |VAR_{global}|)$. Although the average analysis times per variable and function show comparably large differences there is no remarkable increase tendency, and hence the estimated time bound seems to match the results of the test.

The space needed by the interprocedural analysis depends on whether the call path sensitive or insensitive algorithm is chosen. In the first case it was assumed that the space needed by the algorithm grows at most with the size of the call graph and the size of the program ($O(|E_{CG_p}| \cdot |STMT_p|)$), although a significantly better result was expected. A look at the table shown in figure 4.9 shows that these expectations have not been too high. The table contains the number of value nodes as well as the number of value nodes per

[2]Note that the function calls calling functions indirectly (through pointers) are already included in the function calls processed during the call path insensitive phase.

[3]This has not to be the case in general since it could be that the call path insensitive analysis detects some more functions that can possibly be called by function calls using function pointers.

[4]The current implementation ignores function calls calling functions where the corresponding function definition is missing. This avoids the unnecessary merging of value nodes representing global variables when library functions like `printf` are called.

program	global variables	calls	time	time per variable and call	time	time per variable and call
	v_g	c	t_{inter} [s]	$\frac{t_{inter}}{v_g \cdot c}$ [μs]	t^{ins}_{inter} [s]	$\frac{t^{ins}_{inter}}{v_g \cdot c}$ [μs]
queens	6	3	0.008	444	0.007	389
quicksort	3	9	0.013	481	0.012	444
wf1	6	15	0.036	400	0.034	378
factor	27	3	0.018	222	0.012	148
banner	51	0	< 0.001	–	< 0.001	–
cal	85	25	0.293	138	0.236	111
cmp	63	38	0.640	267	0.521	218
compress	89	28	0.663	266	0.708	284
gnugo	163	103	6.305	376	3.466	206
bc	314	493	115.106	744	72.631	469
patch	391	348	61.849	455	73.517	540
bison	556	484	220.937	821	153.696	571
flex	2 279	1 057	3 713.906	1 542	981.889	408
diff	363	330	47.792	399	31.996	267

Figure 4.8: Interprocedural analysis times

program	nodes	calls	fig nodes	fig nodes per node	fig nodes	fig nodes per node
	n	c	fig_{vn}	$\frac{fig_{vn}}{n}$	fig^{ins}_{vn}	$\frac{fig^{ins}_{vn}}{n}$
queens	56	3	26	0.46	26	0.46
quicksort	83	9	52	0.63	45	0.54
wf1	155	15	100	0.65	67	0.43
factor	162	3	129	0.80	115	0.71
banner	281	0	213	0.76	213	0.76
cal	522	25	433	0.83	271	0.52
cmp	792	38	859	1.08	357	0.45
compress	1 334	28	882	0.66	607	0.46
gnugo	2 674	103	2 976	1.11	909	0.34
bc	5 144	493	9 119	1.77	1 850	0.36
patch	5 370	348	12 715	2.37	2 408	0.45
bison	9 969	484	3 173	0.32	2 469	0.25
flex	10 104	1 057	17 100	1.69	3 365	0.33
diff	11 747	330	11 739	1.00	2 267	0.19

Figure 4.9: Function interface graph sizes of the interprocedural analysis

statement (SUIF tree node) for both the call path sensitive and the call path insensitive algorithm. Although the number of value nodes per statement increases with the program size, this value grows far more slowly than the number of function calls. Altogether, this means that the space needed by the call path sensitive algorithm grows as expected in section 3.3.5. In the case of the call path insensitive algorithm this number decreases with growing program size, which shows that the linear growth with the program size that was assumed in section 3.3.5 is a suitable space bound as well.

The alias analysis has been applied to a series of C programs successfully. The results show that even the larger programs can be analysed with reasonable time and space costs. However, the structure of the program plays an important role if it comes to analysis times. The more effects of the program are kept local by encapsulating data into functions, the better are the results. As expected, it can be seen that programs avoiding the use of global variables whenever possible can be analysed much faster than other ones.

Chapter 5

Related work

In the following of this chapter an overview of the differences between our approach and other existing approaches dealing with alias analysis will be given. Furthermore other algorithms related to the ones used by our approach, as well as some basic properties of such algorithms will be discussed.

5.1 Other alias analysis algorithms

Most of the alias analysis algorithms make certain restricting assumptions about the analysed programs, e.g. that the program does not contain type casts, structures or other constructs of the C programming language causing problems. There are only very few approaches which try to deal with all or at least almost all of the problems caused by analysing C programs.

