UNIVERSITÄT PASSAU

Fakultät für Informatik und Mathematik

Bachelor's Thesis
in Computer Science

Configurable Pointer-Alias Analysis for CPAchecker

Stefan Weinzierl

Supervisor:

Prof. Dr. Dirk Beyer

October 31, 2016

Eidesstaatliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig und nur unter Zuhilfenahme der ausgewiesenen Hilfsmittel angefertigt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach anderen gedruckten oder im Internet verfügbaren Werken entnommen sind, habe ich durch genaue Quellenangaben kenntlich gemacht.

Passau, den 31. Oktober 2016

____

Stefan Weinzierl

Table of Contents

1. Introduction 5

1.1. Motivation 5

1.2. Outline of this Thesis 6

2. Pointer 6

2.1. Pointer Operations 7

2.2. Types of Pointers 8

2.3. Function Pointers 9

2.4. Arrays 10

2.5. Dynamic Memory Allocation 10

2.6. Pointer to Pointer 10

2.7. Conclusion 11

3. Pointer-Alias Analysis 12

3.1. Definition of Pointer-Alias Analysis 12

3.2. Modeling Pointer Relations 12

3.3. Different Approaches of Pointer-Alias Analysis 14

3.3.1. May vs. Must 14

3.3.2. Intra-procedural vs. Inter-procedural 14

3.3.3. Flow-sensitive vs. Flow-insensitive 14

3.3.4. Context-sensitive vs. Context-insensitive 15

3.3.5. Other Configurations 16

3.3.6. Conclusion 17

3.4. Outlining the Pointer-Alias Algorithms for CPAchecker 18

3.4.1. Andersen's Pointer Analysis 18

3.4.2. Steensgaard's Pointer Analysis 20

4. Automated Software Verification in Respect of Configurable Program Analysis and CPAchecker 24

4.1. Program Analysis and Model Checking 24

4.1. Control Flow Automaton 25

4.2. Configurable Program Analysis 25

4.3. CPAchecker 27

4.3.1. Architecture regarding CPA 27

4.3.2. CPA Algorithm 29

4.3.3. Evaluation 30

4.3.4. Explicit-Value Analysis 31

5. Applying Configurable Pointer-Alias Analysis in CPAchecker 32

5.1. PointerCPA 32

5.2. Implementing Andersen's and Steensgaard's analysis 33

5.2.1. Extracting Pointer-Relevant Information 33

5.2.2. Points-To Map 33

5.3. Steensgaard's and Andersen's Analyses in CPAchecker 35

5.4. Configurable Pointer-Alias Analysis for Explicit Value Analysis 37

5.5. Conclusion 39

6. Future Work 41

7. Bibliography 42

1. Introduction

There are many program languages that support the use of pointers. In the context of software verification, pointers are extremely finicky to analyze because addresses in memory can be referenced and manipulated arbitrarily. Additionally, many existing analyses require valuable information about pointers in order to stay accurate. At the same time, however, efficiency also needs to be considered to be applicable for large programs. This thesis provides an implementation of configurable pointer-alias analysis for the software verification framework CPAchecker that allows user-dependent application of pointer analysis. Furthermore, the presented implementation is integrated in CPAchecker's explicit-value analysis which is one of its main analyses that benefits from this approach.

1.1.Motivation

In order to highlight the motivation behind this thesis, consider the following exert of a C program. In this thesis, every code sample resolves code in C.

```
int main() {                      //1
          int i, *ptr;            //2
          i = 0;                  //3
          ptr = &i;               //4
          *ptr = incr(i);         //5
          if (i == 1) {           //6
          return(0);              //7
          }                       //8
ERROR:    return(-1);             //9
}                                 //10
int incr(int j) {                 //11
          j = j + 1;              //12
          return(j);              //13
}                                 //14
```

Code Sample 1

CPAchecker's value analysis tracks integer values for each variable and is used as an auxiliary-analysis to determine whether the given input file (code sample 1) satisfies its specification. In line 3 the integer variable `i` is assigned to the value `0`. Since the value analysis does not consider pointers, line 4 and 5 are ignored. Eventually, the condition in line 6's `if`-block will mistakenly be evaluated as `false`, as the analysis still only holds value `0` for `i`. Ultimately, the analysis will end up at the error location, thinking that the given input program in code sample 1 violates the specification. However, when pointers are considered, the analysis then would be able to see that in line 5, the value of `i` is incremented to `1`, due to pointer dereferencing. This time around, the condition in line 6 will be `true` and, consequently, recognize that the specification is met. When an analysis lacks alias information like the one in code sample 1 (`*p` and `i` are aliasing the same memory location) it becomes that more imprecise. On the other hand, alias information must be computed efficiently to be applicable for large programs. This problem motivates not only the inclusion for pointer-alias analysis in CPAchecker's explicit-value analysis, but also emphasizes on the importance of providing different approaches that analyze pointers both accurately and efficiently.

1.2.Outline of this Thesis

To clarify some of the frequently used terms in the later chapters and, to demonstrate what has to be considered when analyzing pointers, we start off by briefly discussing pointers in C. Building up on that, we define and contrast the various different categories of pointer-alias analysis. Then, we provide an overlook of two pointer analyses that serve as the configurations for the later implementation. It is also important to establish familiarity with the environment of CPAchecker. To support that, a short glance at the field of automated software verification is provided, while emphasizing on the components of a configurable program analysis (CPA). Then, CPAs and CPAchecker are introduced, described and evaluated. Finally, the new configurable pointer-alias analysis is introduced and formally described. In conclusion, we discuss the future work to support, extend and enhance this approach in CPAchecker.

2. Pointer

Pointers in C are very powerful. They allow to reference addresses in memory and therefore, make a program more efficient. There are many applications that pointers establish which increases the necessity of analyzing them. This chapter introduces common pointer operations which many pointer-alias analyses are based on and provides a general overview of pointers in C based on the C standard [CS05].

2.1. Pointer Operations

Whenever an object or a variable has to be referenced by its memory address pointers are used. A pointer is a variable that either holds the value `NULL` or a memory address. That address in memory may be the one of another variable, object of reference type (such as structs and unions) or may even be independently allocated during the execution of a program. Since pointers also have types, we additionally use the asterisks symbol ('*') in order to highlight a pointer type[1]. Suppose the following three lines of code:

```
int i, *p;          //1
i = 0;              //2
p = &i;             //3
```

Code Sample 2

In line 1, the integer variable `i` and an integer pointer `p` are declared. When line 1 is executed, the compiler will allocate memory addresses for both variables. In line 3 the address of variable `i` is taken and assigned to the pointer `p` using the ampersand ('&') which is often referred to as the "address-of"-operator. We now say "`p` points to `i`" as an equivalent statement for "the value of `p` is the address of `i`". Figure 2 will finally result in the following naive representation of the memory[2]:

Variable	Memory address	Value
`i`	*1000*	0
`p`	*1004*	*0000*

The operation in line 3 as a whole is called **referencing** because the value of `p` now refers to the address of `i`. Referencing also describes an alias relationship, as in code sample 2 for example, the address '*1000*' can be accessed with `i` or `*p` respectively. When a pointer refers to an

[1] See https://www.tutorialspoint.com/cprogramming/c_pointers.htm
[2] We use cursive numbers to distinguish memory addresses from numeric values.

address, it is possible to use that address for further operations. Consider for example code sample 3 as subsequent lines of code sample 2:

```
int j = *p;           //4
*p = 1;               //5
```

Code Sample 3

The asterisks symbol on the right-hand side in line 4 dereferences the pointer `p`, meaning it returns the value the referenced pointer is pointing to, namely `i`. We speak of dereference-read, since - in the case of line 4 - the dereferenced pointer is taken and assigned to another variable. Therefore, line 4 can be rewritten as '`int j = i;`' which would result in the same outcome. In contrast to that, line 5 introduces dereference-write because we explicitly change the value of the memory address that the pointer is pointing to. Again, we could replace '`*p`' with '`i`' and would get the same result. Note here that dereference-read and dereference-write could both happen in the same line. In general, we talk about **dereferencing**, whenever we would like to get the value of a pointer. Dereferencing is the second pointer operation of three. The last one, being **aliasing**, takes place when a pointer is referenced with another pointer.

```
int *q = p;          //6
```

Code Sample 4

Again, take code sample 4 as subsequent line of code samples 2 and 3. Here in line 6, a new declared pointer variable `q` is referenced not with an explicit memory address but with a pointer. The way this is handled is that it copies the address `p` is pointing to and assigns it as value to the new pointer `q`. Thus, `q` and `p` now both point to the same memory address being the one of `i`. Therefore, `q` and `p` are aliases that point to the memory address '*1000*'.

