FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Source-Level Interpreter for C Programs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Stephan Krall
Matrikelnummer 0526372

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Dr. Helmut Veith
Mitwirkung: Dipl.-Inf. (FH) Andreas Holzer, M. Sc.

Wien, 07.05.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Source-Level Interpreter for C Programs

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Stephan Krall
Registration Number 0526372

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Dr. Helmut Veith
Assistance: Dipl.-Inf. (FH) Andreas Holzer, M. Sc.

Vienna, 07.05.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Stephan Krall
Strozzigasse 6-8 1.7.099.1.1, 1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

Zum Ende des Studium möchte ich die Gelegenheit ergreifen einigen Personen, die mir mit Rat und Tat zur Seite standen, meinen Dank auszusprechen. Allen voran möchte ich meinen Eltern für ihre unerschütterliche Unterstützung während meiner Studienzeit und darüberhinaus danken. Ein besonderer Dank gilt vor allem meiner Mutter die für alle Probleme immer ein offenes Ohr hatte und zu jeder möglichen und unmöglichen Zeit meine größte Stütze war. Meinem Vater, welcher mir stets mit Ratschlägen zur Seite stand, möchte ich für seine moralische und finanzielle Unterstützung danken. Ebenso möchte ich meinen beiden Brüdern danken, die es in vielen Gesprächen schafften auftretende Selbstzweifel unverzüglich zu beseitigen.

Ein ganz spezieller dank gebührt meinem betreuenden Professor Univ.-Prof. Dr. Helmut Veith, der es mir ermöglichte meine Diplomarbeit am Institut für Informationssysteme zu verfassen, sowie meinem Betreuer Dipl.-Inf. (FH) Andreas Holzer, M. Sc., welcher mir die wissenschaftliche Arbeitsweise näher gebracht hat, stets ein offenes Ohr für Fragen und Probleme hatte und mich im gesamten Verlauf der Diplomarbeit tatkräftig unterstützt hat.

Abstract

The FShell Query Language (FQL) [8, 9] introduced a new formal method for the specification of code coverage criteria. An FQL specification declaratively describes which part of a program shall be covered by a test suite. These code coverage criteria can then be used by an FQL-backend to either generate test suites or measure the coverage of a given test suite. CPA/Tiger is a FQL test generation backend for C-Code, which is based on the formal framework of configurable program analysis (CPA) [3]. An integral component of CPA/Tiger is an interpreter, that can only handle programs, which do not contain complex data structures. Therefore we develop a C-Interpreter which models the heap accurately. The new interpreter is implemented as a CPA which enables a simple integration into CPA/Tiger. The accurate emulation of the heap when interpreting a program is resource consuming and therefore it is not practical to save every program state in an unoptimized data structure. We therefore use the fat node data structure which enables us to save all program states, that are computed by the interpreter. We experimentally evaluate our new interpreter with the existing one to assess the runtime differences between the two interpreters.

Kurzfassung

Die FShell Abfragesprache (FShell Query Language, FQL) [8, 9] erlaubt es deklarativ Code-Abdeckungskriterien zu spezifizieren. Diese Spezifikationen können dann zum einen dazu verwendet werden die durch eine Testsuite erreichte Codeabdeckung zu ermitteln und zum anderen dazu genutzt werden eine abdeckende Testsuite zu erzeugen. CPA/Tiger ist ein FQL Testfallgenerator für C Programme welcher auf dem Konzept der konfigurierbaren Programmanalyse (CPA) [3] basiert. Eine wesentliche Komponente von CPA/Tiger ist ein Interpreter, der nur C-Programme behandeln kann, die keine komplexen Datenstrukturen enthalten. Daher wird ein neuer Interpreter entwickelt, der den Heap eines Programms auf exakte Weise abbildet. Dieser Interpreter wird als statische Programmanalyse realisiert was eine einfache Integration in CPA/Tiger ermöglicht. Die exakte Nachbildung des Heaps über eine Programmausführung hinweg ist resourcenaufwändig und daher ist es nicht praktikabel sämtliche Programmzustände in einer unoptimierten Datenstruktur zu speichern. Die Verwendung der Fat Node Datenstruktur erlaubt die Speicherung aller Programmzustände, welche der Interpreter erzeugt. Der neue Interpreter wird mit dem bereits bestehenden Interpreter experimentell verglichen um die Laufzeitunterschiede der beiden Interpreter abzuschätzen.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Testing is an essential part of every software development process. Test suites are used to verify program behaviour and/or detect errors in programs. Testing has also several problems. One of the problems is that testing is not complete which means that for a complex program it is impossible to test every aspect of the program in limited time. Therefore it is essential to know when to stop testing. Good indicators for this question are code coverage criteria like basic block coverage. If for example a test suite does not meet the basic block coverage criterion then it is probable that we have dead code which is never executed in the program. This can indicate a faulty requirement analysis if the test cases are derived from the requirements.

The tool CPA/Tiger [5] enables the generation or evaluation of test suites which fulfill certain code coverage criteria. These criteria are specified in CPA/Tiger by using the FShell Query Language (FQL) [8] [9] which is a specification language for code coverage criteria. CPA/Tiger is furthermore based on the configurable program analysis (CPA) [3] [1]. An FQL query describes a finite set of test goals. In order to cover a code with respect to an FQL query, a test suite shall match each test goal by a test case specification.

Figure 1.1 gives an overview of the CPA/Tiger work flow. CPA/Tiger derives form an FQL specification test goal automata $A_1, ..., A_n$ where each automaton represents a test goal. Then CPA/Tiger iteratively selects an automaton A_i. A coverage analysis is then performed to check if the existing test case A_i is covered by the test cases which have already been generated by CPA/Tiger. If this is the case then we return to the test goal selection and select the next test goal A_{i+1}. If the test goal is not covered by the existing test cases then we perform a static program analysis. The result of this analysis either shows that a test goal is infeasible or generate a test cases which is then added to the list of existing test cases.

The main purpose of the coverage analysis is to increase the performance of CPA/Tiger. Without a coverage analysis every time a test goal is selected a static program analysis has to be performed which is time consuming. Furthermore the coverage analysis can be used without static program analysis to check if a given test suite fulfills a certain code coverage criterion represented by an FQL query. The coverage analysis consists of an interpreter CPA which is able to interpret C-programs. Currently this interpreter only support integer variables, but no heap manipulation pointer arithmetic, or bit operations. This naturally limits the the applicability of CPA/Tiger and in this thesis an analysis is developed to remedy this situation. Thus if the coverage analysis supported C-programs with complex data structures and complex program constructs (e.g. pointer arithmetic) then the coverage of test suites for these C-programs could be measured.

Therefore we implement a new interpreter CPA for the CPA backend of CPA/Tiger. The interpreter CPA keeps track of the heap and performs bit-precise operations. We implement the interpreter CPA in such a way that it is possible to extend it to allow more advanced data structure analysis. We then experimentally compare the time performance of the new implemented interpreter CPA with the old interpreter CPA in order to evaluate the performance changes to the CPA/Tiger tool. Finally we give an overview of the overall result and discuss possible enhancements to improve the interpreter performance and memory consumption. With the new interpreter CPA it will then be possible to measure the coverage of complex C-programs for FQL code coverage criteria.

1.2 Outline

In Chapter 2 we will discuss the preliminaries for the implementation of the interpreter CPA. We will discuss the program representation in Section 2.1. For the program representation we use CFAs [5]. We will discuss an extension to the current CFAs used by the CPA/Tiger tool to allow the representation of C-Program which contain function pointers. We then give a description of the configurable program analysis framework in Section 2.2. Chapter 3 contains implementation details for the interpreter CPA. We will first outline the features of the interpreter CPA and discuss the memory model (Section 3.1) used by the interpreter. In this Section we will also discuss the problem of the memory consumption when using a simple memory model for the interpreter and present a technique to overcome this problem. We then give a definition of the interpreter CPA in Sections 3.1, 3.2, 3.3. Section 3.2 contains the semantics of the interpreter. Next we give an overview over the related work in Chapter 4. In Chapter 5 we will conduct a series of experiments where we compare the memory and speed performance of our interpreter Implementation with the existing interpreter. Finally, we give an overview of the results and an overview of the possible future work in Chapter 6.

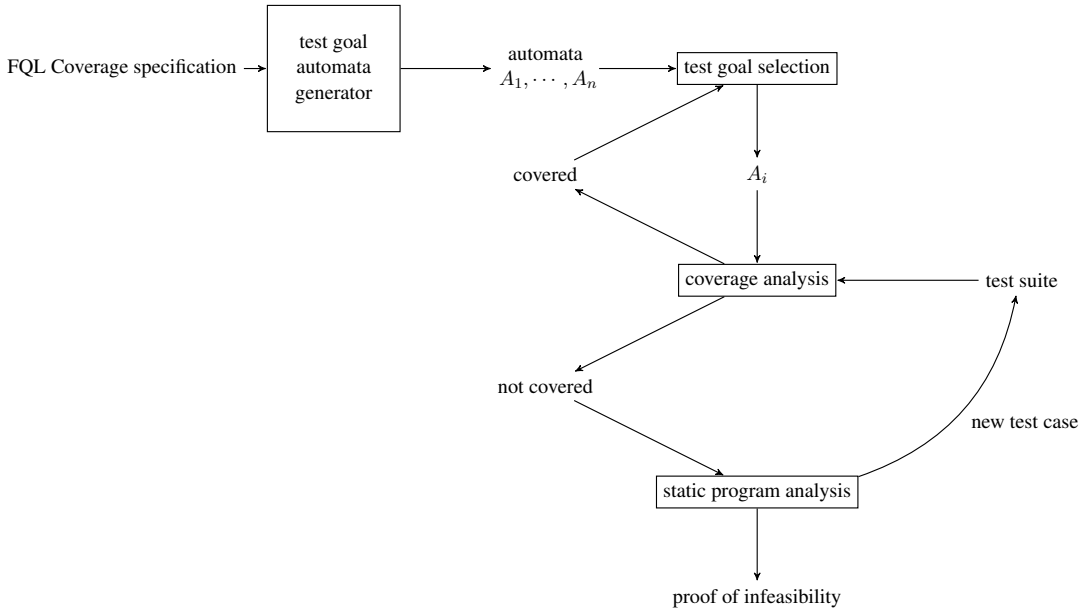


Figure 1.1: Overview of CPA/Tiger workflow

CHAPTER 2

Preliminaries

In this Chapter we give an overview of the concepts and frameworks which are an integral part of CPA/Tiger that we extend. In Section 2.1 we discuss control flow automata which serve as our Program Representation in CPA/Tiger. Section 2.2 introduces the configurable program analysis framework which is the underlying program analysis tool used by CPA/Tiger.