The algorithm that comes closest to our approach was described in Wilson's lately published Ph.D. thesis ([Wil97]). An earlier version of the algorithm had already been described in [WL95]. This algorithm is one of the few that are able to handle all those C language features that are handled by our algorithm as well. However, the way in which these problems have been solved differs in some aspects. Wilson uses so-called partial transfer functions that are computed for every function of the program. Unlike function interface graphs, partial transfer functions do not represent all the functions effects since they are designed to represent function calls having a certain alias pattern. Since a function can be called from within many different calling contexts, there are multiple transfer functions necessary to represent a function in this case. To deal with the effects of structures, unions and type casts the algorithm uses a memory representation quite similar to ours, however pointer arithmetic is sometimes handled less precisely. Another difference between the algorithms is how recursive function calls are handled: our approach does not need any special treatment for recursive functions, whereas the other algorithm has to re-compute the partial transfer functions for recursive function calls until stabilisation is reached which may lead to several re-computation iterations in some cases.

The problems caused by dealing with pointers, labels, function pointers or variables have as well been addressed in [Wei80]. Unlike in our approach, the possible aliases are computed by building relations representing dependencies between the variables of a statement. The algorithm is dominated by the computation of a transitive hull to find the dependencies between all variables. Besides, there is a further significant difference between this and our approach since it is assumed that there is absolutely no information about the control flow available. This means that neither the call path nor the intraprocedural control flow is taken into consideration. Although the intraprocedural control flow has only partially been taken into consideration by our analysis, the computation of the SSA form depends on the control flow as well, and hence makes the analysis more control flow sensitive. Since our analysis is call path sensitive and at least partially control flow sensitive it will hence produce more precise results in most cases.

Two further flow insensitive algorithms that are able to deal with all the properties of the C programming language are presented in [BCCH95] and [Ste96]. The first algorithm propagates sets of aliased variables through the call graph, as well as through the control flow graphs of the different functions. Like our algorithm, it updates the call graph whenever new aliases of function pointers have been found. However, since this algorithm computes the effects of a function depending on the aliases that are found to hold at the call site, it is necessary to re-compute the previous results if new aliases are found later. Unlike our approach, this algorithm may need an alternating sequence of re-computations on the interprocedural and intraprocedural level until stabilisation is reached.

The second algorithm ([Ste96]) uses sets of rules that are comparable to those used within type systems. These rules are needed to compute sets of abstract memory locations that may be aliased. The results are roughly comparable to those of Weihl's algorithm, and are hence less precise than ours. However, unlike Weihl's algorithm, this one is significantly faster, since it can be performed within almost linear time.

Besides the approaches mentioned so far, there are several more trying to find the aliases produced by C program code. E.g. the algorithms presented in [Cou86, Deu94, EGH94, LR92, LRZ93, PRL91, Ruf95, HHN94, ZRL96] all present alias analysis algorithms based on the C programming language, but all of them make considerable restrictions to the analysed programs. Type casts and unions are one of the major sources of problems when dealing with the alias analysis of C code. Therefore many algorithms cannot deal with these commonly used properties of the C programming language (e.g. [Deu94, HHN94, LR92, LRZ93, ZRL96]). Other algorithms can handle type casts and unions but assume that there are no type casts between pointer and non-pointer types (e.g. [Ruf95]). Even if the problems caused by type casts and unions have been solved, other problems remain unsolved in some cases. E.g. the algorithm presented in [EGH94] cannot handle label and jump statements without transforming the control flow into data flow as described in [EH93]. Although the transformation does not change the program input / output semantic, it may still affect (enlarge) the sets of *possible* aliases that will be detected. The algorithm presented in [PRL91] cannot handle recursive data structures and multi-level pointers and the algorithm presented in [Cou86] is purely intraprocedural and hence has to make extremely conservative assumptions about the existing aliases when a function is entered. The different restrictions that have been made by the above algorithms were necessary to use techniques that will result in more precise results in some cases. Since many of these restrictions (e.g. not allowing type casts) will result in the fact that most real C programs cannot be analysed, we accepted the additional imprecision caused by our algorithm to be able to analyse most of the existing programs.

Many of the techniques described above focus on finding aliases between either stack or heap locations. In [GH98] it is described how both techniques can be combined within a single framework. A further algorithm computing aliases of C programs is presented in [YRLS97]. This approach deals with the problem of computing alias information incrementally. Such information can be particularly useful within programming environments to aid the user while programming. The problems posed by polymorphism are discussed in [TAFM97]. Especially the effects of virtual functions like those used in C++ programs have been discussed here.

Many of the problems solved by our algorithm are caused by those properties of the C programming language that have simply been ignored by many other algorithms. Several of these as well as similar problems are reported to occur when C code is to be parallelised (e.g. [AJ88]).