2.2. Types of Pointers

The type of a pointer is very important because it indicates how to treat its value. Take code samples 2-4 for example. As pointer `p` is dereferenced in line 4, its type is the only indicator for letting the compiler know how to interpret its presented memory address, when the pointer is dereferenced. In this case, since the type of p is int* the memory address that p is pointing to is treated as int. The types a pointer can have range from primitive types like the ones of char and int to more complex ones like arrays or user defined types such as structs and unions, to even a type-less variant using void. We speak of "pointer to int[3]", whenever we mean "int*". Additionally, it is allowed in C to cast pointers to a different type. That way, we just alter the interpretation of the memory address of a pointer, e.g. let p be a pointer to int, who points to an arbitrary integer variable[4]. Then, we cast p to be a pointer to char. The value of the integer variable that p points to, now is interpreted as a char. Pointer casts should be used with care, since altering the interpretation of a memory address can potentially end up to unwanted results.

2.3. Function Pointers

C even allows pointer to functions. Instead of pointing to the address of an object or variable, function pointers contain the address to a function which represents executable code[5]. Function pointers inherent their type from the function they are pointing to. For example, for a function like `int add(int a, int b)` a function pointer to int is required. The syntax of function pointers also requires to indicate the function's parameters. In the example of add, we would declare a corresponding function pointer like this: `int (* fn_ptr) (int, int)`. Pointer operations like referencing and aliasing apply in the expected way. However, when a function pointer is dereferenced one is able to invoke the function that particular pointer is pointing to. If fn_ptr happens to point to the function add, for given integer variables i and j, a function call trough pointer would look like this: `(*fn_ptr)(i, j);` Here we are dereferencing the function pointer to get its value, being the address of function add. Function pointers allow great flexibility on what functions to use but it might not be as transparent anymore as explicitly invoking a function.

[3] int is interchangeable with any other type a pointer can have.
[4] See http://ecomputernotes.com/what-is-c/function-a-pointer/type-casting-of-pointers
[5] See http://www.cprogramming.com/tutorial/function-pointers.html

2.4. Arrays

Suppose an array is declared in C: `int array[];` Internally, arrays are represented as pointers that point to the address of the first member[6]. Thus, the identifier `array` can be used just like any other pointer variable, as all pointer operations apply.

2.5. Dynamic Memory Allocation

As mentioned in chapter 4.1, it is possible in C to dynamically allocate memory for pointers. This is helpful when an exact size needs to be allocated in runtime[7]. Take the following code sample for example:

```
int *p = (int*) malloc(sizeof(int));     //1

free(p);                                 //2
```

Code Sample 5

The function call `malloc` allocates the requested size of memory and returns a pointer to `void` that points to the first byte. Line 1 of code sample 5 will allocate enough memory for an integer variable and return this memory address as `void` pointer. Since, we cast this type from `void*` to `int*` the memory address pointer `p` is pointing to will be interpreted as integer. Every dynamically allocated space in memory has to be freed independently (line 2), using the function `free`.

2.6. Pointer to Pointer

We can even declare a pointer that points to a pointer and so on[8]. Consider code sample 6:

[6] See http://www.peacesoftware.de/ckurs12.html
[7] See http://www.programiz.com/c-programming/c-dynamic-memory-allocation
[8] See https://www.tutorialspoint.com/cprogramming/c_pointer_to_pointer.htm

+

```
int i, *q, **r, ***s;          //1

q = &i;                        //2

r = &q;                        //3

s = &r;                        //4
```

Code Sample 6

In line 4, pointer `s` points to `r` , `r` points to `q` and `q` points to `i`. This concludes in the following figure:

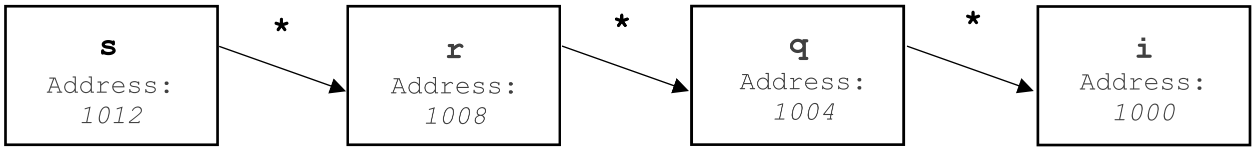

Figure 1 Pointer to Pointer

The labeled arrows illustrate how many times the deference operator (‘*’) must be used to get from one pointer to the desired address, e.g. `***s` refers to the memory address of `i` (*1000*), whereas `*s` refers to the address of `r` (*1008*).

2.7.Conclusion

Pointers in C have a wide range of application which makes them a powerful tool for programming. As powerful as they are however, as dangerous can they be in the context of software verification. Being able to tell what a pointer points to and which memory addresses it aliases becomes very important when trying to analyze the source code of a program, as dereferencing pointer may alter the entire memory in a worst-case scenario. This motivates the field of pointer-alias analysis, covered in the next chapter.

3. Pointer-Alias Analysis

Pointer-alias analysis was introduced for the purpose of tracking and analyzing pointer variables as well as strengthening existing analyses with pointer information. Research found a lot of different approaches, all of them are somewhere on the precision/efficiency scale and thus, have their own application. Every approach in pointer-alias analysis can be classified in several different categories. This chapter introduces the field of pointer-alias analysis while contrasting the different classifications.

3.1. Definition of Pointer-Alias Analysis

Unfortunately, literature is ambiguous about defining pointer analysis. Sometimes, it is distinguished between alias analysis and points-to analysis. Alias analysis is devoted to finding pointer aliases for equal memory locations, whereas points-to analysis determines the possible runtime values of a pointer [A94]. In this thesis, we use the term pointer-alias analysis to describe the task of statically collecting information about pointers. That includes determining runtime values but also inferring alias relationships. To clarify that, suppose the following code sample:

```
int i, *p;          //1
p = &i;             //2
*p = 1;             //3
```

Code Sample 7

When `p` is referenced with `i` in line 2, we say that `p` points to `i`. This information is considered by a pointer-alias analysis. In line 3, we then infer from the previous line that `*p` aliases `i`. So, while basically performing the task of a points-to analysis[9] we consider the inferred aliases, too.

3.2. Modeling Pointer Relations

The first step that needs to be addressed when analyzing pointers is modeling their points-to relationships. After all, the applied pointer analysis is desired to be used as auxiliary analysis for other software verification approaches. Therefore, a representation is desired that approximates

[9] Most points-to analyses consider aliasing but we would like to highlight this fact.

runtime values of pointers but also can be used to extract alias information from it. Literature differs between three representation classes:

1) **Points-to sets** [A94] represent the points to information of every pointer $[p \rightarrow \{i\}]$. A different representation of the points-to sets used in this thesis is $pts(p) = \{i\}$. This is used to get alias information.
2) **Alias pairs** [A94] describe a set of tuples $(*\,p\,,\ i)$ containing aliases that represent the same location in memory.
3) **Equivalence sets**[10] are sets of aliased memory locations $(*\,p, i)$.

The following code sample will be used to distinguish the three modelling methods from above:

```
int i, j, *p, *q, (*fn_ptr)(int);         //1
int id(int i) {return(i);}                //2
fn_ptr = &id;                             //3
p = &i;                                   //4
q = &j;                                   //5
```

Code Sample 8

The result of each approach is:

1) **Points-to sets:** $[p \rightarrow \{i\}], [fn\_ptr \rightarrow \{id\}], [q \rightarrow \{j\}]$
2) **Alias pairs :** $(*\,p, i), (*\,r, i), (*\,p, *\,r), (*\,q, j)$
3) **Equivalence sets:** $\{*\,p, *\,r, i\}, \{*\,q, j\}$

For larger programs, alias pairs explode rather quickly storage-wise [A94]. Points-to sets have proven to be not only more economical, but also provide alias information that can be inferred from them. Thus, we use points-to sets to describe pointer relations for the different categories covered in the next chapter.

[10] See http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec06-PointerAnalysis.pdf slide 11

3.3. Different Approaches of Pointer-Alias Analysis

The values of pointers may change a lot during the execution of a program. Thus, the analysis tends to get expensive rather quickly which is impeding its scalability. Most pointer analyses tolerate imprecision in order to support the application for larger programs. That imprecision is often expressed as some form of insensitivity. This chapter will contrast the various categories in which pointer-alias analyses are classified.

3.3.1. May vs. Must

The certainty of a pointer-alias analysis is distinguished between may and must analyses which are sometimes referred to as existential and universal analyses [A94]. Both analyses compute a set of abstract locations for every pointer. If there exists at least one program path where a given points-to relation is valid, then the may-analysis will consider it. In contrast, must-analysis requires the given points-to relation to be valid on all program paths. Therefore, may-analysis produces an over-approximation of points-to relations while must-analysis produces an under-approximation.