2.1 Program Representation

Before transforming a C program into a control flow automata the program is translated into a subset of the C language the so called C Intermediate Language (CIL) [13]. The purpose of this step is to simplify the program analysis by replacing complex program statements with simpler but semantically equivalent C code. A list of standard translations for CIL can be found in section 4 of the CIL web documentation which is located under http://www.eecs.berkley.edu/ necula/cil. Furthermore we use the flags –save-temps and –dosimplify (c.f. Section 8.15 in the web documentation) on the CIL-Translator to achieve two important types of simplifications for a CIL program.

We represent C programs as control flow automata (CFA) [5] [2]. A CFA (L,G) consists of Nodes L and directed edges $G \subseteq L \times Ops \times L$. Nodes represent program locations. An edge $(\ell, op, \ell') \in G$ is used to describe the transition from a program location ℓ to a program location ℓ' while executing the operation op. The set Ops consist of several types of statements. These are: definition, assign and assume statements of the form assume[c] where c is a condition. In the following subsections we discuss simplifications to the C program and the types of the CFA edges used by the interpreter CPA .

C Intermediate Language

The first type of simplification is the translation of simple field access via the $->$ operator for composite data structures into field access via pointer arithmetic. This allows our program anal-

```c
struct data{
  int data1;
  int data2;
};


int main(){
  int x,y;
  y =3;
  x=9;
  struct data v;
  struct data *pnt;
  pnt = &v;
  pnt->data2 = (4+y)*x;

  return 0;
}
```

```c
/* Generated by CIL v. 1.3.7 */
/* print_CIL_Input is true */
struct data {
   int data1 ;
   int data2 ;
};

int main(void)
{ int x ;
  int y ;
  struct data v ;
  struct data *pnt ;
  unsigned int __cil_tmp5 ;
  unsigned int __cil_tmp6 ;
  int __cil_tmp7 ;
  {
  y = 3;
  x = 9;
  pnt = & v;
  __cil_tmp5 = (unsigned int )pnt;
  __cil_tmp6 = __cil_tmp5 + 4;
  __cil_tmp7 = 4 + y;
  *((int *)__cil_tmp6) = __cil_tmp7 * x;
  return (0);
  }
}
```

Table 2.1: Translation from C-program to CIL-program

ysis to omit dealing with the -> semantics. The other type simplification is the strict usage of three address codes in CIL programs, i.e. any assign-statement in a translated CIL-program is of the form: `result = operand1 operator operand2`. An example of a translation of a C-program which uses the field access operator -> into a CIL-program which uses three address code can be seen in Table 2.1.

Next we are going to describe intra- and inter-procedural CFAs. An intra-procedural CFA represents a single functions whereas a inter-procedural CFA can represent complete C-Programs with several functions. Thus every inter-procedural CFA is also a intra-procedural CFA.

Intra-Procedural CFA

A intra-procedural CFA represents the body of a C function. The intra-procedural CFA has the following types of CFA edges: *Definition*, *Assignment*, *Skip* , *Assume* . The translation from a C

Let ℓ_j be a program location and p be a program statement. Depending on the type of program statement we have the following translations:

Assignment,Definition:
Add a CFA edge e of the following form $e = (\ell_j, p, \ell_{j+1})$.

If:
Let p be of the form `if(cond)`. Add two Assume edges for the program locations ℓ_{true} and ℓ_{false} with $e_1 = (\ell_j, \text{assume [cond]}, \ell_{true})$ and $e_2 = (\ell_j, \text{assume [}\neg\text{cond]}, \ell_{false})$. ℓ_{true} is the program location which is reached in a program execution when the condition of the if statement evaluates to true while ℓ_{false} is the program location which is reached in the program execution when the condition of the if statement evaluates to false. Let ℓ_{j+1} be the program location after the if block. Then we have to add an edge $e_3 = (\ell_t, \text{empty}, \ell_{j+1})$ where ℓ_t is the end location of the then block. Furthermore if there exists an else block for the if statement we have to add the edges $e_4 = (\ell_f, \text{empty}, \ell_{j+1})$ where ℓ_f is the end location of the else block.

while(true):
Add two assume edges similar to an if statement of the form `if(1)`. Furthermore let ℓ_e be the end program location of the while body. Add an edge $e = (\ell_e, \text{empty}, \ell_j)$.

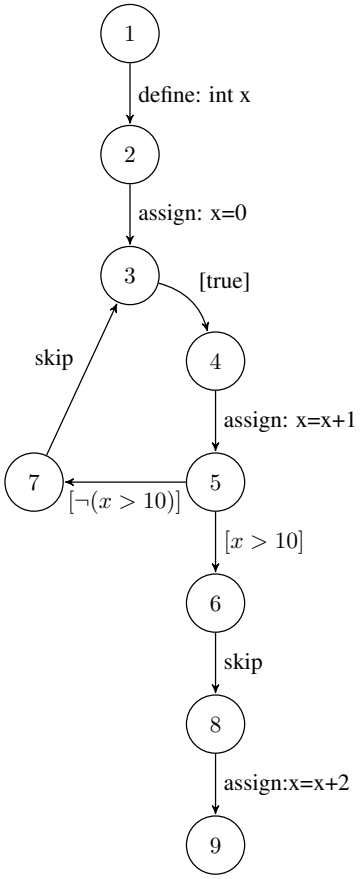
goto n:
Add an edge $e = (\ell_j, \text{empty}, n)$.

Figure 2.1: Overview over translations for conversion from C function to intra-procedural CFA

function to an intra-procedural CFA is defined in Figure 2.1.An example of an intra-procedural CFA can be seen in Figure 2.2.

Inter-Procedural CFA

The inter-procedural CFA extends the intra-procedural CFA by adding 3 types of CFA edges: *Function Call*, *Function Exit*, *Function Return*. For each function f, we define a unique *FunctionDefinitionnode* ℓ_f which marks the start location of the function. The set of FunctionDefinitionnodes is $\mathbb{F}$. This enables us to analyze C programs where function calls occur. The translation of a C program containing functions without function pointers uses the translation procedure of the intra-procedural CFA and extends it in the following way:
Function call: Let p be of the form `x = func(a,b)` or `func(a,b)`. Furthermore let f_b be the start location of func and f_e be the end location of func. Then add two CFA edges $e_c = (\ell_j, \text{call p}, f_b)$ and $e_x = (f_e, \text{exit p}, \ell_{j+1})$.
return e: add a CFA edge $e = (\ell_j, \text{return: e}, f_e)$.
An example of a translation of a C-program into a CFA can be seen in Figure 2.3.

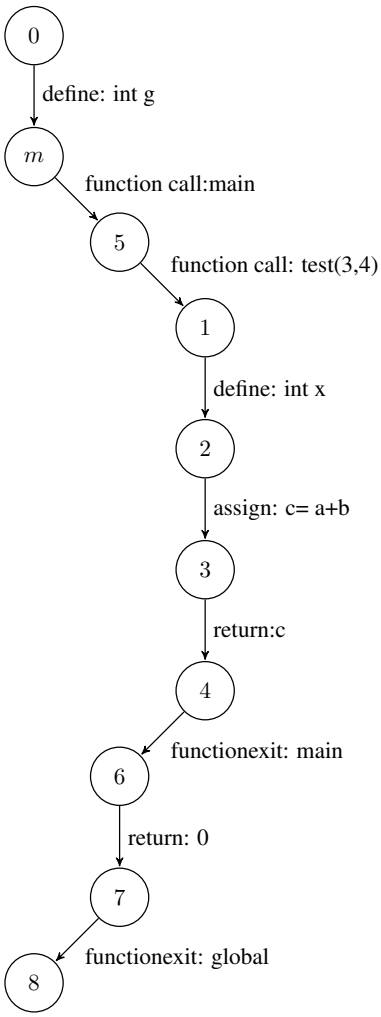



```
1:int x;
2:x=10;
3:while(true){
4: x=x+1;
5: if(x>10){
6:  goto 8;
   }
7:}
8:x=x+2;
9:
```

Figure 2.2: Intra-procedural CFA example

Function Pointers

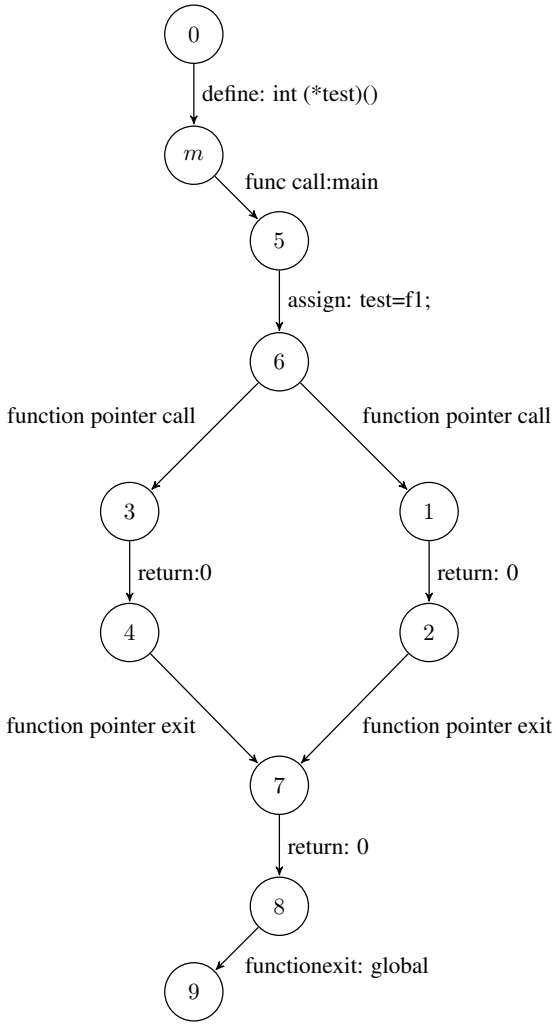
In the following we extend the definition of an inter-procedural CFA such that we can handle function calls via function pointers. We add two additional types of CFA edges: *function pointer Call* and *function pointer Exit*. CFA edges of type function pointer Call are used for function calls via a function pointer. When translating a C-Program into a CFA consider a program call via function pointer op at location ℓ. For this statement we have to introduce for all functions $f_1, ..., f_n$ with definition locations $d_1, ..., d_n$ CFA edges $e_1, ..., e_n$ of type function pointer Call where $e_i = (\ell, op, d_i)$. Furthermore, we have to add function pointer Exit edges for all functions: Let $x_1, ..., x_n$ the end program location for functions $f_1, ..., f_n$ then we add CFA edge of type function pointer Exit $k_1, ..., k_n$ where $k_i = (x_i, \text{fexitp}, \ell')$ where ℓ' is the successor location to ℓ. An example of a CFA of this type can be seen in Figure 2.4.




```
0:int g;
  int test(a,b){
1:  int c;
2:  c=a+b;
3:  return c:
4: }

m:int main(){
5:  test(3,4);
6:  return 0;
7: }
8:
```