However, there are as well many problems that have been solved by other algorithms which do not deal with C code as well. Most of the older alias analysis algorithms have been designed to analyse programs written in less problematic languages like Fortran or Pascal or subsets of these languages. Since these languages restrict the use of pointers, some of the problems that have been discussed within this work will not even occur there. An algorithm dealing with Pascal-like pointers is presented in [CR82]. However, most of the algorithms based on such languages do not handle pointers at all, and hence the problem of finding aliases can be reduced to find aliases due to call by reference parameters (e.g. [Ban79, BC92, CBC93, Coo85, CK91, HS94]). A commonly used technique that can be used in such cases is the computation of sets holding the values that are used and / or modified by a certain statement (USE and MOD sets). These sets can then be iterated based on the control flow or call graph. Almost all of the approaches given above are based on this data flow analysis technique. Several of the approaches computing USE or MOD or similar sets use so called interprocedural flow graphs which join the information contained in the call graph with the information contained in the control flow graphs of the different functions (e.g. [HS94, PRL91]). This avoids propagating the alias information on two different levels (control flow graph and call graph).

All the algorithms that have been mentioned up to here have been designed to work with existing imperative languages. However, there are other algorithms which deal with other languages as well. E.g. all the algorithms presented in [CWZ90, Deu90, HPR89, NPD87] are based on a language that is a mixture of an usual imperative language and Lisp. Like other imperative languages, these languages have features like loops and assignment statements, but the memory allocation is based on Lisp-like structures. Finally, there are approaches like the one presented in [LH88] dealing with purely functional languages like Lisp.

Besides the differences between the analysed languages there are several other differences between the various alias analysis algorithms. Many of them are interprocedural algorithms like ours, but there are as well purely intraprocedural ones like the one described in [Cou86]. On the one hand intraprocedural algorithms have to be very conservative in some cases, and are hence less precise. This is caused by the fact that there is nothing known about the possible aliases between the functions parameters and / or global variables. On the other hand it is possible that intraprocedural algorithms are more precise since they can use algorithms that are not suitable for interprocedural algorithms due to time or space costs.

The different approaches that have been made to compute the aliases of a program use different representations of the aliases. Some of these approaches have already been mentioned in section 3.2.2. Various algorithms exist which use graphs similar to our function interface graphs to represent the alias relations. E.g. the algorithms presented in [CWZ90, LH88] both use such graphs. All of these graphs are a compact representation of the memory locations and / or the values stored at these locations. The other alternative to represent aliases is to compute sets of access paths that are aliased. In many cases this is done by using the above-mentioned USE and MOD sets.

A further problem that occurs, no matter if access paths or graphs are used to represent the aliases, arises from the use of recursive data structures. Recursive data structures are not in the centre of interest for imperative languages, but they are essential for languages using Lisp-like data structures. Hence some of the algorithms designed to work with imperative languages do not deal with the problems caused by recursive data structures, whereas algorithms using Lisp-like data structures have to. Here the major problem is that the access paths which access recursive data structures can become infinitely long. Nevertheless it is necessary to find a finite representation for such access paths. A commonly used approach is to limit the number of references by a certain number (k-limiting). In this case all elements that are reachable by using more than the specified number of references (k) are treated as the same element. Since k-limiting is quite imprecise in some cases, there have been various similar approaches trying to increase the precision (e.g. [CWZ90, HPR89, LH88]). As an example the algorithm presented in [LH88] does this by the use of regular expressions representing the possibly infinite sequences of references.

Most of the algorithms try to analyse existing programs as they are (as our algorithm does), whereas other algorithms are based on additional information that has to be provided by the user or a preceding analysis pass. E.g. the algorithm presented in [HHN92, HHN94] describes an approach that is based on axioms specifying certain properties of linked lists, trees or other more or less regular data structures. Based on such information more precise results can be computed in some cases. The major problem here is that some of the functions which access recursive data structures like trees have to build or modify these data structures and that this can temporarily violate some of the data structures properties like being a tree or non-circular. Unfortunately it was not described how such rules could be generated automatically, which leaves this work to the user.

5.2 Other interprocedural analysis algorithms

Many of the problems that have been described within this work are not specific for alias analysis algorithms, but occur in other interprocedural analysis algorithms as well. Two commonly used techniques here are the above-mentioned computation of USE and MOD (or similar) sets (e.g. [Bar77, Bar78, CK84, CK88]) and the computation of definition-use chains (e.g. [All74, HS94]). Based on such information, various other problems can be solved like live variables, available expressions, etc.. Another way of solving such problems is to propagate the necessary information directly using the call graph and the control flow graph (e.g. [Mye81, Bur90]) or an interprocedural control flow graph (e.g. [SP81]). A similar approach is described in [Cal88, Goo97] where so-called program summary graphs are used to propagate the information.

A slightly different approach is discussed in [CCKT86, JM82, GT93] where some constant propagation algorithms based on a semi-lattice are presented. These algorithms compute functions representing a functions effect that are then used to propagate the constants through the call graph. Another algorithm that can be used for constant propagation can be found in [RL86]. However, this algorithm is based on definition-use chains and does not propagate the constants along the control flow graph directly.

Some problems specific to interprocedural program optimisation and parallelisation are discussed in [MSE95]. The algorithm presented there is based on the computation of so-called Bernstein sets which are extended versions of the USE and MOD sets.