3.3.2. Intra-procedural vs. Inter-procedural

Another important aspect of pointer analysis is the way of handling function calls. Intra-procedural analysis makes worst-call assumptions about function calls. Therefore, it considers that function calls may alter the state of all variables visible to the procedures[11]. In contrast, inter-procedural is way less conservative in the sense that all function calls are evaluated. That includes checking the function body to see what the computed result is. These approaches are in a way connected with context-(in)sensitivity, discussed in chapter 3.3.4.

3.3.3. Flow-sensitive vs. Flow-insensitive

One way of expressing imprecision in pointer-alias analysis is flow-insensitivity, meaning that the analysis does not take control flow into account. More specifically a flow-insensitive approach assumes that statements of a program can be executed in any order [A94]. Most notably

[11] See http://dragonbook.stanford.edu/lecture-notes/Columbia-COMS-W4117/07-10-16.html

however, is that flow-sensitivity in contrast, computes a points-to relation for every program point, whereas the insensitive approach approximates the set a pointer can point to for the entire program. For demonstration purposes consider the following code sample:

```
int i, j, *p;         //1
p = &i;               //2
p = &j;               //3
```

Code Sample 9

A flow-insensitive pointer-alias analysis does not consider that line 2 is executed before line 3. Therefore, it inaccurately merges both possibilities in the computed points-to set $[p \rightarrow \{i, j\}]$. The points-set considering flow-sensitivity is empty for line 1. For lines 2 and 3, it is $[p \rightarrow \{i\}]$ and $[p \rightarrow \{j\}]$, respectively. So, considering the control flow, the analysis approximates at each program point what a pointer points to.

Traditionally, flow-sensitive analyses are very slow which hinders their scalability[12]. Furthermore, the effectiveness of flow-sensitive information turns out to be not as impactful in C [A94], making the more expensive approach less appealing.

3.3.4. Context-sensitive vs. Context-insensitive

This distinction considers calling context of a function. While intra- and inter-procedural approaches analyze how the function is evaluated (worst-call assumption or actual evaluation), context-sensitivity describes how these results are being treated for each calling context. As intra-procedural is very inaccurate, this chapter will only outline the behavior of calling contexts based on an inter-procedural analysis.

Being context-insensitive in pointer analysis means that function-calls are propagated from one call to another [WL04]. Suppose two pointers to interger `q`, `p` are given for the following code sample.

[12]Research found a way for flow-sensitive pointer-alias analysis to be applicable for millions of lines of code. See: https://www.cs.ucsb.edu/~benh/research/papers/hardekopf11flow.pdf

```
int i, j;           //1
p = id(&i);         //2
q = id(&j);         //3
```

```
int* id(int* r) {
        return (r);
}
```

Code Sample 10

A context-sensitive pointer analysis returns a computed result for every call site. Thus, the function call in line 2 returns value `&i` and in line 3, it returns `&j`. For every other call of the function `id` (or any other given function), a context-sensitive pointer analysis will return the appropriate result. The counterpart, being a context-insensitive pointer analysis, returns one result applicable for every call site. Every time `id` is being called, the result will be merged with the previous one, possibly creating ambiguous values for the return variable `r` in `id`. Applied on the code sample above, context-insensitive pointer analysis will mistakenly conclude that `id(&i);` may also return value `&j`. Therefore, the analysis suffers from the unrealizable path problem which describes program paths that cannot occur in a real execution of the input program [W97]. That imprecision is critical factor for speeding up the context-insensitive approach. On the other hand, filtering out unrealizable paths is what makes context sensitivity more precise but also less efficient.

3.3.5. Other Configurations

There are some minor considerations of (in-)sensitivity that can be considered for analyzing pointers. C especially offers a lot of smaller configurations because of the numerous possibilities to create and deal with pointers (see chapter 2).

One problem that can be considered in pointer analysis, is how to deal with structures and unions. In C, structures and unions are user-defined, making them a sophisticated tool for independent applications. One way of dealing with pointers to structs and unions is to unify them by type. Suppose a struct type is defined as follows: `struct Person { int age; } p1, p2;` Now we define a pointer to the struct-type `Person` via: `Person *ptr1 = &p1;` With the insensitivity defined above, we are not able to track individual variables. Therefore, the resulting points-to set would inaccurately be $[\, ptr1 \rightarrow \{Person\}\,]$. Suppose a second pointer ptr2 to Person is declared and assigned to the address of p2. Again, the declared pointer analysis would not be able to tell the difference between ptr1 and ptr2, as second points-to set $[ptr2 \rightarrow \{Person\}]$ is equal to the first one. Hereof, we exploit the fact that struct (and union) variables of the same type may flow together, whereas a different struct-types cannot flow together [A94]. When analyzing programs that contain excessive use of struct and union variables, this may not be desired. It would be more accurate to track pointers to their actual struct-variables instead of unifying them by type. Thus, the more precise approach would compute the points to sets $[ptr1 \rightarrow \{p1\}]$ and $[ptr2 \rightarrow \{p2\}]$.

Another aspect of how to configure pointer analysis is field-sensitivity. Field-sensitivity describes the functionality of keeping track of every field member of reference type [A13]. Again, using structures as example, consider the following code sample:

```
int i, j;                          //1
struct S {int *p; int *q;} st;     //2
S *ptr = &st;                      //3
ptr->p = &i;                       //4
ptr->q = &j;                       //5
```

Code Sample 11

The pointer dereferences happening in line 4 and 5, access field members p and q of st, respectively. Using field-sensitivity, a pointer analysis would differentiate between its field members, creating the corresponding points-to sets $[\,S\; st.p \rightarrow \{i\}\,]$ and $[\,S\; st.q \rightarrow \{j\}\,]$. In contrast, a field-insensitive pointer analysis will result in the points to set $[\,S\; st \rightarrow \{i,j\}\,]$, inaccurately implying that p may point to j or q may point to i.

3.3.6. Conclusion

There are many possibilities of designing pointer-alias analyses. Emphasizing on correct results for example, a flow- and context-sensitive approach would be advisable. However, these sensitivities are very inefficient separately, let alone combined. In order to support the application for larger programs and therefore, be more of practical use, it is required to tolerate imprecision. This motivates the further investigation of the two flow-insensitive approaches introduced in the next chapter.

3.4. Outlining the Pointer-Alias Algorithms for CPAchecker

In the last chapter, we introduced several categories that are used to classify pointer-alias analyses. As in any other analyses in software verification, there is the obvious trade-off between precision and efficiency. When examining the field of flow-insensitive pointer-alias analyses, there are two fundamental approaches that are frequently mentioned in literature. Steensgaard's pointer analysis is notorious for being very fast, supporting virtually unlimited scalability. Andersen's approach is much more accurate while sacrificing efficiency. Since these two approaches form a perfect foundation for a configurable pointer analysis, as they cover both extreme points, accuracy and efficiency, we chose to implement them in CPAchecker. This chapter provides a generic outline of the algorithm's functionality. Later in chapter 5, we will describe their formal implementation in CPAchecker.

3.4.1. Andersen's Pointer Analysis

Andersen's pointer analysis is a flow- and context-insensitive, inter-procedural may-pointer-alias analysis based on subset constraints. The analysis is basically structured in two phases: Generating a constraint system of a given input program and solving it afterwards. Pointer information will be represented as points-to sets (see chapter 3.2) which are inferred from the constraint system. The constraint rules are classified by pointer operations (see chapter 2.1) and defined as follows[13]:

1) **Referencing:** $p = \&i; \Longrightarrow i \in pts(p)$
2) **Dereferencing-read:** $p = *q; \Longrightarrow \forall x \in pts(q): pts(x) \subseteq pts(p)$
3) **Dereferencing-write:** $*p = q; \Longrightarrow \forall x \in pts(p): pts(q) \subseteq pts(x)$
4) **Aliasing:** $p = q; \Longrightarrow pts(q) \subseteq pts(p)$

[13]See http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec06-PointerAnalysis.pdf slide 15

The first rule states that whenever a pointer is referenced using the &-operator, a base-constraint is generated. Therefore, we know that the points-to set of p contains at least the address of variable i. Since the analysis is flow-insensitive, this approximation will hold for all program points. The second and third rule express a complex constraint and indicate that a pointer is dereferenced. In case of dereference-read (rule 2), the left-hand side becomes a superset of all points-to sets that the dereferenced pointer is pointing to. On the other hand, dereference-write (rule 3) implies that the all points-to sets of the dereferenced pointer on the left-hand side become a superset of the right-hand side's points-to set. The last rule considers aliasing and generates a base constraint. The points-to set of the pointer on the left becomes a superset of the points-to set of the pointer on the right.