Figure 2.3: Inter-procedural CFA example without function pointer




```
0: int (*test)();
   int f1(){
1:  return 0;
2: }

   int f2(){
3:  return 0;
4: }
m:  int main(){
5:  test=f1;
6:  (*test)();
7:  return 0;
8: }
9:
```

Figure 2.4: Inter-procedural CFA example with function pointer

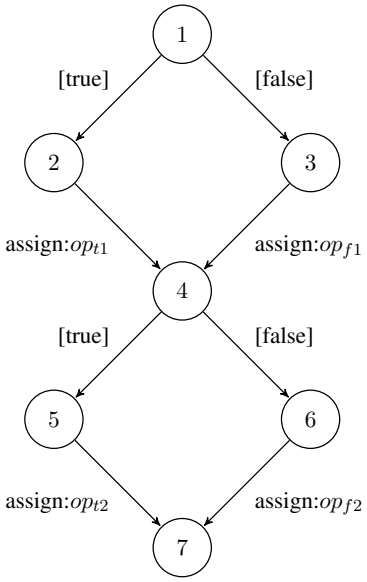


Figure 2.5: Example for CPA merge

2.2 The Configurable Program Analysis Framework

In this Section we give a summary over the CPA framework [3] [1]. A program in the CPA framework is presented as a CFA as described in Section 2.1. In the following, we refer by (L,G) to the CFA of a program under investigation.

Configurable Program Analysis with Dynamic Precision Adjustment

A *configurable program analysis* $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathit{merge}, \mathit{stop}, \mathit{prec})$ consists of an abstract domain D, a set of precisions Π, a transfer relation $\rightsquigarrow$, and functions *merge*, *stop*, *prec*.

D is the *abstract domain* $D = (C, \mathbb{E}, [\![\cdot]\!])$:
C is the set of concrete states. $\mathbb{E}$ is a semi lattice of the form $\mathbb{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ where E is the set of abstract states, $\top$ is the top element $\top \in E$, $\bot$ is the bottom element $\bot \in E$, $\sqsubseteq$ is a partial order $E \times E$, $\sqcup$ is the least upper bound: $\sqcup : E \times E \to E$. $[\![\cdot]\!]$ is a concretization function which maps abstract states to sets of concrete states: $[\![\cdot]\!] : E \to 2^C$. An abstract state e represents the set $[\![e]\!]$ of concrete states

Π is the *set of precisions*. A precision $\pi \in \Pi$ is used during successor computation. The tuple (e, π) is called *abstract state* e *with precision* π. The introduction of a precision enables the CPA algorithm to analyze each abstract state with an individual precision.

$\rightsquigarrow$ is the *transfer relation* $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ where G is the set of CFA edges of a given pro-

gram. The transfer relation encodes the semantics of the program. An element $(e, g, e', \pi) \in \rightsquigarrow$ assigns to a given abstract element e a tuple (e', π) labeled with CFA edge g. Then the *transfer relation* and *abstract domain* have to fulfill the following requirements such that the *configurable program analysis* is sound and progress of it is guaranteed:

$$[\![\top]\!] = C \text{ and } [\![\bot]\!] = \emptyset$$
$$\forall e, e' \in E : [\![e]\!] \cup [\![e']\!] \subseteq [\![e \sqcup e']\!]$$
$$\forall e \in E : \exists e' \in E, \exists \pi \in \Pi : e \rightsquigarrow (e', \pi)$$
$$\forall e \in, g \in G, \pi \in \Pi : \bigcup_{c \in [\![e]\!]} \{c' | c \xrightarrow{g} c'\} \subseteq \bigcup_{e \overset{g}{\rightsquigarrow} (e', \pi)} [\![e']\!]$$

The function *merge*: $E \times E \times \Pi \to E$ (with the restriction that $e' \sqsubseteq$ *merge*(e, e', π) in order to provide soundness) is used to weaken the second parameter e' using the first parameter e and returns a new abstract state with precision π. One purpose of *merge* can for instance be the reduction of abstract states that need to be considered by the CPA algorithm. Consider for instance the CFA in Figure 2.5. Given an abstract state e_0 at program location 1 a simple analysis would yield four states e_1, e_2, e_3, e_4 at program location 7. Each of the four states would represent a program path. For instance e_1 would represent the computation sequence 1-2-4-5-7. We now merge the two intermediate states e', e'' in program location 4 into a new abstract state $\overline{e}$ such that e' and e'' are subsumed by $\overline{e}$. Then we would only have two states $\overline{e}_1, \overline{e}_2$ to consider in program location 7. $\overline{e}_1$ represents the computation sequences 1-2-4-5-7 and 1-3-4-5-7.

The purpose of the function *stop*: $E \times 2^E \times \Pi \to \mathbb{B}$ is to determine if a given abstract state e with precision π is subsumed by a set of abstract states R. The following restriction is imposed on *stop* in order to guarantee soundness of the analysis: *stop*$(e, R, \pi) = true$ implies $[\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$.

prec: $E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ is the *precision adjustment function*. The following must hold in order for soundness of the analysis:
$\forall e, e' \in E, \pi, \pi' \in \Pi, R \subseteq E \times \Pi : (e', \pi') =$ *prec*$(e, \pi, R) \Rightarrow [\![e]\!] \subseteq [\![e']\!]$
So *prec* allows the widening of the first parameter e when calculating a new precision π'.

CPA Algorithm

The CPA algorithm given in Algorithm 2.2.1 keeps track of 2 sets reached and waitlist. Each set contains pairs of the form (e, π) where e is an abstract element and π is a precision. The algorithm computes the set of reachable abstract states for a given program (which is encoded in the transfer relation) and an initial pair (e_0, π_0) of abstract element e_0 and and precision π_0. The set of reachable abstract states is an over-approximation of a set of concrete states. The algorithm performs the following steps:
First we initialize the waitlist with a set of abstract start states W_0. As long as the waitlist is not empty take a pair (e, π) from the waitlist. For each successor of the form $(\hat{e}, \pi)$ calculate a new pair (e', π') with a new precision π' which is based on the set of pairs in reached using the function *prec*$(\hat{e}, \pi,$ reached$)$. This new pair is then merged with each pair (e'', π'') with a resulting abstract state $\overline{e}$. If $\overline{e} \neq e''$ then add the pair $(\overline{e}, \pi')$ to reached and waitlist and remove

Algorithm 2.2.1 CPA Algorithm [1]

Input: a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathit{merge}, \mathit{stop}, \mathit{prec})$, a set $R_0 \subseteq E \times \Pi$ of abstract states with precision, a subset $W_0 \subseteq R_0$ of frontier abstract states with precision, where E denotes the set of elements of the semi-lattice of D

Output: a set of reachable abstract states with precision, a subset of frontier abstract states with precision

```
1:  reached := R_0;
2:  waitlist := W_0;
3:  while waitlist ≠ ∅ do
4:    choose (e, π) from waitlist; remove (e, π) from waitlist;
5:    for each ê with e ⇝ (ê, π) do
6:      (e', π') := prec(ê, π, reached);
7:      for each (e'', π'') ∈ reached do
8:        ē := merge(e', e'', π');
9:        if ē ≠ e'' then
10:         waitlist := (waitlist ∪ {(ē, π')}) \ {(e'', π'')};
11:         reached := (reached ∪ {(ē, π')}) \ {(e'', π'')};
12:     if ¬ stop(e', {e | (e, ·) ∈ reached}, π') then
13:       waitlist := waitlist ∪ {(e', π')};
14:       reached := reached ∪ {(e', π')};
15: return (reached, ∅)
```

the pair (e'', π'') from reached and waitlist. Finally the algorithm adds (e', π') to the set waitlist and reached if e' is not subsumed by the set of abstract states of reached.

Composite CPA

It is possible to combine different CPAs to a single Composite CPA. The abstract Element of the Composite CPA are then the Cartesian products of the abstract Elements of the different CPAs. The transfer relation $\overset{C}{\rightsquigarrow}$ of the Composite CPA simple triggers for an abstract element $e = (e_1, ..., e_n)$ the transfer relations $\overset{1}{\rightsquigarrow}, ..., \overset{n}{\rightsquigarrow}$ of the corresponding CPAs. For the exact formal definition of the Composite CPA please refer to [1].

CHAPTER 3

Interpreter CPA

In this Section, we introduce the interpreter CPA $\mathbb{I} = (D, \Pi, \Delta, \mathit{merge}, \mathit{stop}, \mathit{prec})$. The Sections 3.1, 3.2,3.3 describe our interpreter CPA $\mathbb{I}$. In Section 3.1 the *abstract domain* of D is defined. In Section 3.2 we give a Definition of our interpreter transfer relation Δ. In Section 3.3, we present the definitions for the functions *merge*, *stop* and *prec*.