How interprocedural analysis techniques can be integrated into a software development environment is discussed in [CKT85]. Here interprocedural data flow information is collected by an intelligent editor that keeps track of the source code modifications and stores its result in a data base. This data base can later be used for various purposes like constant propagation, code optimisation or alias computation. One of the major problems is to avoid the exhaustive computation of data flow information when only small changes were made to a previously analysed program. A further algorithm dealing with related topics is presented in [DGS97], where a demand driven approach is compared with an exhaustive analysis strategy. Instead of computing data flow information for the complete program, the presented algorithm computes only information that has been requested by the user or the program using the algorithm. Depending on the task to be performed this may help to save time in many cases.

All the above-mentioned algorithms are either based on interprocedural flow graphs or on call graphs. In cases where function variables are allowed, the computation of the call graph becomes more complicated. The algorithm presented in [HK92] shows how the call graph can be computed efficiently if functions are passed as parameters as done in languages like Fortran.

Although all the above-mentioned algorithms differ in several aspects, there is one aspect common to all these algorithms: since they are not intended to solve pointer alias problems they can at the most handle aliases produced by reference parameters and hence have to make conservative assumptions when pointers are involved.

5.3 SSA form

Although many aspects of SSA form have already been discussed in section 3.1, a more complete overview will be given in the following. SSA form was first mentioned in [AWZ88]. Since then it has been used frequently, and its computation has been improved several times. One essential part of the SSA form computation is to find the places where ϕ- functions are necessary. All the algorithms presented in [CFR$^+$89, CFR$^+$91, CF95] deal with the problem of making the SSA form computation faster and more efficient. As stated in section 3.1, the computation of the ϕ- functions can be, and usually is, based on a preceding computation of the dominator tree and dominance frontiers. Algorithms to compute the dominator tree have been presented in [LT79] and [Har85]. An incremental algorithm computing dominator trees has been described in [SGL97].

Once the SSA form has been computed, there are various applications that can benefit from the additional information provided by a program in SSA form. At first, SSA form has been used to implement a more efficient algorithm which eliminates redundant computations ([RWZ88]). A more recent approach deals with partial redundancies as well ([CCK$^+$97]) which is done by assigning the values of sub-expressions to temporary variables. Various other analysing and optimising algorithms take advantage of SSA form as well. E.g. the algorithm described in [MJ92] uses SSA form to compute dependencies and to perform various code optimisations.

Like many others, our algorithm is based on a preceding transformation of the code into SSA form. Our algorithm does not really depend on the transformation of the code into SSA form, but it is more precise if the transformation is done. However, there are other algorithms like [CG93] or [HH98] that integrate the SSA form computation into the alias analysis itself. In this case the computation of the SSA values is an essential part of the alias computation since indirect variable modifications are taken into consideration as well and produce new SSA values. This means that if e.g. a variable `a` is modified because `b` holds `a`'s address and a statement like $*$`b` $=$ `c` is executed, this results in a new SSA value for `a` to be generated. Our SSA algorithm simply ignores such effects and leaves it to the following analysis parts to detect the indirect modification of `a`.

Usually SSA form treats all elements of an array as a single element, which of course is a source of imprecision. In [KS98] it is discussed how this imprecision can be avoided. The algorithm presented there computes a so-called array SSA form that takes the index used to access the array elements into account as well. Furthermore, it has been shown how this extension of SSA form can be used in automatic parallelisation.

5.4 Data structures used by data flow analysis algorithms

Our algorithm is based on the SSA form computation, which again is based on the control flow graph and the corresponding dominator tree. However, a wide variety of other data structures exists which are used for similar purposes like data dependence graphs or control dependence graphs. Both data and control dependencies are represented in the program dependence graph which was first mentioned in [FOW87]. An application for these graphs can be found in [Agr94] and [HRB90], where program dependence graph based algorithms are presented in the context of program slicing.

Further concepts are described in [JP93] and [JPP94]. The first describes how dependence flow graphs can be computed and used for the computation of SSA values, whereas the second introduces so-called program structure trees, representing a program based on its minimal single-entry-single-exit regions. Besides the computation of the program structure trees itself, an algorithm is described that shows how these trees can be used to calculate ϕ- functions efficiently.

The algorithm described in [CCF91] constructs graphs that are smaller than the usual control flow graphs and shows how these graphs can be used to solve several data flow problems. An extension of the dominator tree concept where a data structure called augmented post dominator tree is used to represent control dependencies more efficiently, is presented in [PB95].

Most of the existing data flow analysis frameworks are based on the names of variables or on the lexical name of expressions. The algorithm presented in [BA98] shows how it is possible to keep track of computed values (not only of those stored by a certain variable) using so-called value name graphs. This algorithm is able to detect equal computations even if they are lexically different.