The following code sample exemplifies how the constraint system is constructed:

```
int *i, *p, *q, **r, **s, **t;//1
p = &i;                       //2
r = &p;                       //3
s = &q;                       //4
*s = p;                       //5
s = *r;                       //6
t = r;                        //7
```

$$i \in pts(p)$$
$$p \in pts(r)$$
$$q \in pts(s)$$
$$\forall x \in pts(s): pts(p) \subseteq pts(x)$$
$$\forall x \in pts(r): pts(x) \subseteq pts(s)$$
$$pts(r) \subseteq pts(t)$$

Code Sample 12

We infer from the constraints on the right the following points-to sets:

$$i \in pts(p) \Longrightarrow [p \rightarrow \{i\}]$$
$$p \in pts(r) \Longrightarrow [r \rightarrow \{p\}]$$
$$q \in pts(s) \Longrightarrow [s \rightarrow \{q\}]$$
$$\forall x \in pts(s): pts(p) \subseteq pts(x) \Longrightarrow pts(p) \subseteq pts(q) \Longrightarrow [q \rightarrow \{i\}]$$
$$\forall x \in pts(r): pts(x) \subseteq pts(s) \Longrightarrow pts(p) \subseteq pts(s) \Longrightarrow [s \rightarrow \{q, i\}]$$
$$pts(r) \subseteq pts(t) \Longrightarrow [t \rightarrow \{p\}]$$

Consequently, the resulting points-to sets of Andersen's analysis are computed from the generated constraints and defined as follows:

$$[p \rightarrow \{i\}], [r \rightarrow \{p\}]\ , [q \rightarrow \{i\}], [s \rightarrow \{q, i\}], [t \rightarrow \{p\}]$$

The actual values of each pointer in line 7 are denoted as:

$$[p \rightarrow \{i\}], [r \rightarrow \{p\}], [s \rightarrow \{i\}], [q \rightarrow \{i\}], [t \rightarrow \{p\}]$$

Comparing Andersen's results with the actual values of the pointers of code sample 12, we conclude that Andersen is pretty accurate, only computing the additional approximation that pointer `s` may also point to `q`. However, the more aliasing or dereferencing there is in the input program, the less precise the results will get due to the corresponding constraints which continuously propagate the according subsets. Furthermore, we evaluate the efficiency of the algorithm. There are $O(n)$ constraints that can be generated (given the size input program n). In a worst-case scenario, each pointer variable points to every other variable of the program. For an aliasing constraint that means that there are $O(n)$ base constraints propagated from one pointer to another, resulting in a runtime of $O(n^2)$. However, we have not considered dereferencing yet. In case of dereference-read, the dereferenced pointer (right-hand side) potentially propagates $O(n)$ subset constraints to every element (in worst case $O(n)$) that the referenced pointer (left-hand side) is pointing to. Therefore, the algorithm runs in $O(n^3)$ in the worst-case[14] [A13]. Later in chapter 5, we implement Andersen's pointer analysis through creating and maintaining an auxiliary-graph that solves the generated constraints.

3.4.2. Steensgaard's Pointer Analysis

Because Andersen's implementation is still, despite being flow- and context-insensitive, considered to be too slow for larger programs, as its runtime is cubic in the worst-case, this chapter presents an even faster approach to pointer analysis, originally proposed by Bjarne

[14] Further research (Manu Sridharan, Stephen J. Fink, *"The Complexity of Andersen's Analysis in Practice"*, Watson Research Center, 2009) of Andersen's approach evaluated, that the algorithm, in practice, runs in $O(n^2)$ time, if the program has only restricted amount of dereferences and operates on a sparse flow graph.

Steensgaard. His algorithm is classified as flow- and context-insensitive, inter-procedural may-pointer analysis that is based on equality constraints.

Similar to Andersen's pointer analysis, the algorithm's foundation is generating and solving constraints. However, as oppose towards Andersen's approach, the constraint system in Steensgaard's pointer analysis is unification based. Meaning, that in case of aliasing, pointer information flows bidirectional. That way, the analysis gains imprecision while at the same time speeding up the analysis in contrast to Andersens'. Steensgaard's analysis runs almost in linear time [S95]. In fact, it runs in $O(n\alpha(n,n))$ time, where n is the size of the input program and $\alpha(n,n)$ is the reverse Ackermann's function (that for large n, is close to being constantly 1). That creates the problem of representing the result in linear space. Since worst case assumption about a pointer is, that it points to every other variable in the program, for n pointers it will occupy $O(n^2)$ space. This is solved by restricting variables to have constant space. That space is expressed as abstract locations. Every variable, including pointers, identify an abstract location at the beginning of the analysis. Each abstract location that contains a pointer is restricted to point to maximum one other abstract location. When a pointer may point to two different memory locations during the execution of the algorithm, then these abstract locations will be joined. If other abstract locations happen to point to one of the joined abstract locations, then they will be updated to point to the new joined abstract location. Furthermore, when joining two abstract locations that contain pointers and thus also point to their respective abstract location, then their abstract locations have to be joined, too. This procedure must be repeated until the condition holds again. We define the following constraint rules:

1) **Referencing:** $p = \&i; \Longrightarrow join(*p, i)$
2) **Dereferencing-read:** $p = *q; \Longrightarrow join(*p, **q)$
3) **Dereferencing-write:** $*p = q; \Longrightarrow join(**p, *q)$
4) **Aliasing:** $p = q; \Longrightarrow join(*p, *q)$

The method $join(p1, p2)$ for abstract locations $p1$ and $p2$ is defined as follows:

```
join(p1, p2)
        if (p1 == p2)
                return
        p1next = *p1;
        p2next = *p2;
        unify(p1, p2)
        join(p1next, p2next)
```

Figure 2 taken from [A13]

Thus, only two distinct abstract locations are joined. Furthermore, the respective abstract locations that the two joined locations `p1` and `p2` may point to, are also joined. The recursive call indicates that this is repeated until there are no locations left to join. The method `unify` takes care of the actual unification of two abstract locations and handles the dependencies that need to be updated.

The first rule states that referencing pointer p with address i joins the two abstract locations $*p$ and i. Suppose pointer p already points to an abstract location containing j. This means, that after $join(*p, i)$, the abstract location pointer p is pointing to, now contains i, too. Furthermore, every pointer that pointed to an abstract location containing either i or j, now points to the new abstract location containing (at least) both. Joining abstract locations and updating other ones that may point to one of them is the main source of the analysis's imprecision. At the same time, it is the reason for its fast performance. The second and third rules follow the same structure but consider the respective dereferences. Suppose again code sample 12 (from chapter 3.4.1) as an example. The following constraints are generated and transformed into points-to sets, for a more descriptive representation:

$$join(*p, i) \Longrightarrow [p \rightarrow \{i\}]$$
$$join(*r, p) \Longrightarrow [r \rightarrow \{p\}]$$
$$join(*s, q) \Longrightarrow [s \rightarrow \{q\}]$$
$$join(**s, p) \Longrightarrow [q \rightarrow \{i\}]$$
$$join(*s, **r) \Longrightarrow [s \rightarrow \{q, i\}]$$
$$join(*t, *r) \Longrightarrow [t \rightarrow \{p\}]$$

Though above representations of the points-to sets are correct for their respective situations, there is one aspect to look out for. When $join(*s, **r)$ is called, the abstract locations represented by i and q have to be joined because the abstract location s can only point to one other abstract location. The method `unify` then propagates the new joined location to every abstract location that pointed to either i or q. Thus, the resulting points-to sets of Steensgaard's analysis are: $[p \rightarrow \{q, i\}], [r \rightarrow \{p\}], [s \rightarrow \{q, i\}], [q, i \rightarrow \{q, i\}], [t \rightarrow \{p\}]$. Compared to the actual values ($[p \rightarrow \{i\}], [r \rightarrow \{p\}], [s \rightarrow \{i\}], [q \rightarrow \{i\}], [t \rightarrow \{p\}]$) it is discernible that, even for a small program like the one in code sample 12, the analysis's precision is very poor. But the virtual unlimited scalability, due to almost linear runtime, is undeniably beneficial for many applications.