3.1 The Abstract Domain

The abstract elements of the abstract domain of $\mathbb{I}$ are concrete memory states. In the following we describe our memory model.

Features

Our CPA is able to handle several complex C-constructs like function pointers or pointer calculations. The complete list of features the CPA supports can be seen in Table 3.1

The memory model has certain central features namely:

1. simple pointer arithmetic (+ and -) within a data structure
2. address offset calculations within a data structure
3. prohibit pointer arithmetic outside of a defined data structure or memory block
4. save data and addresses(for data and functions) separately

These features enable the interpreter to determine whether a pointer calculation is valid or not. That is whether a pointer calculation happens within a data structure or not. Furthermore the separation of data and addresses enables the interpreter to detect suspicious access behavior like access of a function-address memory location for data or vice versa as can be seen in Figure 3.1.

Integer data types (signed/unsigned)	**Operations for integer types**

`CHAR`
`SHORT`
`INT`
`LONG`
	Arithmetic operations (+,-,*,/)
	Logical operations (`LAND`, `LOR`, `LNOT`)
	Binary operations (`BAND`, `BOR`, `BNOT`)

Pointer	**Array**
Supported data types: integer data types,structs/unions, Enums,typedefs Pointer Arithmetic: +,- Pointer specific Operations: *,& Use of variables of type `INT` and `LONG` as pointer	Supported data types: integer data types, Structs/Unions, enums, typedefs, pointer Definition of an array with fixed or variable size; Multidimensional arrays Access of arrays via pointer arithmetic
Struct/Union	**Typedef**
Supported Member-Data types: Integer types, structs/Unions, enums,typedefs, pointer Definition of struct and unions Definition of struct/union variables Forward declarations Access of fields via pointer arithmetic	Supported data types:Integer types, structs/unions, enums, typedefs,pointer
Function pointer:	**Enumerate**
Definition of function pointer Call of a function through function pointer Pointer on function pointer	Definition of enums Definition of enum variables
General program constructs	**Casts**
Assignment Function call: - Supported parameter types: `INTEGER, ADDRESSES, TYPEDEF` - Return type: Any of the upper defined recursive function calls	Cast from pointer to `INT` or `LONG` Cast from integer data type to another integer data type Cast between pointer data types

Table 3.1: Features

Definition

In this section we give a description of the *abstract domain* D of our interpreter CPA. As described in Section 2.2 the domain D is of the form $D = (C, \mathbb{E}, [\![\cdot]\!])$. C is the set of concrete memory states. $\mathbb{E}$ is a flat semilattice. A Hasse diagram describing $\mathbb{E}$ can be seen in Figure 3.2. The concretization function $[\![\cdot]\!]$ is defined in the following way:

$$[\![e_i]\!] = \{c_i\}, c_i \in C$$
$$[\![\mathit{Error}]\!] = \{\}$$
$$[\![\top]\!] = C \text{ and } [\![\bot]\!] = \{\}.$$

```
int test(){
  return 0;
}
int (*p)();
int main(){
  int c = 3;
  p = test:
  c = ((int)p)+3; //access function pointer for integer
  return 0;
}
```

Figure 3.1: Example for suspicious access behavior for function pointer

So each abstract state of the form e_i represents exactly one concrete memory state. The interpreter transfer relation which is defined in Section 3.2 uses the abstract state *Error* to indicate faulty or undefined program behavior. $\top$ and $\bot$ are defined as described in Section 2.2.

An abstract state $\overline{s}$, $\overline{s} \in E$ has the form $\overline{s} := (Stacks, M)$. $Stacks$ is a list of partial functions $Scope_i$ with $Scope_i : Var \hookrightarrow \mathbb{A}$ where $\mathbb{A}$ is the set of addresses and *Var* the set of currently defined variables. An address is of the form $(MID, Offset)$, where $Offset \in \mathbb{N}$. As can be seen in Figure 3.6 the memory identifier (*MID*) is used to identify the memory block while the offset determines the memory cell of the given memory block. For an element $a \in \mathbb{A}$ we denote with a_m the *MID* and a_o the memory offset *Offset*.

M is used to describe the memory and has the form $M = (M_S, M_D)$. The function M_S: $MIDs \hookrightarrow N$ is used to determine the size of a given memory block. The function M_D: $MIDs \rightarrow (\mathbb{N} \hookrightarrow Data)$ is used to present the current memory state. Thus we have for every memory block (which has a unique *MID*) a partial function which represents the contents of the memory block. If a memory block is created/deleted a *MID* is added/removed to *MIDs*. The set $Data$ is of the form $Data = EMPTY \cup \{0...255\} \cup \mathbb{A} \cup \mathbb{F}$ where $\mathbb{F}$ is the set of all FunctionDefinitionnodes (c.f Section 2.1(Interprocedural CFA)).

Implementation

The memory model is used by the interpreter to represent the current(concrete) memory state of the program execution. The memory model can be divided into two parts. The *low-level memory model* consisting of memory blocks and memory cells which represent the memory while the *high-level memory model* consists of a list of scopes called stack where each scope contains a set of variables.

The main element of the low-level memory model is a memory block. A memory block consists of an array of memory cells where each index(named memory location) can hold a different type

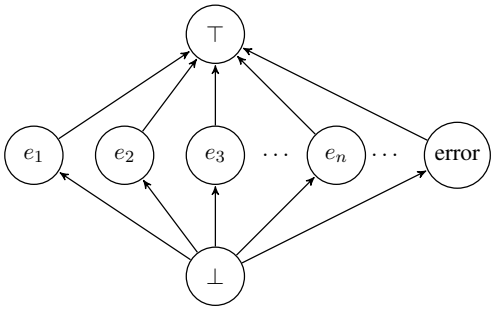


Figure 3.2: Hasse diagram of the flat lattice for the *abstract domain*

of memory cell. There are three types of memory cells : data-, address- and function memory cells.

A *data memory cell* (DMC) holds a byte. So for example to store a 32-bit integer value a memory block of size 4 or more is required. An *address memory cells* (AMC) can store one address. An address consists of a reference to a memory block and an offset. Thus an address is pointing to a memory cell within an memory block as can be seen in Figure 3.6. A *function memory cell* (FMC) can hold a reference to a function definition node of the CFA of the C program. The memory locations of the array are all empty when a memory block is initialized the first time representing uninitialized memory. It should be noted that when a memory location already holds a certain type of memory cell the memory cell can not be overwritten by another type of memory cell.

Variables are an integral part of the high-level memory model. There are 6 types of variables: primitive variables, pointer variables , array variables, composite variables (Struct and Union), function-pointer variables and Enumeration variables.

Each kind of variable has a reference to an address. Furthermore each variable has a name and additional type information specific to the variable. Each variable belongs to exactly one scope.

The interpreter holds a linked list of scopes called stack. A scope contains a variable environment and contains a reference to its parent scope. The local variables of a function are represented by a scope. The stack contains a reference to the current scope s_c. The initial stack holds a reference to the global scope which stores the global variables. When the interpreter enters a function a new scope s_n is created and s_c is set as parent scope of s_n. Stack then sets s_n as the new current scope. When the interpreter leaves the function the parent scope of s_n, s_c is restored as new current scope.
Addresses are the interface between the high- and low-level memory model. Every variable has an address which together with the variable type can be used to read the variable value out of

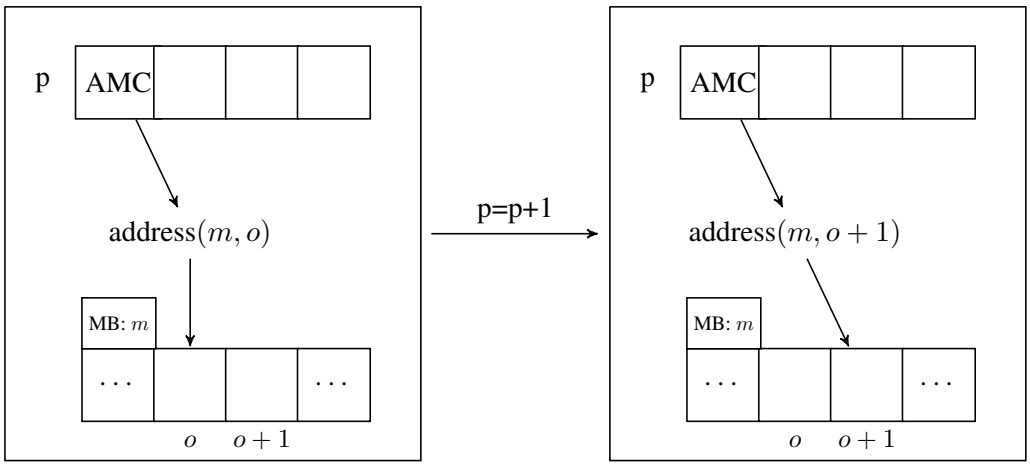


Figure 3.3: Example of pointer arithmetic

memory block. Addresses play an important role in pointer arithmetic. When pointer arithmetic is performed an address is taken and a new address is calculated by changing the offset of the given address. Figure 3.3 displays an example for pointer arithmetic. A pointer p is incremented by one. The memory block of p holds an AMC which holds an address that points to another memory block m with offset o. By incrementing p, a new address is created which points to memory block m with offset $o + 1$. If $o + 1$ exceeds the size of m an error occurs. Otherwise the new address is saved in the AMC of p.

The naive Implementation of the memory model described above can lead to a memory overflow for a complex program easily: When the interpreter enters a new program state the old instance of the memory model has to be copied and then modified in order to reflect the current memory state. This approach is quite inefficient. In order to avoid this problem small modifications to the memory model have to be made. These modifications are based on the formal data structure called *fat node* [7]. The basic idea is to only have one representation of the memory where views represent the current memory state depending on the program state. This means that only information that needs to be changed that is modified in the memory model which leads to a smaller memory consumption. A fat node has the form of a tree where nodes represent computation steps in a program and branches represent different computation paths in a program. In the following we call a fat node a *version tree*. A *version history* is a path in a version tree from a node to the root.