5.5 Complexity of alias analysis

All the alias analysis algorithms mentioned so far have to make some restrictions to compute the results within reasonable time. Depending on the different restrictions, the problem of finding aliases becomes more or less expensive. An overview of these costs is given in [LR91]. Here the costs depend on whether single or multi level pointers, structures or reference formal parameters are allowed or not. The effects of the different restrictions are discussed for may and must alias analysis, as well as for intraprocedural and interprocedural analysis.

Later it was shown in [Lan92] and [Ram94] that the may alias problem is not recursive (undecidable) and that the must alias problem is not recursive enumerable (uncomputable) even if all paths through the program are assumed to be executable. It was shown that precise alias analysis becomes $\mathcal{NP}$-hard if multi level pointers are taken into account (even in the case of intraprocedural analysis without dynamic memory allocation). Further, it was shown in [Hor97] that the same still holds for flow insensitive analysis. A more general investigation was made in [BR87], where several incremental iterative data flow analysis algorithms (including alias analysis) and their costs have been discussed.

As reported in the sections dealing with the time and space bounds of our algorithm, the worst case time and space bounds differ significantly from what can be inspected if the algorithm is applied to real programs. Such effects can be inspected for many other algorithms as well, and they are caused by a series of properties real programs have. An empirical study of C programs where such properties are discussed can be found in [RP88]. Here programs are analysed to collect statistical information about the number of parameters, the parameter types, the number of called functions or the frequency of pointer usage for a given function. The frequency of dynamic pointer references has as well been discussed by Miller in [Mil88]. He found out that most of the analysed programs contained less than 10 percent statements using pointer references, most of them contained in calls of library functions.

Although many researchers have provided empirical results for their alias analysis algorithms, it is in many cases quite complicated to compare two of these algorithms. The problem is that everyone analyses his own individual set of programs, and that, due to differing program representations, the measured values can not be compared with each other. This problem has been addressed in [HP98, SH97, SRLZ98], where at least some of the differing approaches have been compared under equal conditions.

5.6 Possible applications for alias analysis

The probably most significant application for alias analysis is code optimisation. A wide variety of algorithms exists which optimise code for different purposes. Many sequential compilers optimise their code using techniques like constant propagation, dead code elimination, redundancy elimination and variable lifetime analysis or by optimising the use of registers. These and various other data flow problems have been discussed in great detail over the years (e.g. [BC85, BC94, DGS93, Hec77, KRS94, Pin93, KH93, RM88]).

Although such optimisation techniques can be very useful and produce significant speed ups for sequential programs, the improvements gained can be much bigger when sequential programs are optimised to be executed on a parallel computer. In this case even some of the more expensive optimisation techniques that are otherwise not used can become worthwhile. A wide variety of algorithms exists which parallelise sequential code for the different parallel target machine architectures (e.g. [ACK87, AL93, BC86, FG94, GK94, LP94, PP94, MSE93, SSOG93, Tur86, ZC91]). However, even those programs that have already been designed to work on a parallel computer can be further optimised based on alias information (e.g. [GS93, MR93]). Most of the algorithms described in the above-mentioned articles are based on Fortran since Fortran is much more restrictive than the C programming language. Probably C was not chosen since it is comparably hard to determine if two arrays or their elements are aliased or not within C programs. Once it has been determined that no aliases exist, most of the techniques used for the Fortran programs can be adapted to C programs easily.

Besides optimising algorithms, alias analysis can as well be particularly useful when memory allocation / usage errors have to be found. There are approaches to detect such errors at execution time (e.g. [ABS94]) as well as at compile time (e.g. [Bou93, Eva96, JJKSb97, RSY94]). All these algorithms would benefit from including alias information since conservative assumptions that had to be made could be avoided in some cases then.

A further application for alias analysis are programming environments which provide alias information to aid the user when developing a program (e.g. [CKT85, Kai89, RTD83]). Such information could be helpful to detect possible errors before they will occur or to avoid them right from the start.

Chapter 6

Conclusion and future work

The alias analysis algorithm presented here is able to deal with almost all the properties of ANSI C programs. The analysis is call path sensitive, but not fully flow sensitive. Since the algorithm was especially designed to deal with all the properties of the C programming language it was not possible to rely on the available type information. Therefore conservative assumptions had to be made in some cases. To avoid the loss of precision caused by these assumptions the user can still switch to the more precise version of the algorithm if the analysed program does not produce aliases based on casts from pointer to integer values and back.

The algorithm was implemented using the SUIF compiler system, and it has been applied to several existing C programs. Many of the other alias analysis algorithms are not able to analyse the same programs since they make restrictions on the C programming language. Furthermore, the largest program contained in our test suite is much larger than the largest programs in the test suites used to test many of the other algorithms. Altogether this shows that our algorithm is a suitable tool to analyse real life C programs efficiently.