4. Automated Software Verification in Respect of Configurable Program Analysis and CPAchecker

To find application for the previously introduced algorithms, it is important to clarify the background and formalities of CPAchecker. Software verification is a broad topic that is devoted to checking if software satisfies a given specification [C88]. Since modern society relies a lot on correctly working software, it is of no surprise that software verification is a heavily researched field in computer science. For readability's sake, we will narrow this topic down to automated software verification. Generally, there are two main categories of automated software verification. Dynamic verification deals with analyzing a program's behavior by executing its code in an exhaustive and failure-provoking fashion. Static approaches deal with the task of checking if the given requirements for a program are met without utilizing the execution of its code. In respect of configurable program analysis which is introduced later, this chapter only introduces two static approaches, namely program analysis and model checking, each of which serve as the foundation of many automated software verification tools.

4.1. Program Analysis and Model Checking

Program analysis is a method of statically collecting information about a given input program in order to highlight its behavior. Program analyses are concerned with efficiency and effectiveness [BHT07]. Therefore, most program analyzers are path-insensitive, meaning program states with equal execution paths are merged. This imprecision is tolerated for the sake of speeding up the analysis. Typical applications of program analyses are bug-finding and compiler optimization. (CITIT)

"**Model checking** is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model" [BK08]. For that, model checking requires two important components: the desired model to check and the given specification. Typically, the system model is represented as finite state graph or the inferred reachability tree. Using exhaustive exploration of the system model, each reached system state will be examined whether the given property specification is satisfied or not. Model checkers, while clearly emphasizing on correctness, are very expensive in practice and thus, focus more on smaller programs [BHT07].

Since model checking and program analysis both express static software verification approaches and due to further examination of the relation between the two, a possibility was found to express them both in a single formal setting. This setting was introduced as configurable program analysis (CPA). The goal of combining them is to benefit both components: model checking would benefit from merging some states and thus, be applicable for larger programs while program analyses would improve their accuracy by merging less states [BHT07]. Before defining a configurable program analysis in detail, we describe a control flow automaton as it is the main data structure used by CPAs.

4.1.Control Flow Automaton

A control-flow automaton (CFA) is a possible way of representing a program. A CFA [BHT07] defines a graph which consists of L nodes modelling the program counter (pc), an initial node pc_0 representing the entry point of a program and a set of edges $G \subseteq L \times Ops \times L$ defining the control flow from one node to another. A concrete state assigns to each variable from $X \cup \{pc\}$ a value. Each edge $g \in G$ defines a (labeled) transfer relation $\overset{g}{\to} \subseteq C \times \{g\} \times C$, whereas C is the set of all concrete strates. We obtain the complete transfer relation $\to$ by unifying all edges: $\to\, = \bigcup_{g \in G} \overset{g}{\to}$. We call a concrete state c_n reachable from region r, if there exists a sequence of concrete states $\langle c_0,\ c_1, \dots, c_n \rangle$ with $c_0 \in r$ and for all $1 \leq i\ \leq n$, holds $c_{i-1} \to c_i$. Reachability of concrete state c_n from r is denoted by $c_n \in Reach(r)$.

4.2.Configurable Program Analysis

A configurable program analysis[15] $\mathbb{D} = (D, \rightsquigarrow, \mathrm{merge}, \mathrm{stop})$ [BHT07] is defined through four components which will be introduced separately.

[15] Continuing research of that field even introduced an enhanced configurable program analysis that allows dynamic precision adjustment (CPA+). However, the core of CPA still preserves and may be more convenient for describing the concept of CPA.
For more information, read: http://www.sosy-lab.org/~dbeyer/Publications/2008-ASE.Program_Analysis_with_Dynamic_Precision_Adjustment.pdf

1) The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ contains a set of concrete states C, a semi-lattice $\mathcal{E}$ and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consists of five components: a set of (possibly infinite) elements E which represent the abstract states of the domain, a top element $\top \in E$, a bottom element $\bot \in E$, a preorder $\sqsubseteq \subseteq E \times E$ and a total function $\sqcup$: $E \times E \to E$, representing the join operator of the semi-lattice. Finally, the concretization function $[\![\cdot]\!] : E \to 2^C$ assigns each abstract state to its corresponding set of concrete states. For soundness, the abstract domain has to meet the following requirements:
   a) $[\![\top]\!] = C$ and $[\![\bot]\!] = \emptyset$
   b) $\forall e, e' \in E : [\![e]\!] \cup [\![e']\!] \subseteq [\![e \sqcup e']\!]$ (the join operator is precise or it over-approximates)
2) The transfer relation $\rightsquigarrow \subseteq E \times G \times E$ assigns for a given CFA edge $g \in G$ all possible subsequent abstract states $e'$to every abstract state $e \in E$. We write $e \overset{g}{\rightsquigarrow} e'$ if $(e, g, e') \in \rightsquigarrow$, and $e \rightsquigarrow e'$ if there is a CFA edge g with $e \overset{g}{\rightsquigarrow} e'$. For soundness, the transfer relation is bound to fulfill the following requirements:
   a) $\forall e \in E : \exists e' \in E : e \rightsquigarrow e'$ (totality of transfer relation)
   b) $\forall e \in E, g \in G : \bigcup_{c \in [\![e]\!]} \{c' \mid c \overset{g}{\to} c'\} \subseteq \bigcup_{e \overset{g}{\rightsquigarrow} e'} [\![e']\!]$ (over-approximation of operations)
3) The merge operator $\mathrm{merge} : E \times E \to E$ merges two abstract states and their information. The requirement for soundness is:
   a) $\forall e, e' : e' \sqsubseteq \mathrm{merge}(e, e')$.

      That way, the result of merge can only be equally or more abstract than the second parameter. We infer two aspects: the result of merge can only be between e' and $\top$ and secondly, merge is not commutative. The two merge-operators covered in this thesis are: $\mathrm{merge}^{sep}(e, e') = e'$ and $\mathrm{merge}^{join}(e, e') = e \sqcup e'$. Note that merge differs from the lattice's join operator $\sqcup$ but can be based on it, as seen in the latter example merge^{join}.
4) The termination check $\mathrm{stop} : E \times 2^E \to \mathbb{B}$ checks if the abstract state (first parameter) is covered by the set of abstract states (second parameter). We require for soundness:
   a) $\forall e \in E, R \subseteq E : \mathrm{stop}(e, R) = true \implies [\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$

      Therefore, the termination check is able to search for an element that subsumes the first parameter e.

4.3.CPAchecker

CPAchecker is an open source software verification tool based on the concept of configurable program analysis (see chapter 3.2). Therefore, CPAchecker is neither model checker nor program analyzer, it is a combination of both. The framework is written in Java and focuses on the verification of C programs. CPAchecker was used to e.g. successfully spot bugs in Linux kernel[16] and is most famous for its flexible environment. Furthermore, CPAchecker is a regular competitor in the competition on software verification[17], winning several awards[18]. This chapter provides an internal look at the verifier's underlying architecture, the functionality of its main algorithm as well as an evaluation of the platform. Finally, we propose how configurable pointer-alias analysis can be used to benefit one of the main analyses in CPAchecker.

4.3.1. Architecture regarding CPA

Since the goal of CPAchecker is to combine various approaches of model checking and program analysis, it is required to provide a formulism for such. Thus, the composite pattern was integrated, allowing to use multiple CPAs for the main algorithm.

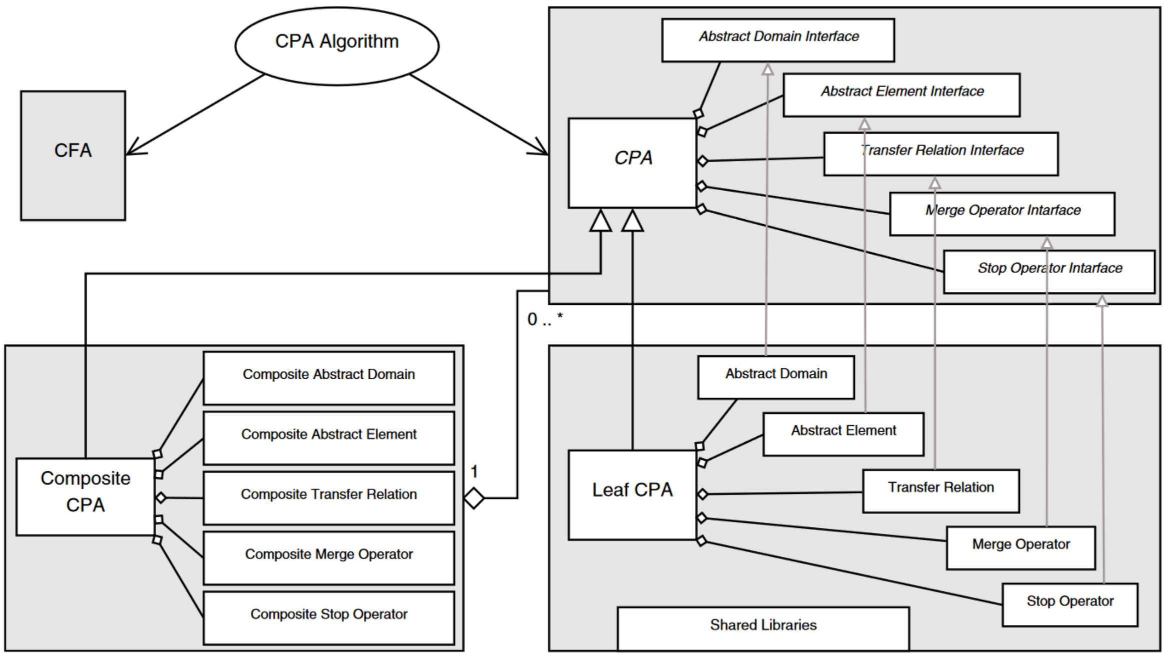

Figure 3 Structure of CPA in CPAchecker (taken from [BK11])