The general idea is to store for each field of an object, whose value could change, a version tree instead of its scalar value(reference or data). The shape of the version tree is the same for each field. Each node of the tree either stores the value of the field associated with the tree or contains no value. Furthermore every node points to exactly one predecessor node. Information of a field can be accessed via views. Every view v has a start node n. For a given object o with field z the value of z for v the $value(z, v)$ is determined as described in Algorithm 3.1.1. The

Element	Field
stack	`list` scopes
scope	`list` variables
memory blocks	`array` memory cells
AMC	`address` addr
DMC	`byte` data
FMC	`FunctionDefinitionNode` function

Table 3.2: Fields of memory elements that have hash maps

purpose of this Algorithm is to retrieve the most recent value for field z of view v. Thus it begins to look up the content of the start node $node(z, v)$. If the node contains a value we return it otherwise we iterate over the version history of the start node until a value can be obtained. If no node contains a value then we return error. An example for a version tree for the two fields x and y can be seen in Figure 3.4. We have three views for variables x and y. For instance v_3 has start node 7 and thus we have the version history for v_3: $7 \rightarrow 6 \rightarrow 5 \rightarrow 2 \rightarrow 1$. So we start by looking up the value of node 7 of variable x. Node 7 is empty, we continue with node 6 which has the value 3. For variable y node 7 is empty so we continue with node 6 which is also empty. Node 5 holds the value 8. Thus we have that x=3 and y=8. For view v_1 we have that x=4 and y=-1.

Whenever we add a new node to the version tree we have to update for each field its tree which is inefficient. Thus we modify our approach in the following way: Instead of saving a version tree for each field we save a map and maintain only one version tree. The nodes of the version tree do not contain values anymore but serve instead as keys for the maps. The value of a field f with view v can be retrieved as described in Algorithm 3.1.2. The purpose of Algorithm 3.1.2 is the same as Algorithm 3.1.1. The difference between Algorithm 3.1.2 and Algorithm 3.1.1 is that instead of looking up the contents of nodes the values of a hash map are retrieved where the nodes are the keys. An example of a version tree and fields with a map can be seen in Figure 3.5. For a view v_2 we have the following version history: $4 \rightarrow 2 \rightarrow 1$. We then use the map of field f to retrieve its value. First we look up the value of key 4 which returns empty. Then we follow the version history to node 2 and lookup its value in the map which again is empty. Finally we lookup the value of the key 1 which returns 4. We perform the same procedure with field g. We lookup the value for key 4 which returns 88. Thus we have that $f = 4$ and $g = 88$ in view v_2.

We then adapt our memory model with the fat node approach by modifying memory element fields that change over the course of the computation. Table 3.2 shows for the elements of the memory model which fields have hash maps. Element fields like for instance the predecessor scope that do not change during computation do not need to be saved hash maps.

Algorithm 3.1.1 $value(z, v)$

Input: a field z and a view v
Output: a value(either data or reference depending on type of the field) or error

```
1: state := node(z, v);
2: while state ≠ null do
3:     if state.value ≠ null then
4:         return state.value
5:     state := state.prev;
6: return error
```

Algorithm 3.1.2 $value_n(f, v)$

Input: a field f and a view v
Output: a value(either data or reference depending on type of the field) or error

```
1: state := node(v);
2: while state ≠ null do
3:     if f.get(state) ≠ null then
4:         return f.get(state)
5:     state := state.prev;
6: return error
```

3.2 The Transfer Relation

We gave a general overview over the transfer relation in Section 2.2 (CPA). In this section we will define the interpreter transfer relation as a function. The purpose of the interpreter transfer relation is to model the C semantics. Therefore we define the transfer function $\Delta : E \times G \to E$ where for a given abstract State e and program edge g there exits only one resulting state e'.

Helper Functions and Grammars

Before defining the transfer relation we introduce a set of helper functions and grammars: The function $addr_{\bar{s}}$: $Var \to \mathbb{A}$ returns the address of a variable in abstract state $\bar{s}$ where $addr_{\bar{s}}$ is defined as: $addr_{\bar{s}} = \cup_{i=0}^{n} Scope_i$ where $Scope_i$ is part of $\bar{s}$. The union of the functions $Scope_i$ is indeed a function $addr_{\bar{s}}$ because each variable belongs to exactly one stack function and is thus undefined in the other functions. Furthermore we have that $addr_{\bar{s}}$ is a total function because $\cup_{i=0}^{m} domain(Scope_i) = Var$.

The function *type*: $Var \to \mathbb{T}$ assigns to every variable a type. Note that the set of abstract states E is no parameter of *type* because the type of a variable is already determined before runtime and does not change during runtime. $\mathbb{T}$ is the set of types for which we assume that they have been gathered by a first parse of the program. The function *bt*: $PVar \to \mathbb{T}$ is used to determine for a variable of type pointer its base type where *PVar* is the set of variables which are pointers with $PVar \subseteq Var$. For instance, for the declaration `int ***a;` the basetype $bt(a)$ yields int.

The function *tc*: $Var \to classes$ with

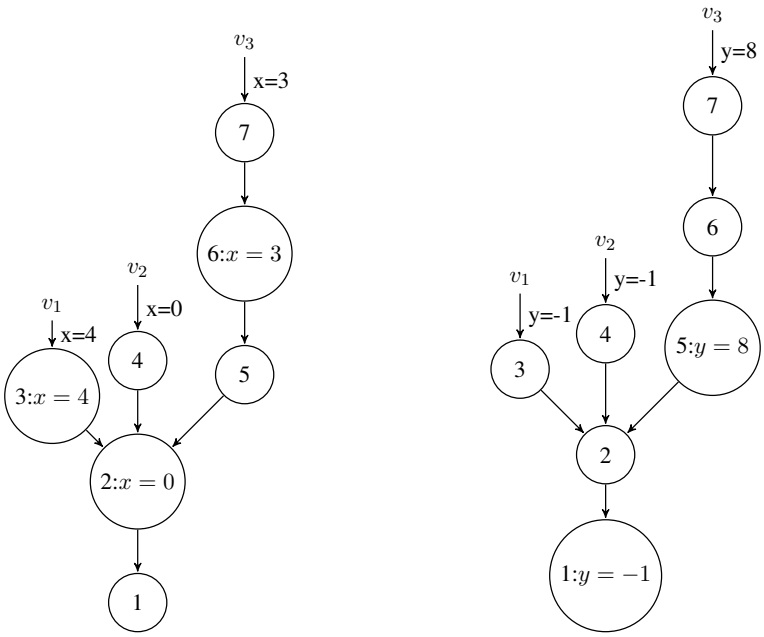


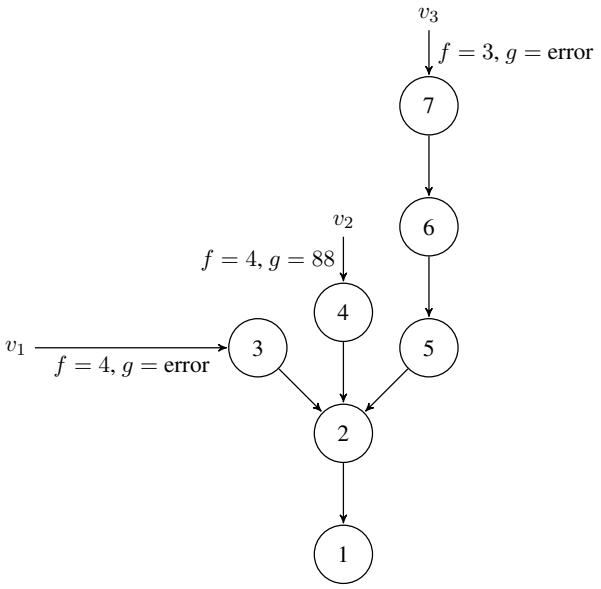
Figure 3.4: Example for fat nodes for fields x and y

classes $= \{$*Prim*, *PrimPnt*, *Pnt*, *Struct*, *Union*, *Enum*, *FPnt*, *array*$\}$ assign each variable to a certain class of type. Variables which belong to the class *Prim* are variables whose type is primitive (SHORT,INT,LONG) and the variables contain primitive or empty data. Primitive variables which are used to for pointer offset calculations (thus containing an address or empty data) are belonging to the class *PrimPnt*. A variable that belongs to the class *PrimPnt* can not belong to the class *Prim* in subsequent interpreter elements. The class *Pnt* stands for pointer while the class *FPnt* stands for function pointers.

In the following we denote with A_m the memory block and A_o the offset of an address A. Then *dV*: *Var* x $E \rightarrow \mathbb{N} \cup \{$*Error*$\}$ is a function which is used to retrieve the numeric value for an interpreter element and a variable which belongs to typeclass *Prim*. More specifically *dV* is defined in the following way:

$$dV(v, \overline{s}) = \begin{cases} nr & nr = \sum_{n=0}^{N} nr_i * 2^{8*i}, nr_i = \mathit{M}_D(\mathit{addr}_{\overline{s}}(v)_m)(\mathit{addr}_{\overline{s}}(v)_o + i), \mathit{tc}(v) = \mathit{Prim} \\ \mathit{Error} & \text{otherwise} \end{cases}$$

with $N = \mathit{size}(\mathit{type}(v)) - 1$. *dV* is only defined for variables belonging to the type class *Prim* because addresses can not be decoded into numerical form (in this model). The function *dAddr* is used to extract the numeric value for an interpreter element, a basetype and an address and is defined in a similar way.



Map for field f	
Key	Value
Node 1	4
Node 7	3
Map for field g	
Key	Value
Node 4	88

Figure 3.5: Example for fat nodes for fields f and g

rexpr	::=	*id* | $\mathbb{N}$ | *urexpr* | *brexpr*
urexpr	::=	*SOP rexpr*
brexpr	::=	$rexpr_1$ *BOP* $rexpr_2$
SOP	::=	+ | - | * | & | ! | inv | (*cast*)
BOP	::=	+ | - | / | * | & | '|' | && | '||'
lexpr	::=	*id* | $*$*lexpr* | (*cast*)*lexpr*

Table 3.3: Grammars for right hand side (*rexpr*) and left hand side (*lexpr*) expressions

Assignments are of the form *lexpr* $=$ *rexpr*. The grammars for *lexpr* and *rexpr* are given in Table 3.3. *id* is a string and *cast* are all cast expressions which occur in a program.