However, there are several aspects / improvements which have not been treated here and hence are topics for future improvements. Currently, the dereference operations used to specify the references between the different nodes of the function interface graphs are not very precise. If different dereference operations are applied to the same node it is comparably fast assumed that any possible dereference operation could have been used, which is reflected by the use of an unknown offset value. In some cases it would be possible to avoid unknown offsets if a different data structure had been used to store the range of values that can be accessed. However, the sets of values that can be accessed by a particular dereference operation can become fairly complex. E.g. the values that can be accessed by an expression like `a.c1[i1].c2[i2]` depend on the offsets of the structure components of `c1` and `c2`, as well as on the size of the elements of the array `c2`. In this case the accessible locations are neither a connected region nor are they finite. The exact set of locations that can be accessed can be represented by sets of linear equations. However, it is not clear if the additional costs used to store the equations are justified by the hereby gained increase of the precision. Especially uniting dereference operations or testing if two dereference operations can access the same value becomes much more complicated if this has to be done exactly. It remains to be found out whether a compromise can be found that produces more precise results than the current algorithm and is nevertheless able to compute united dereference operations or the intersections of the access ranges of two dereference operations efficiently enough.

Besides increasing the precision of the analysis it might be interesting to adapt the analysis to other programming languages. In many cases this does not cause any problems since the corresponding language is more restrictive than the C programming language (e.g. Pascal or Fortran). However, there are as well languages which cause problems that have not yet been handled by our algorithm. Although C++ or Java are both derived from the C programming language, they cannot be analysed properly by our algorithm as it is so far. The major problems caused by these languages are virtual function calls and exception handling. Virtual functions could be integrated into our algorithm quite easily by handling them similar to function calls using function pointers. In fact virtual functions are only a syntactically nicer and more structured version of function pointers. Instead of using function pointers explicitly the user defines a virtual function, and the compiler generates a function call using a function pointer that has been looked up from the virtual function table. Of course virtual functions have some other properties that allow the compiler to perform some checks that would otherwise be impossible. The major problem caused by virtual functions is that they are used frequently within C++ programs, whereas function pointers are not used very often. This means that the call graph is blown up if many virtual functions are used, and that this is surely reflected by increasing time and space costs. The second problem is even harder to handle. Exceptions as they are used by C++ or Java may cause jumps out of functions back to a directly or indirectly calling function. This concept is in some aspects similar to the use of `setjmp` and `getjmp` functions. Of course it is possible to handle exceptions in a way similar to `setjmp` and `getjmp` functions, although this does not seem to be a very satisfying solution in this case. On the one hand exceptions are surely used more often by C++ programs than `setjmp` and `getjmp` functions are used within C programs, and on the other hand exceptions are not commonly used only to jump to a single central error handling function. Here there is surely a need to find more satisfying solutions in the future.

A further interesting extension of the currently existing algorithm would be to develop an incremental version of the algorithm. The problem arising here is that it is quite hard to find out which nodes have been merged due to code that has been removed after an incremental update, and to undo the merging steps caused by this code. If one does not want to worsen the analysis precision by simply ignoring this problem one has to find a way to keep track of the changes made. On the function level as well as on the program level it is possible to store the function interface graphs belonging to the different nodes of the control flow graph or the call graph. In this case only the merging process of the function interface graphs has to be repeated if a function interface graph belonging to one of the nodes was changed. While this might be an acceptable solution on the function level, where the number of function interface graphs as well as their sizes are comparably small, the same scheme will result in extremely long update times on the program level. Here it might be useful to keep copies of the various intermediate function interface graphs. Of course this would increase the space needed by the algorithm significantly. Another alternative might be to extend the data structures so that nodes that have been merged can be separated again later. Of course this will lead to increasing space costs as well, but the increase rate will probably be smaller than in the previous case.

Bibliography

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301. ACM, June 1994.

[ACK87] Randy Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 63–76. ACM, January 1987.

[Agr94] Hiralal Agrawal. On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312. ACM, June 1994.

[AJ88] Randy Allen and Steve Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 241–249. ACM, June 1988.

[AL93] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–125. ACM, June 1993.

[All74] F. E. Allen. Interprocedural analysis and the information derived by it. *Lecture Notes in Computer Science*, 23:291–314, September 1974.

[Ame78] American National Standards Institute. *Fortran-77*, 1978.

[Ame89a] American National Standard for Information Systems, New York. *Programming Language - C*, December 1989.

[Ame89b] American National Standard for Information Systems, New York. *Rationale for American National Standard for Information Systems - Programming Language - C*, December 1989.

[And92] Lars Ole Andersen. Partial evaluation of C and automatic compiler generation. In Gerhard Goos and Juris Hartmanis, editors, *Lecture Notes in Computer Science: 4th International Conference on Compiler Construction*, pages 251–257. Springer Verlag, October 1992.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau.* Addison Wesley (Germany) GmbH, 1988.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11. ACM, January 1988.

[BA98] Rastislav Bodik and Sadun Anik. Path-sensitive value-flow analysis. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 237–251. ACM, January 1998.