[16] See http://cpachecker.sosy-lab.org/achieve.php
[17] See http://sv-comp.sosy-lab.org/2017/
[18] See http://cpachecker.sosy-lab.org/achieve.php

Figure 1 demonstrates the internal structure for configurable program analyses in CPAchecker. The eponymous CPA (see chapter 4.2) defines an abstract data structure that may consist of a composite CPA (when the usage of more than one CPA is desired) or of a leaf CPA (used for representing a single CPA). A composite CPA simply expresses the combination of other existing CPAs. There are two requirements for integrating a new CPA [BK11]. First, the CPA desired to be added has to be registered in the global properties file and second, the new CPA must implement the CPA interface alongside with all required methods.

A composite CPA $C = (\mathbb{D}_1, \mathbb{D}_2 \rightsquigarrow_\times, \mathrm{merge}_\times, \mathrm{stop}_\times)$ consists of a finite number of CPAs as well as the composite transfer relation $\rightsquigarrow_\times$, composite merge operator $\mathrm{merge}_\times$ and composite termination check $\mathrm{stop}_\times$. For simplicity reasons, we only use two CPAs $(\mathbb{D}_1, \mathbb{D}_2)$ but theoretically, the following applies to any finite number of CPAs. Additionally, two new operators $\downarrow$ and $\preccurlyeq$ are introduced [BHT07]:

1) The strengthening operator $\downarrow: E_1 \times E_2 \rightarrow E_1$ strengthens abstract states from the lattice set $E_1$with information from lattice elements of E_2. The requirement $\downarrow(e, e') \sqsubseteq e$ must hold.
2) The comparison operator $\preccurlyeq \subseteq E_1 \times E_2$is used to compare elements from different lattices.

The three composites $\rightsquigarrow_\times, \mathrm{merge}_\times$ and $\mathrm{stop}_\times$ as well as the strengthening operator $\downarrow$ and the comparison operator $\preccurlyeq$ are expressions over the components $\mathbb{D}_1$ and $\mathbb{D}_2$ $(\rightsquigarrow_i, \mathrm{merge}_i, \mathrm{stop}_i, [\![\cdot]\!]_i, E_i, \top_i, \bot_i, \sqsubseteq_i, \sqcup_i)$.

For the sake of completeness, we state how a composite CPA $C = (\mathbb{D}_1, \mathbb{D}_2, \rightsquigarrow_\times, \mathrm{merge}_\times,\ \mathrm{stop}_\times)$ is constructed to a configurable program analysis $\mathbb{D}_\times = (D_\times, \rightsquigarrow_\times, \mathrm{merge}_\times, \mathrm{stop}_\times)$, using the following rules [BHT07]:

1) The product domain $D_\times$ is simply defined as the direct product of each domain from the CPAs. $D_\times = D_1 \times D_2 = (C, \mathcal{E}_\times, [\![\cdot]\!]_\times)$. The product lattice is constructed accordingly $\mathcal{E}_\times = \mathcal{E}_1 \times \mathcal{E}_2 = (E_1 \times E_2, (\top_1, \top_2), (\bot_1, \bot_2), \sqsubseteq_\times, \sqcup_\times)$ with $(e_1, e_2) \sqsubseteq_\times (e_1{}', e_2{}')$ iff $e_1 \sqsubseteq_1 e_1{}'$ and $e_2 \sqsubseteq_2 e_2{}'$ and $(e_1, e_2) \sqcup_\times (e_1', e_2') = (e_1 \sqcup_1 e_1', e_2 \sqcup_2 e_2{}')$.

2) Finally, the product concretization function $[\![\cdot]\!]_\times$ is constructed such that $[\![(d_1, d_2)]\!]_\times = [\![d_2]\!]_\times \cap [\![d_2]\!]_\times$ holds.

4.3.2. CPA Algorithm

The main CPA algorithm (see below) operates on a source code file written in C and aims to perform a reachability analysis. For this, the algorithm computes for a given CPA and an initial abstract state, a set of reachable concrete state that is over-approximated [BHT07]. This algorithm can be configured by including different software verification approaches expressed as CPAs. Hereof, all four components of a CPA are relevant for the algorithms precision and performance.

Before the analysis starts, the input file is transformed into a syntax tree and further into control-flow automata. The set of CFA and a CPA which is likely to be a composite CPA are the main components for the algorithm (see figure 5). Based on its CPA and an additional initial abstract state e_0, a set of reachable abstract states are computed. For this, two sets of abstract states are maintained: $\mathrm{reached}$, representing all reachable abstract states and waitlist containing the abstract states that have not been processed yet. The transfer relation of the CPA computes for the current abstract state e its successors. The algorithm considers each successor e' of e and combines it with an existing abstract state from $\mathrm{reached}$, using the merge operator. If the merging added new information to the abstract state, the old abstract state will be replaced in $\mathrm{reached}$. In case that the new abstract state is not covered by $\mathrm{reached}$, it will be added to both $\mathrm{reached}$ and $\mathrm{waitlist}$ [BHT07].

The CPA Algorithm $CPA(\mathbb{D}, e_0)$ Taken from [BHT07]

```
Input: a configurable program analysis $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$,
       an initial abstract state $e_0 \in E$, let $E$ denote the set of elements of the semi-lattice of $D$
Output: a set of reachable abstract states
Variables: a set reached of elements of $E$, a set waitlist of elements of $E$
  waitlist := $\{e_0\}$
  reached := $\{e_0\}$
  while waitlist $\neq \emptyset$ do
    pop $e$ from waitlist
    for each $e'$ with $e \rightsquigarrow e'$ do
      for each $e'' \in$ reached do
        // Combine with existing abstract state.
        $e_{new}$ := merge$(e', e'')$
        if $e_{new} \neq e''$ then
          waitlist := (waitlist $\cup \{e_{new}\}) \setminus \{e''\}$
          reached := (reached $\cup \{e_{new}\}) \setminus \{e''\}$
      if $\neg$ stop$(e'$, reached$)$ then
        waitlist := waitlist $\cup \{e'\}$
        reached := reached $\cup \{e'\}$
  return reached
```

4.3.3. Evaluation

There are many model checkers and program analyzers to choose from[19]. The aspects that lets CPAchecker stand out the most is its flexible environment and the possibility to experiment with several combinations of different verification approaches, leading to new and unexpected results. Furthermore, CPAchecker contributes a lot to the reliability and integration of new introduced verification components. Since the field of software verification is heavily researched, there is an overwhelmingly amount of new ideas that need to be considered and evaluated [BK11]. Claims that a new approach outperforms current state-of-the art algorithms lack a lot of reliability because most underlying frameworks use different tools and auxiliary-analyses, making it virtually impossible to compare them to others. For the same reasons, it is hard for third parties to comprehend the full conceptional understanding behind these approaches. Formulizing these ideas as well-structured and sound CPAs in CPAchecker, establishes a common ground with many

[19] For example, the competitor list of 5th International Competition on Software Verification (SV-COMP'16).

other existing analyses which makes it convenient to integrate and compare new approaches to each other and thus, enhance intelligibility and transparency of such.

4.3.4. Explicit-Value Analysis

CPAchecker has many different analyses. One of them is explicit-value analysis, that aims to track integer values for a specified set variables of a given input program. This task requires a composite CPA consisting of a CPA responsible for tracking program locations and another one for explicit values [BL13]. However, the explicit-value analysis does not support pointers which is a major weak spot since pointers are able to change values of variables completely. Therefore, we propose an approach to strengthen the explicit-value analysis with pointer information derived from a new implemented configurable pointer-alias analysis through the strengthening operator $\downarrow$ in the following chapter.