For the evaluation of right hand expressions, we define the function $eval_r$: *rexpr* $\rightarrow \{Error\} \cup \mathbb{Z} \cup Var \cup \mathbb{F} \cup \mathbb{P}$ where $\mathbb{P}$ is a set of tuples $\mathbb{P} = \{(Address, Type, Level) | Address \in \mathbb{A}, Type \in \mathbb{T}, Level \in \mathbb{N}\}$. Tuples of $\mathbb{P}$ serve as intermediate results in the definition of the evaluation functions. It must be noted that depending on the evaluation function ($eval_r$ or $eval_l$) the semantic of the tuples is different. $eval_r$ uses *Address* as data while $eval_l$ uses *Address* as memory location in which to write data. A level greater 0 indicates that the memory at the *Address* location holds an address while otherwise it holds data. Depending on the *Level*, *Type* can be either seen as data type($Level = 0$) or base type for a pointer($Level \geq 0$).
To simplify the notation we introduce for an element $p \in \mathbb{P}$ the following abbreviations: p^a for its address, p^t for its basetype, and p^l for its level.

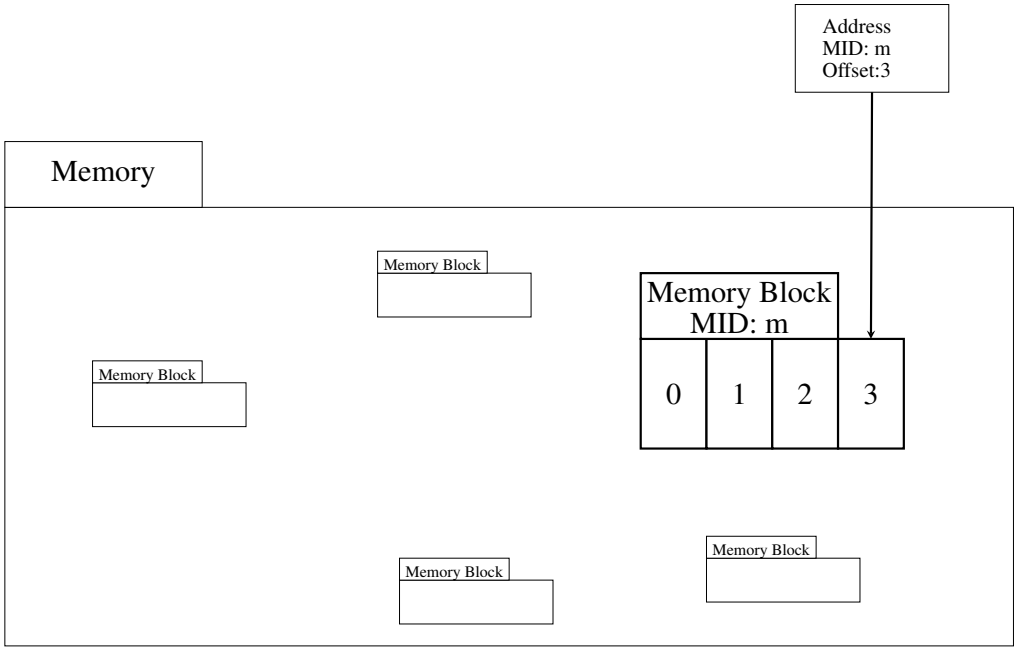


Figure 3.6: Conceptual view of the memory model

$eval_r$ is defined recursively. We define the following notations:
Let $e = eval_r(rexpr, \bar{s})$, $Scope_n$ be the topmost stack and $Scope_0$ the global stack of the abstract Element $\bar{s}$. Let $e_1 = eval_r(r_1, \bar{s})$ and $e_2 = eval_r(r_2, \bar{s})$ and for an address A and number n we introduce the following abbreviation: $A \mathbin{\mathtt{op}} n = (A_m, A_o \mathbin{\mathtt{op}} n)$ where A_m is the *MID* and A_o is the offset of the address A and $\mathtt{op} \in \{+, -\}$. The address with empty reference (being a null pointer) is called $A_\perp$. A tuple of the form $(A, T, L)^*$ (where A is an address, T is as type, and L is a number) denotes that the implementation uses a temporary variable to represent the tuple. Definitions of $eval_r$ for unary and base-case *rexpr* are given in Table 3.4 while definitions of $eval_r$ for binary and cast *rexpr* are given in Table 3.6.

For the left hand side expressions *lexpr* there exists an evaluation function $eval_l$: *lexpr* $\rightarrow \{Error\} \cup Var \cup \mathbb{P}$ where $eval_l$ is defined recursively with $x = eval_l(lexpr, \bar{s})$. The Definitions can be found in Table 3.5

The Transfer Function

We give a definition of the transfer function for the following types of CFA edges which can be found in Section 2.1: *definition*, *assignment*, *skip* , *assume*,*function call*, *function exit*, *function return*,*function pointer call* and *function pointer exit*.

rexpr	$eval_r(rexpr, \overline{s})$	
nr	nr	$nr \in \mathbb{Z}$
id	v	$v \in var, name(v) = id,$ where $v \in Scope_n(Var) \cup Scope_0(Var)$
	f	$f \in \mathbb{F}, name(f) = id$
	Error	otherwise
$-$*rexpr*	$-e$	$e \in \mathbb{Z}$
	Error	otherwise
&*rexpr*	$(addr_{\overline{s}}(e), typ(e), 1)$	$e \in Var, tc(e) \in \{Prim, Struct, Enum\}$
	$(addr_{\overline{s}}(e), typ(e), n+1)$	$e \in Var, tc(e) = Pnt, l(e) = n$
	e	$e \in \mathbb{F}$
	Error	otherwise
!*rexpr*	1	$e \in \mathbb{Z}, e = 0$
	0	$e \in \mathbb{Z}, e \neq 0$
	1	$tc(e) = Prim, dV(e) \neq 0$
	0	$tc(e) = Prim, dV(e) = 0$
	Error	otherwise
$*$*rexpr*	$(M_D(M_D(addr_{\overline{s}}(e))), bt(e), l(e)-1)$	$M_D(M_D(addr_{\overline{s}}(e))) \in \mathbb{A}, tc(e) = Pnt, l(e) > 1$
	$M_D(M_D(addr_{\overline{s}}(e)))$	$tc(e) = Pnt, M_D(M_D(addr_{\overline{s}}(e))) \in \mathbb{F}$
	$dAddr(M_D(addr_{\overline{s}}(e)), bt(e), \overline{s})$	$tc(e) = Pnt, l(e) = 1, tc(bt(e)) = Prim \vee$ $\vee tc(bt(e)) = Enum$
	$M_D(M_D(addr_{\overline{s}}(e)))$	$tc(e) = FPnt, l(e) = 1$
	$(M_D(M_D(addr_{\overline{s}}(e))), bt(e), l(e)-1)$	$tc(e) = FPnt, l(e) > 1$
	Error	otherwise
: *rexpr*	$inv(e)$	$e \in \mathbb{N}$
	$inv(dV(e))$	$e \in Var, tc(e) = Prim$
	Error	otherwise

e = $eval_r(rexpr, \overline{s})$

Table 3.4: Definition of evaluation function $eval_r$ for base-case and unary *rexpr*.

Assignment

Let $\overline{s} = (Stacks, M)$ and assignment be of the form *lexpr* = *rexpr*. Then

$$\Delta(\overline{s}, lexpr = rexpr) \quad = \quad \overline{s}'$$

where $\overline{s}'$ is an abstract state of the form $\overline{s}' = (Stacks, M')$. $\overline{s}' = Error$ if $\overline{r} = eval_r(rexpr, \overline{s})$ or $\overline{l} = eval_l(lexpr, \overline{s})$ evaluates to *Error* . $M' = (M_S, M_{D'})$ is derived from M and reflects the changes in memory by assigning a value represented by *rexpr* to the memory location represented by *lexpr*. Before giving a formal definition of $M_{D'}$ we introduce three helper functions: *encnum*, *getaddr* and *data*. The function $encnum(n, t)$: $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is used to determine the numeric value from the t-th byte from number n and has the formal definition:

$$encnum(n, p) \quad = \quad \frac{n}{2^{8*p}} \bmod 2^8$$

$lexpr$	$eval_l(lexpr, \bar{s})$	
id	v	if $v \in$ *Var*
	Error	otherwise
**lexpr*	$(M_D(addr_{\bar{s}}(x)), bt(x), l(x) - 1)$	if $x \in$ *Var*, $tc(x) =$ *Pnt*, $l(x) > 0$
	$(M_D(x^a), x^t, x^l)$	if $x \in \mathbb{P}, x^l > 0$
	Error	otherwise
$eval_l((c)lexpr, \bar{s})$	$(addr_{\bar{s}}(x), type(c), l(c))$	$tc(c) =$ *Pnt*, $x \in$ *Var*
	Error	otherwise

Table 3.5: Evaluation of *lexpr*.