[Ban79] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th Annual ACM Symposium on Principles of Programming Languages*, pages 29–41. ACM, January 1979.

[Bar77] Jeffrey M. Barth. An interprocedural data flow analysis algorithm. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 119–131. ACM, January 1977.

[Bar78] Jeffrey M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[BC85] Jean-François Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[BC86] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 162–175. ACM, July 1986.

[BC92] Michael Burke and Jong-Deok Choi. Precise and efficient integration of interprocedural alias information into data-flow analysis. *ACM Letters on Programming Languages and Systems*, 1(1):14–21, March 1992.

[BC94] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–170. ACM, June 1994.

[BCCH95] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, 892*, pages 234–250. Springer-Verlag, August 1995. Proceedings from the *7th Workshop on Languages and Compilers for Parallel Computing.*

[Bou93] Franiçois Bourdoncle. Abstract debugging of higher-order imperative languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 46–55. ACM, June 1993.

[BR87] M. Burke and B. G. Ryder. Incremental iterative data flow analysis algorithms. Technical Report LCSR-TR-96, Laboratory for Computer Sciences, Hill Center for the Mathematical Sciences, Busch Campus, Rutgers University, New Brunswick, New Jersey 08903, August 1987.

[Bur90] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[Cal88] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–56. ACM, June 1988.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM, January 1993.

[CC95] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[CCF91] John-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 55–66. ACM, January 1991.

[CCK$^+$97] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 273–286. ACM, June 1997.

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 152–161. ACM, July 1986.

[CF95] Ron K. Cytron and Jeanne Ferrante. Efficiently computing phi-nodes on-the-fly. *ACM Transactions on Programming Languages and Systems*, 17(3):487–506, May 1995.

[CFR$^+$89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35. ACM, January 1989.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CG93] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 36–45. ACM, June 1993.

[CK84] Keith D. Cooper and Ken Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 247–258. ACM, June 1984.

[CK88] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–66. ACM, June 1988.

[CK91] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 49–59. ACM, January 1991.

[CKT85] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization on the design of a software development environment. In *ACM SIGPLAN Symposium on Language Issues in Programming Environments*, pages 107–116. ACM, June 1985.

[Com94] Computer Systems Laboratory, Stanford University, CA 94305. *The SUIF Compiler System*, 1994. ftp: suif.Stanford.EDU, WWW: http://suif.Stanford.EDU.

[Coo85] Keith D. Cooper. Analyzing aliases of reference formal parameters. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 281–290. ACM, January 1985.

[Cou86] Deborah S. Coutant. Retargetable high-level alias analysis. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 110–118. ACM, January 1986.

[CR82] Anita L. Chow and Andres Rudmik. The design of a data flow analyzer. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 106–113. ACM, June 1982.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310. ACM, June 1990.

[Deu90] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 157–168. ACM, January 1990.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241. ACM, June 1994.

[DGS93] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–77. ACM, June 1993.

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256. ACM, June 1994.

[EH93] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. Technical Report ACAPS Technical

Memo 76, Architectures, Compilers and Parallel Systems Group, School of Computer Science, McGill University, 3480 University St., Montreal, Canada, H3A 2A7, September 1993.

[Eva96] David Evans. Static detection of dynamic memory errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–53. ACM, May 1996.

[FG94] Allan L. Fisher and Anwar M. Ghuloum. Parallelizing complex scans and reductions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 135–146. ACM, June 1994.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GH98] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 121–133. ACM, January 1998.

[GK94] Mark R. Gilder and Mukkai S. Krishnamoorthy. Automatic source-code parallelization using HICOR objects. *International Journal of Parallel Programming*, 22(3):303–350, March 1994.

[Goo97] David W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 122–133. ACM, June 1997.

[Gou88] K. John Gough. *Syntax Analysis and Software Tools*. Adison-Wesley, 1988.

[GS93] Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–168. ACM, May 1993.

[GT93] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 90–99. ACM, June 1993.

[Har85] Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 185–194. ACM, May 1985.

[Hec77] Mathew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Scientific Publishing Company, 1977.

[HH98] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 97–105. ACM, May 1998.

[HHN92] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3):243–260, September 1992.

[HHN94] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–229. ACM, June 1994.

[HK92] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[Hor97] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[HP98] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In Giorgio Levi, editor, *Lecture Notes in Computer Science, 1503*, pages 57–81. Springer-Verlag, September 1998. Proceedings from the *5th International Static Analysis Symposium.*

[HPR89] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 28–40. ACM, June 1989.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HS94] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[JJKSb97] Jakob L. Jensen, Michael E. Jörgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 226–234. ACM, June 1997.

[JM82] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 66–74. ACM, January 1982.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–89. ACM, June 1993.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–185. ACM, June 1994.

[Kai89] Gail E. Kaiser. Incremental dynamic semantics for language-based programming environments. *ACM Transactions on Programming Languages and Systems*, 11(2):169–193, April 1989.