5. Applying Configurable Pointer-Alias Analysis in CPAchecker

This chapter introduces a new CPA called PointerCPA which can be configurated to use either Andersen's (chapter 3.4.1) or Steensgaard's (chapter 3.4.2) pointer analysis. Furthermore, we present how the new CPA is considered by the explicit-value analysis of CPAchecker. The following implementation is specified in CPAchecker revision 23465.

5.1. PointerCPA

We use the specification of a CPA from chapter 4.2 to define the components of the new PointerCPA $\mathbb{D}_p = (D_p, \rightsquigarrow_p, \mathrm{merge}_p, \mathrm{stop}^{SEP})$ as follows:

1) The abstract pointer domain $D_p = \left(C_p, \mathcal{E}_p, [\![\cdot]\!]_p\right)$ consists of a set of concrete pointer states C_p, the semi lattice $\mathcal{E}_p = (E_p, \top_p, \bot_p, \sqsubseteq_p, \sqcup_p)$ and a concretization $[\![\cdot]\!]_p : E_p, \rightarrow 2^{C_p}$. The abstract states E_p maintain a points-to map (see chapter 6.2.1) that maps to each set of pointers[20] a set of memory locations that it may point to. Pointers may also point to an unknown location. The unknown location represents for example a designated initializer for an array, as they are not supported. In this case, a pointer is mapped to point to every possible variable, denoted by $\top$, considering worst-case behavior. When a pointer points to nothing, it is represented by $\bot$. Depending on which configuration is used, the set of abstract states are further segmented in Andersen-states E_A or Steensgaard-states E_S which produce the pointer information based on their respective algorithms. The top element $\top_p$ of lattice $\mathcal{E}_p$ maps all pointer variables to $\top$, the bottom element $\bot_p$ maps all pointers to $\bot$. The preorder $\sqsubseteq_p = E_p \times E_p$ compares two abstract pointer states. More specifically, it checks if the first parameter contains all the pointer information of the second parameter. The join operator $\sqcup_p : E_p \times E_p \rightarrow E_p$ joins the pointer information of two abstract pointer states. Finally, $[\![\cdot]\!]_p : E_p \rightarrow 2^{C_p}$ assigns to each abstract pointer state the set of concrete pointer states that it represents.
2) The transfer relation $\rightsquigarrow_p \subseteq E_p \times G \times E_p$ extracts pointer-relevant information from the given CFA edge and passes it further so that pointer information can be approximated and

[20] In Andersen's analysis, this set contains only one pointer as explained later.

expressed as new abstract state e'. If additional pointer information was added, then: $\exists g \in G: e \overset{g}{\rightsquigarrow}_p e'$.

3) The merge operator can be configured to be either $\text{merge}^{sep}(e, e') = e'$ or $\text{merge}^{join}(e, e') = e \sqcup_p e'$. The latter case is based on the semi lattice's join operator $\sqcup_p$ and therefore, joins two abstract states and their pointer information. In the context of pointer-alias analysis merge^{join} expresses flow-insensitivity.
4) The termination check is defined as $\text{stop}^{sep}(e, R) = (\exists e' \in R: e \sqsubseteq_p e')$, whereas R is a set of abstract pointer states. It is based on the preorder $\sqsubseteq_p$ given by the lattice.

5.2.Implementing Andersen's and Steensgaard's analysis

This chapter describes formally how both algorithms are implemented in CPAchecker using the new defined PointerCPA.

5.2.1. Extracting Pointer-Relevant Information

Both algorithms obtain pointer-relevant information from the pointer transfer relation $\rightsquigarrow_p$. Each CFA edge is checked if it contains a pointer operation. To clarify, the implementation does not support external function calls, such as dynamic memory allocations using malloc. Furthermore, in some special cases such as designated initializers of arrays, worst-case assumptions are made (array points to $\top$). However, referencing variables or objects, as well as all cases of dereferencing and aliasing are supported. The extracted pointer information is then transformed into sets of locations and evaluated (whether, for example it is a $\top$ or $\bot$ element), before it is passed to the abstract domain states (either Andersen-state or Steensgaard-state) where it is registered in a points-to map.

5.2.2. Points-To Map

For the implementation of both pointer-alias analyses, we chose a more convenient way to handle the generated constraints. Therefore, we dynamically create and maintain a points-to map that keeps track of the constraints established by the respective rules. Each key of the map represents a set of locations containing pointers and each value is a set of locations representing what the pointers are pointing to. The points-to map is interpreted as directed graph; each node represents a set of memory locations and its edges indicate a points-to relationship from source (key) to target (value). With this approach, it is not only possible to track approximated runtime values of pointers as key-value pairs, but also extract alias information for other analyses. The latter is achieved by evaluating an according CFA edge in the transfer relation $\rightsquigarrow_p$ which spots pointer dereferencing and returns the values (set of locations) to the corresponding key. Both implementations use the points-to graph with different conditions, thus their formal implementations are introduced separately.

If the PointerCPA is configurated to use Andersen-states (specified as option "Andersen") as abstract states, the algorithm will produce an Andersen-style points-to graph to resolve the constraints. Since the points-to graph records sets of locations that point to other sets of locations, we force the condition that every source of an edge (keys) must only represent one pointer variable. This ensures, that no nodes are merged. From the constraint rules of chapter 3.4.1, we infer the following conditions for the points-to graph:

1) Referencing: $p = \&i;$: Add the memory location of i to the values (set of locations) of key p. Or to put it in perspective as graph: Create an edge from node p to node i.
2) Dereference-read: $p = *q;$: Add the set of locations that $*q$ represents to the values of key p. In the context of the graph: Add an edge from p to each location that the location set $*q$ represents.
3) Dereference-write: $*p = q;$: Add for each location set that the location set $*p$ represents, the location set of q to the respective values. In the graph, add an edge from each node that p points-to, to the nodes that q points to.
4) Aliasing: $p = q;$: Add the location set of q to the value of key p. Or in the graph, add an edge from node p to every node that q points to.

The functionality will be exemplified in chapter 5.3

When the configuration is set to use Steensgaard-states (specified as option Steensgaard) for the PointerCPA, the algorithm computes the according Steensgaard-style points-to graph. In

Steensgaard's pointer analysis, we force that every source node can only have a single target node. Furthermore, every key that contains more than one variable and therefore, is a joined location set of pointers, has to appear as a value, too, because joining indicates that at least one key would point to both values. The constraint rules are applied as follows:

1) Referencing: $p = \&i;$ $: join$ locations $*p$ and i. In the graph, if p points already to something (suppose node j), then $join$ nodes i and j.
2) Dereference-read: $p = *q;$ $: join$ locations p and $*q$. In the graph, $join$ the node that $*q$ is pointing to with the one that p points to.
3) Dereference-write: $*p = q;$ $: join$ locations $*p$ and q. In the graph, $join$ the node that $*p$ points to with the one that q points to.
4) Aliasing: $p = q;$ $: join$ locations p and q. In the graph, $join$ the respective nodes.

In the rules above, $join$ also considers to update the dependencies.

5.3. Steensgaard's and Andersen's Analyses in CPAchecker

We now outline how the specified PointerCPA operates for the following program, written in C:

```
int main() {
          int i, j, *p, *q, **r, **s;    //1
          p = &i;                        //2
          q = &j;                        //3
          r = &p;                        //4
          s = &q;                        //5
          *r = q;                        //6
          *s = *r;                       //7
          r = s;                         //8
          return (0);                    //9
}
```

Code Sample 13

The pointer transfer relation extracts for each edge of the CFA, pointer information. E.g., for line 2, $\rightsquigarrow_p$ will transform both sides of the assignment to the appropriate format that the points-to map will accept. Depending on what abstract states are used, the algorithms will now dynamically create and maintain a points-to map that expresses the points-to relationships as graph. The edges are labeled with the lines of code sample 13 to illustrate how the constraints are translated to the graph:

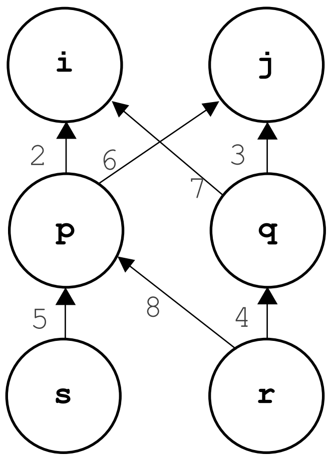

Figure 5 Andersen's points-to graph

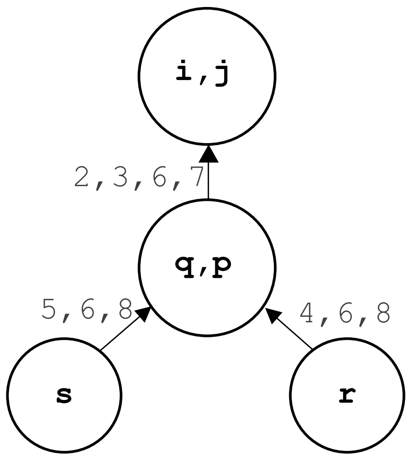

Figure 4 Steensgaard's points-to graph

For every pointer-referencing in Andersen's algorithm, an edge is created from the pointer to its referenced memory location (see edged labeled as `2`, `3`, `4` and `5`). The remaining aliasing operations (dereferencing is a form of aliasing in this case) copy the edge from the right-hand side to the left (see edges labeled as `6`, `7` and `8`) and therefore, become supersets of the respective pointers.