The function *getaddr*: *Var* $\times \mathbb{P} \to \mathbb{A}$ is used to retrieve the address of a variable or a tuple of the form $(a, t, l) \in \mathbb{P}$ and is defined in the following way:

$$getaddr(z, \bar{s}) \quad = \quad \begin{cases} addr_{\bar{s}}(z) & z \in Var \\ z^a & z \in \mathbb{P} \end{cases}$$

The function *data*: $(image(eval_r) - \{Error\}) \times E \times \mathbb{N} \to Data$ is used to determine the new value for the cells of the memory block determined by *lexpr* with:

$$data(r, s, p) \quad = \quad \begin{cases} encnum(r, p) & r \in \mathbb{N} \\ M_D(getaddr(r, s)_m)(getaddr(r, s)_o + p) & r \in Var \vee r \in \mathbb{P} \\ r & r \in \mathbb{F} \end{cases}$$

Furthermore we introduce the following abbreviations: $(a_m, a_o) =$ *getaddr*$(\bar{l}, \bar{s})$ and $(d_m, d_o) =$ *getaddr*$(\bar{r}, \bar{s})$. We can then give the following formal definition of $M_{D'}$:

$$M_{D'}(a_m)(a_o + p) \quad = \quad \begin{cases} data(\bar{r}, \bar{s}, p) & 0 \leq p \leq size(type(eval_l(lexpr))) \\ M_D(a_m, a_o + p) & \text{otherwise} \end{cases}$$

Definition

Let the abstract element be of the form $\bar{s} = (S, M)$. *Definition* is of the form 'type id' where type $\in \mathbb{T}$. Furthermore let S_n be the last scope of the stack S. Then we define the transfer function Δ for *definition*:

$$\Delta(\bar{s}, \text{type } id) = \begin{cases} (S', M') & eval_r(id, \bar{s}) \notin Var \text{ or } S_n(eval_r(id, \bar{s})) = undef \\ Error & \text{otherwise} \end{cases}$$

which means that there is a new abstract element (S', M') if there is not defined a variable in scope S_n. Then the abstract element (S', M') represent the allocation of a new variable which belongs to the current scope with a new memory block. We do this by updating *Var* by adding a

new variable for id. Then S' is retrieved from S by removing the current scope S_n from the list S and adding an updated scope S_n^*: $S' = S_n^*$:tail(S_n)

$$S_n^*(v) = \begin{cases} S_n(v) & \textit{name}(v) \neq id \\ (M_{\text{ID}}, 0) & \textit{name}(v) = id \end{cases}$$

where M_{ID} is a new memory id representing the new memory block for the new variable in *Var*. The tuple $M' = (M_S^*, M_D^*)$ represents the new state of the memory and is defined using M_S and M_D:

$$M_S^*(m) = \begin{cases} size(\text{type}) & m = M_{\text{ID}} \\ M_S(m) & m \neq M_{\text{ID}} \end{cases} \qquad M_D^*(m) = \begin{cases} f & m = M_{\text{ID}} \\ M_D(m) & m \neq M_{\text{ID}} \end{cases}$$

The new partial function $f : \mathbb{N} \hookrightarrow Data$ represents the content of the new empty memory block M_{ID}: $f(n) = \textit{EMPTY}, 0 \leq n < M_S^*(M_{\text{ID}})$

Assume

Let $\bar{s} = (S, M)$ and $e = \textit{eval}_r(\textit{rexpr}, \bar{s})$. *Assume* is either of the type [*rexpr*] or [$\neg$*rexpr*]. We define assume depending on the type of statement. We use the helper function $\textit{dexpr}(p)$: $\textit{image}(\textit{eval}_r) \rightarrow \mathbb{N} \cup \{\textit{Error}\}$ with:

$$\textit{dexpr}(e, \bar{s}) = \begin{cases} e & e \in \mathbb{N} \\ \textit{dV}(e) & e \in \textit{Var}, \textit{tc}(e) = \textit{Prim} \\ \textit{dAddr}(e) & e \in \mathbb{P}, \textit{tc}(e^t) = \textit{Prim}, e^l = 0 \\ 1 & e \in \textit{Var}, \textit{tc}(e) = \textit{Pnt}, M_D(\textit{addr}_{\bar{s}}(e)) \neq A_\perp \\ 0 & e \in \textit{Var}, \textit{tc}(e) = \textit{Pnt}, M_D(\textit{addr}_{\bar{s}}(e)) = A_\perp \\ \textit{Error} & \text{otherwise} \end{cases}$$

Then we have for [*rexpr*] and [$\neg$*rexpr*]:

$$\Delta(\bar{s}, [\textit{rexpr}]) = \begin{cases} \bar{s} & \textit{dexpr}(e, \bar{s}) \neq 0 \\ \perp & \textit{dexpr}(e, \bar{s}) = 0 \\ \textit{Error} & \text{otherwise} \end{cases}$$

$$\Delta(\bar{s}, [\neg\textit{rexpr}]) = \begin{cases} \bar{s} & \textit{dexpr}(e, \bar{s}) = 0 \\ \perp & \textit{dexpr}(e, \bar{s}) \neq 0 \\ \textit{Error} & \text{otherwise} \end{cases}$$

Function Call and Function Return

Before defining the semantics of the transfer function for function call we introduce the abbreviation $\Delta^{m}_{i=0}(\overline{s}, \mathit{command}_i)$ which is defined recursively:

$$\begin{aligned} \Delta^{m}_{i=0}(\overline{s}, \mathit{command}_i) &= \Delta(\Delta^{m-1}_{i=0}(\overline{s}, \mathit{command}_i), \mathit{command}_m) \\ \Delta^{0}_{i=0}(\overline{s}, \mathit{command}_i) &= \Delta(\overline{s}, \mathit{command}_0) \end{aligned}$$

So $\Delta^{m}_{i=0}(\overline{s}, c_i)$ models for a set of CFA-edges $c_0, \cdots, c_i$ the continuous application of the interpreter transfer function to an abstract element $\overline{s}$: $\overline{s} \overset{c_0}{\Rightarrow} \overline{s}_1 \cdots \overset{c_n}{\Rightarrow} \overline{s}_n$ where $\overline{s}_i \overset{c_i}{\Rightarrow} \overline{s}_{i+1}$ stands for $\Delta(\overline{s}_i, c_i) = \overline{s}_{i+1}$.

Let function call be of the form $id = \mathit{fname}(\mathit{arglist})$ where *arglist* is a list of right expressions representing the values of the parameter. Furthermore we have a list *deflist* which holds the definition statements of the parameters of the function. $\overline{s}$ is of the form $\overline{s} = (S, M)$. Then

$$\Delta(\overline{s}, id = \mathit{fname}(\mathit{arglist})) = \overline{s}^*$$

where $\overline{s}^* = (S^*, M^*)$ is constructed as follows: First, we add a new *scope* for our function environment: $S' = S_{n+1}$:S. Then we apply the transfer function to (S', M) for the definition statements in *deflist*:

$$\Delta^{m}_{i=0}((S', M), \mathit{deflist}_i) = (S'', M')$$

which lead to a new set of variables $v_0, ..., v_n$. We then have to assign to these new variables the values in *arglist*. We do this by applying a modified transfer function Γ to (S'', M') with the a list of commands: $id(\mathit{deflist}_0) = \mathit{arglist}_0, ..., id(\mathit{deflist}_n) = \mathit{arglist}_n$

$$\Gamma^{m}_{i=0}((S'', M'), id(\mathit{deflist}_i) = \mathit{arglist}_i) = (S^*, M^*)$$

where the function Γ is the same as Δ with the difference that $\mathit{eval}_r(\mathit{arglist}_i, (S'', M'))$ is replaced with $\mathit{eval}_r(\mathit{arglist}_i, (S, M))$ for assignments. Γ uses $\mathit{eval}_r(\mathit{arglist}_i, (S, M))$ to access the values for $\mathit{arglist}_i$ which either belong to the global scope or $\mathit{top}(S)$.

For function return we introduce the function *retvar*: $S \hookrightarrow$ *ID* which returns for a given *stack* a left hand side expression. The value of the return expression is then assigned to the address represented by this left hand side expression. If there is no left hand side expression for the given *stack* (e.g. a function call of the form `func();` in contrast to `x=func()`) then no operation is performed.

Thus we have for a given return statement of the form 'return *rexpr*' the following definitions: Let $\overline{s}$ be of the form (S, M) where S_n is the topmost *Stack* of *S*. Then $\Delta(\overline{s}, \text{return } \mathit{rexpr}) = \overline{s}'$ where $\overline{s}' = \alpha(\overline{s}, id = \mathit{rexpr})$ if $id = \mathit{retvar}(S_n)$ is not undefined. The function α is the same as Δ with the difference that $\mathit{eval}_l(id, \overline{s})$ for an assignment is being replaced with $\mathit{eval}_l(id, \overline{s}^*)$ with $S^* = S - \{S_n\}$. If $id = \mathit{retvar}(S_n)$ is undefined then $\Delta(\overline{s}, \text{return } \mathit{rexpr}) = \overline{s}$.

Function Exit

For a CFA Edge of the form function exit we have the following definition: Left $\bar{s}$ be of the form (S, M). Then $\Delta(\bar{s}, \mathit{fexit}) = \bar{s}'$ where $S' = \mathit{tail}(S)$. A function can have several *fexit* Edges (where each *fexit* Edge represents a different function call). Therefore we must make sure that the correct *fexit* Edge is selected. We do this by adding the interpreter CPA and a callstack CPA to a composite CPA as described in Section 2.2.

Function Call via Function Pointer

We have a CFA-Edge of the form $(id = (*\mathit{rexpr})(\mathit{arglist}), \mathit{fnode})$. $id = (*\mathit{rexpr})(\mathit{arglist})$ is a C statement denoting a function call via a function pointer. *fnode* is a function definition node. Furthermore let $\bar{s}$ be an abstract state. Then there is an abstract successor state in $E - \{\mathit{Error}, \bot, \top\}$ if the value of the function pointer coincides with the function definition node of specified by the CFA-Edge. Otherwise the abstract successor state is $\bot$. So the transfer function for a function call via a function pointer is described in the following way:

$expr$	$\Delta(\bar{s}, expr)$	
$(id = (*\mathit{rexpr})(\mathit{arglist}), \mathit{fnode}))$	$\Delta(\bar{s}, id = \mathit{name}(\mathit{fnode})(\mathit{arglist}))$	$\mathit{eval}_r(\mathit{rexpr}, \bar{s}) \in \mathbb{F}$, $\mathit{name}(\mathit{fnode}) = \mathit{name}(\mathit{eval}_r(\mathit{rexpr}, \bar{s}))$
	$\bot$	$\mathit{eval}_r(\mathit{rexpr}, \bar{s}) \in \mathbb{F}$, $\mathit{name}(\mathit{fnode}) \neq \mathit{name}(\mathit{eval}_r(\mathit{rexpr}, \bar{s}))$
	Error	otherwise