[KH93] Priyadarshan Kolte and Mary Jean Harrold. Load/store range analysis for global register allocation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277. ACM, June 1993.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 147–158. ACM, June 1994.

[KS98] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 107–120. ACM, January 1998.

[KW92] U. Kastens and W. M. Waite. Modularity and reuseability in attribute grammars. Technical Report CU-CS-613-92, Department of Computer Science, University of Colorado, Boulder, Colorado 80309, September 1992.

[Lan92] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 21–34. ACM, June 1988.

[LP94] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, February 1994.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 93–103. ACM, January 1991.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248. ACM, June 1992.

[LRZ93] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 56–67. ACM, June 1993.

[LT79] Thomas Lengauer and Rober Endre Tarjan. A fast algorithm for finding dominators in a flow graph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[Mil88] Barton P. Miller. The frequency of dynamic pointer references in "C' programs. *ACM SIGPLAN Notices*, 23(6):152–156, June 1988.

[MJ92] Carl McConnell and Ralph E. Johnson. Using static single assignment form in a code optimizer. *ACM Letters on Programming Languages and Systems*, 1(2):152–160, June 1992.

[MR93] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 129–138. ACM, May 1993.

[MSE93] M. Y. Mohd-Saman and D. J. Evans. Investigation of a set of bernstein tests for the detection of loop parallelization. *Parallel Computing*, 19(2):197–207, February 1993.

[MSE95] M. Y. Mohd-Saman and D. J. Evans. Inter-procedural analysis for parallel computing. *Parallel Computing*, 21:315–338, February 1995.

[Mye81] Eugene W. Myers. A precise interprocedural data flow algorithm. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 219–230. ACM, January 1981.

[NPD87] Anne Neirynck, Prakash Panangaden, and Alan J. Demers. Computation of aliases and support sets. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 274–283. ACM, January 1987.

[PB95] Keshav Pingali and Gianfranco Bilardi. APT: A data structure for optimal control dependence computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–46. ACM, June 1995.

[Pin93] Shlomit S Pinter. Register allocation with instruction scheduling: a new approach. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 248–257. ACM, June 1993.

[PP94] Shlomit S. Pinter and Ron Y. Pinter. Program optimization and parallelization using idioms. *ACM Transactions on Programming Languages and Systems*, 16(3):305–327, May 1994.

[PRL91] Hemant D. Pande, Barbara G. Ryder, and William A. Landi. Interprocedural def-use associations in C programs. Technical Report LCSR-TR-162, Laboratory for Computer Science Research, Hill Center for the Mathematical Sciences, Busch Campus, Rutgers University, New Brunswick, New Jersey 08903, April 1991.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, September 1994.

[RL86] John H. Reif and Harry R. Lewis. Efficient symbolic analysis of programs. *Journal of Computer and System Science*, 32:280–314, June 1986.

[RM88] Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 285–293. ACM, January 1988.

[RP88] B. G. Ryder and H. D. Pande. The interprocedural structure of C programs: An empirical study. Technical Report LCSR-TR-99, Laboratory for Computer Science Research, Hill Center for the Mathematical Sciences, Bush Campus, Rutgers University, New Brunswick, New Jersey 08903, February 1988.

[RSY94] James R. Russell, Robert E. Strom, and Daniel M. Yellin. A checkable interface language for pointer-based structures. In *ACM SIGPLAN Proceedings of the Workshop of Interface Definition Languages*, pages 59–73. ACM, August 1994.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22. ACM, June 1995.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 12–27. ACM, January 1988.

[Sed92] Robert Sedgewick. *Algorithmen.* Addison Wesley (Germany) GmbH, 1992.

[SGL97] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems*, 19(2):239–252, March 1997.

[SH97] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to-analysis. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM, January 1997.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233, 1981.

[SRLZ98] Philip A. Stocks, Barbara G. Ryder, William A. Landi, and Sean Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 21–31. ACM, March 1998.

[SSOG93] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22. ACM, May 1993.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23th Annual ACM Symposium on Principles of Programming Languages*, pages 32–41. ACM, January 1996.

[TAFM97] Paolo Tonella, Giuliano Antoniol, Roberto Fiutem, and Ettore Merlo. Flow insensitive C++ pointers and polymorphism analysis and its application to slicing. In *International Conference on Software Engineering*, pages 433–443. ACM, May 1997.

[Tur86] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 83–94. ACM, January 1980.

[Wil97] Robert Paul Wilson. *Efficient, context-sensitive pointer analysis for C programs.* PhD thesis, Stanford University, December 1997.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM, June 1995.

[YRLS97] Jyh-Shiarn Yur, Barbara G. Ryder, William A. Landi, and Phil Stocks. Incremental analysis of side effects for C software systems. In *International Conference on Software Engineering*, pages 422–432. ACM, May 1997.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley Publishing Company, 1991.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–92. ACM, October 1996.