In the result of Steensgaard's analysis, it is harder to spot what each edge contributes to the graph. After the referencing edges (2-5) are processed, the points-to graph of Steensgaard and Andersen are the same. Line 6 introduces aliasing between the locations `p` and `q` because `*r` aliases `p`. Both pointers already point to distinct abstract locations `i` and `j`, respectively. Therefore, the abstract locations have to be merged to one that contains both, represented by the node (**`i,j`**). The points-to relationships of `p` and `q` must then be updated to new joined node. Line 7 introduces no new pointer information because `p` and `q` already point to the same abstract locations (node (`i,j`)). Finally, line 8 aliases pointers `s` and `r`. As both pointers point to different abstract locations `p` and `q`, the nodes are joined to (**`q,p`**). The resulting node points to the unification of each source node's points to set. However, both points-to set are already the same. This concludes the operation for Steensgaard's analysis.

5.4. Configurable Pointer-Alias Analysis for Explicit Value Analysis

To integrate the implementation of the new configurable pointer-alias analysis in the existing explicit-value analysis, a composite CPA is required that uses all necessary CPAs from the value analysis and the new PointerCPA. With the given strengthen operator of the resulting composite CPA, we are able to provide the computed pointer information from the PointerCPA. In the implementation, we consider the following cases:

1) If the left-hand side of an assignment is a dereferenced pointer that aliases tracked variables and if the types are compatible, we update for each such variable its value with the right-hand side's one.
2) If the right-hand side of an assignment is a dereferenced pointer that aliases exactly one tracked variable with explicit-value and the type is compatible to the left-hand side's type, then we update the value of the left-hand side with the new value.
3) If the right-hand side of an assignment is a dereferenced pointer that alias more than one memory location, then the assignment is ambiguous and the value of the left-hand side is forgotten.

Cases 1) and 2) as well as cases 1) and 3) may happen in the same line. To demonstrate the functionality, consider the following code sample:

```
int main() {
int i, j , *p, *q;              //1
i = 0;                          //2
j = 1;                          //3
p = &i;                         //4
q = &j;                         //5
*p = 2;                         //6
*q = *p;                        //7
int k = 42;                     //8
q = &k;                         //9
i = *q;                         //10
*p = k;                         //11
*q = id(k);                     //12
return(0);                      //13
}
```

```
int id(int r) {
return(r);
}
```

Code Sample 14

The values tracked by the value analysis result as follows:

```
i = 0; j = 1; k = 42;
```

For above results, no pointers are considered. Now suppose, that the operating composite CPA of the value analysis also consists of the introduced PointerCPA. Through the strengthening operator ↓: $E_1 \times E_2 \rightarrow E_1$, we now strengthen the abstract states of the value analysis E_1, with the abstract states from our Pointer CPA E_2. Since code sample 14 is a small program, we choose Andersen's analysis in order to be more precise, as the additional efficiency of Steensgaard is not of much use here.

With the first rule defined above, we infer in line 6 for example, that `*p` is aliased with `i`. Therefore, the value of `i` is changed to `2`. In line 7, rules 1) and 2) apply. From the pointer-alias analysis we know, that `*q` is aliased with `j`; hence we change the value of `j` to `2`, too. In line 10, Andersen’s analysis returns that `*q` is ambiguous for the memory addresses of `i` and `j`. Thus, there is no definite assignment to make. The value of `i` is forgotten (rule 3). Line 11 maps the variable `i` to the value of `k`. Line 12 will map both aliases of `*q`, namely `i` and `j`, to the value of `id(k)` (42). In conclusion, with the help of alias information from PointerCPA, the following mapping of values is produced:

```
i = 42; j = 42; k = 42;
```

That is equal to the values an actual execution of the program would have produced. However, we have to point out that this program is not a very common one. Not only is there no use for it (it should serve as demonstration how the applied PointerCPA works in the context of the value analysis), but when executing the program in code sample 14, lines 10 and 11 would both assign value `42` to variable `i`. Therefore, skipping line 11 the strengthen would produce the result

```
j = 42; k =42;
```

while the value of variable `i` is forgotten.

5.5.Conclusion

With the new PointerCPA, it is finally possible to consider pointers when using the explicit-value analysis. Both approaches of pointer-alias analysis design the algorithm to be either precise or efficient. To proof that Steensgaard’s analysis is in fact faster than Andersen’s, or in return, that Andersen’s analysis is more accurate than Steensgaard’s we would have to refer to several benchmarks results that compare both approaches while highlighting the differences. Furthermore, to underline the significant impact of the new implementation, we should compare the benchmark results with a version of CPAchecker that does not support configurable pointer-alias analysis. Unfortunately, this is not covered in this thesis. As a small consolation, we introduce and compare the results of a test task, specified in the framework as “high_degree_of_indirection_true_unreach-label.c”. The code of this test task is seen in the following code sample:

```
void test(int x) {
if (!x) {
ERROR: goto ERROR;
}
void main() {
int x, y, *p1, **p2, ***p3, *q1, **q2, ***q3;
p1 = &x;
p2 = &p1;
p3 = &p2;
q3 = p3;
q2 = *q3;
q1 = *q2;
y = *q1;
test(x == y);
}
```

Code Sample 15

The name of the test-program indicates that in a normal execution of the program, the error label is never reached and therefore, this program should be evaluated as true. Using CPAchecker's value analysis for this, as integer variables are required to be tracked, the test will eventually be false. However, using the same specification with additional configurable pointer-alias analysis, both Steensgaard's and Andersen's algorithm evaluate the given program as true. That is just one example of when the PointerCPA can be used to enhance other analyses.

6. Future Work

With the new PointerCPA one is able to configure pointer-alias analysis relatively precise and very efficiently. But the flexible environment of CPAchecker allows even more inclusions of different-designed pointer-alias analyses, as the ones discussed in chapter 3.3 demonstrate. For example, emphasizing more on precision, approaches that regard flow- and context-sensitivity can be included to get more accurate results. This addresses the problem discussed at the end of chapter 5.4. Additionally, several other smaller configurations can be included, like the support of external function calls and designated initializers of reference typed objects that may find application in some cases and thus, can be individually turned off and on. Furthermore, the integration of the new PointerCPA can be extended in CPAchecker. The predicate abstraction is a good example for such. In addition, to evaluate the new results, we have to perform reliable benchmark tests to see if the theoretical trade-offs can be found in practical results.

7. Bibliography

[A13] Jonathan Aldrich, *“Lecture Notes: Pointer Analysis”*, 15-819O: Program Analysis, 2013

[A94] Lars Ole Andesen, “*Program Analysis and Specialization for the C Programming Language”*, Copenhagen, pages 111-152, 1994

[BHT07] Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz, *“Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis”*, LNCS 4590, pages 504-518, Springer-Verlag, CAV 2007

[BK08] Christel Baier, Joost-Pieter Katoen, *“Principles of Model Checking”*, The MIT Press, 2008, page 30

[BK11] Dirk Beyer, M.Erkan Keremoglu, *“CPAchecker: A Tool for Configurable Software Verification“*, LNCS 6806, pages 184-190, Springer-Verlag, CAV 2011

[BL13] Dirk Beyer, Stefan Löwe, *“Explicit-State Software Model Checking Based on CEGAR and Interpolation“*, LNCS 7793, pages 146-162, FASE 2013

[C88] James S. Collofello, *“Introduction to Software Verification and Validation”*, SEI Curriculum Module SEI-CM-13-1.1, Arizona State University, 1988

[CS05] Committee Draft, *“ISO/IEC 9899:2011”*, The C-Standard, 2011

[S95] Bjarne Steensgaard, *“Points-to Analysis in Almost Linear Time”*, Association for Computing Machinery, 1995

[W97] Robert Paul Wilson, *“Efficient, Context-Sensitive Pointer Analysis For C Programs”*, Dissertation, Stanford University, 1997

[WL04] John Whaley, Monica S. Lam, *“Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams”*, PLDI’04, Washington, 2004