Function Exit via Function Pointer

For this type of CFA edge we introduce a new helper function *retnod*: *Stacks* $\rightarrow L$. *retnod* contains for each Stack s the return location. Let the CFA edge be of the form $(\mathit{fexit}, \mathit{exitnode})$ and $\bar{s}$ be an abstract state of the form (Stacks, M) where S_n is the current Stack. Then there is an abstract successor state in $E - \{\mathit{Error}, \bot, \top\}$ if $\mathit{retnod}(S_n) = \mathit{exitnode}$. Otherwise the abstract successor state is $\bot$. We have then the following definition of the transfer function for this type of CFA edge:

$$\Delta(\bar{s}, (\mathit{fexit}, \mathit{exitnode})) = \begin{cases} \Delta(\bar{s}, \mathit{fexit}) & \mathit{exitnode} = \mathit{retnod}(S_n) \\ \bot & \mathit{exitnode} \neq \mathit{retnod}(S_n) \\ \mathit{Error} & \text{otherwise} \end{cases}$$

3.3 Definition of *merge*, *stop*, *prec*

The definition of *merge* is motivated by the fact that the interpreter analysis never combines two abstract states and the constraint that for $e_2 \sqsubseteq \mathit{merge}(e_1, e_2, \pi)$: $\mathit{merge}(e_1, e_2, \pi) = e_2$ where $e_1; e_2 \in E, \pi \in \Pi$

An abstract state is never subsumed by a set of other abstract states. In a simple `while(true){}` loop the CPA algorithm would never terminate. $\mathit{stop}(e_1, R, \pi) = false; e_1 \in E, R \subseteq \{(e, p') | e \in E, p' \in \Pi\}, \pi \in \Pi$

We have only one precision in Π we thus give the following definition which respects the CPA algorithm and the interpreter semantics: $\mathit{prec}(e, \pi, R) = (e, \pi); e \in E, \pi \in \Pi, R \subseteq \{(e, p') | e \in E, p' \in \Pi\}$

3.4 Implementation Details

Next we discuss some of the implementation details of the interpreter CPA. Additionally we give an overview over different approaches which are similar to our interpreter CPA approach.

Table 3.7 shows the sizes of the primitive data types as used by the interpreter CPA. The data type `void` can only be used as a base type for a pointer. Currently the interpreter CPA supports no operations containing floating point variables like float or double. The type for complex data structures which we call in the following *dynamic type* must be gathered at runtime contrary to our previous simplifications. This dynamic type is divided into three classes of types by the interpreter CPA: enum-, typedef-, and compositetypes (struct and union). Each dynamic type declaration contains a name which identifies the dynamic type. We have for each of the three dynamic type classes an object (called *library*) which holds a hash map which saves the type information for the corresponding type class. Each entry of the hash maps contains as Key the name of the dynamic type and as value the dynamic type information. The information in these libraries is used by the interpreter to create complex variables with the type information specified in the libraries.

The first variable defined is __BLAST_NONDET which can be used to initialize variables to predefined values at runtime without changing the code. When creating the interpreter CPA an array of values of type integer can be parsed to the initial interpreter element. The constructor of interpreter element then creates a new object of type NonDetProvider which holds the array a and an index i. Whenever the variable __BLAST_NONDET is assigned to a variable v the value $a[i]$ is assigned v and i is incremented by one. The index variable is not of type integer but a fat-node type (i.e. of type Hashmap<InterpreterElement,Type>). __BLAST_NONDET can only be used in assignments and must not be used in other program constructs like if or while because this can lead to undefined program behaviour. Consider the code in Figure 3.7 and assume the next two values which are returned by __BLAST_NONDET are 4 and 3. First we have for

$$\Delta(\overline{s}, [\text{\_\_BLAST\_NONDET} == 3]) = \bot$$

because __BLAST_NONDET returns first 4 and $4 == 3$ is false. Then we have for

$$\Delta(\overline{s}, [\neg(\text{\_\_BLAST\_NONDET} == 3)]) = \bot$$

because __BLAST_NONDET returns 3 and $\neg(3 == 3)$ is false which means that the interpreter CPA will stop the execution of this branch which is undefined behaviour.

At the beginning of the execution the interpreter CPA creates a scope named *global* where the variable __BLAST_NONDET is defined. After this a scope named *main* is created. After this the dynamic types and global definitions are parsed by the interpreter CPA. This then means that global definitions do not belong to the scope global but to the scope main. The interpreter CPA can furthermore not deal with global definitions which contain an assignment like for instance `int x=3;`. CIL can not simplify these statements to definition and assignment statements because the are not defined within a function and assignments are not allowed outside of a function.

```
if(__BLAST_NONDET == 3){
  ...
} else{
  ...
}
```

Figure 3.7: Code example which leads to undefined behaviour when using __BLAST_NONDET

rexpr	$eval_r(rexpr, \bar{s})$	
r_1 *nop* r_2	e_1 *nop* e_2	$e_1 \in \mathbb{N}, e_2 \in \mathbb{N}$
	$dV(e_1)$ *nop* $dV(e_2)$	$e_1e_2 \in Var, tc(e_1) = tc(e_2) = Prim$
	$dV(e_1)$ *nop* e_2	$e_1 \in Var, tc(e_1) = Prim, e_2 \in \mathbb{N}$
	$((addr_{\bar{s}}(e_1)$ *nop* $e_2), bt(e_1), l(e_1))$	$e_1 \in Var, tc(e_1) = Pnt, e_2 \in \mathbb{N},$ *nop* $\in \{+, -\}$
	$((addr_{\bar{s}}(e_1)$ *nop* $dV(e_2)), bt(e_1), l(e_1))$	$e_1e_2 \in Var, tc(e_1) = Pnt, tc(e_2) = Prim,$ *nop* $\in \{+, -\}$
	$(e_1^a$ *nop* $dV(e_2)), e_1^t, e_1^l)$	$e_1 \in \mathbb{P}, e_2 \in Var, tc(e_2) = Prim,$ *nop* $\in \{+, -\}$
	Error	otherwise
$r_1 < r_2$	1	$e_1 < e_2, e_1e_2 \in \mathbb{N}$
	0	$e_1 \geq e_2, e_1e_2 \in \mathbb{N}$
	1	$e_1 < dV(e_2), e_1 \in \mathbb{N}, e_2 \in Var, tc(e_2) = Prim$
	0	$e_1 \geq dV(e_2), e_1 \in \mathbb{N}, e_2 \in Var, tc(e_2) = Prim$
	1	$dV(e_1) < dV(e_2), e_1e_2 \in Var, tc(e_1) = tc(e_2) = Prim$
	0	$dV(e_1) \geq dV(e_2), e_1e_2 \in Var, tc(e_1) = tc(e_2) = Prim$
	Error	otherwise
$r_1 \geq r_2$	$1 - eval_r(r_2 < r_1, \bar{s})$	
$r_1 \leq r_2$	$eval_r(r_1 < (r_2 + 1), \bar{s})$	
$r_1 > r_2$	$eval_r((r_1 - 1) \geq r_2, \bar{s})$	
$r_1 = r_2$	$eval_r(r_1 \leq r_2, \bar{s}) \wedge eval_r(r_1 \geq r_2, \bar{s})$	
$r_1 \Vert r_2$	0	$((e_1 = 0, e_1 \in \mathbb{N}) \vee (dV(e_1) = 0, e_1 \in Var, tc(e_1) = Prim) \vee$ $(M_D(addr_{\bar{s}}(e_1)) = A_\perp, tc(e_1) \in \{Pnt, PrimPnt\}, e_1 \in Var)) \wedge$ $((e_2 = 0, e_2 \in \mathbb{N}) \vee (dV(e_2) = 0, e_2 \in Var, tc(e_2) = Prim) \vee$ $(M_D(addr_{\bar{s}}(e_2)) = A_\perp, tc(e_2) \in \{Pnt, PrimPnt\}, e_2 \in Var))$
	1	$((e_1 \neq 0, e_1 \in \mathbb{N}) \vee (dV(e_1) \neq 0, e_1 \in Var, tc(e_1) = Prim) \vee$ $(M_D(addr_{\bar{s}}(e_1)) \neq A_\perp, tc(e_1) \in \{Pnt, PrimPnt\}, e_1 \in Var)) \vee$ $((e_2 \neq 0, e_2 \in \mathbb{N}) \vee (dV(e_2) \neq 0, e_2 \in Var, tc(e_2) = Prim) \vee$ $(M_D(addr_{\bar{s}}(e_2)) \neq A_\perp, tc(e_2) \in \{Pnt, PrimPnt\}, e_2 \in Var))$
	Error	otherwise
r_1 && r_2	$eval_r(:(:r_1 \Vert :r_2), \bar{s})$	
$r_1 \mid r_2$	$or(e_1, e_2)$	$e_1e_2 \in \mathbb{N}$
	$or(e_1, dV(e_2))$	$e_1 \in nat, e_2 \in Var, dV(e_1) = Prim$
	$or(dV(e_2), dV(e_2))$	$e_1e_2 \in Var, dV(e_1) = dV(e_2) = Prim$
	Error	otherwise
r_1 & r_2	$eval_r(!(!r_1 \mid !r_2), \bar{s})$	
(c)*rexpr*		$tc(c) = Prim$:
	e	$e \in \mathbb{N}$
	$dV(e)$	$e \in Var, tc(e) = Prim$
	e	$e \in Var, tc(e) = Pnt$
	$(addr_{\bar{s}}(e), type(c), 0)^*$	$e \in Var, tc(e) \in \{array, FPnt\}$
		$tc(c) = Pnt$:
	$(e^a, bt(c), l(c))$	$e \in \mathbb{P}$
	e	$e \in \mathbb{N}, e = 0$
	$dV(e)$	$e \in Var, tc(e) = Prim, dV(e) = 0$
	$(M_D(addr_{\bar{s}}(e)), bt(c), l(c))^*$	$e \in var, tc(e) \in \{PrimPnt, Pnt\}$
	Error	otherwise

Table 3.6: Evaluation of binary and cast *rexpr*.

Type	Size
void	1
char	1
short	2
int	4
float	4
double	8

Table 3.7: Overview of type sizes for primitive data types

CHAPTER 4

Related Work

A memory model similar to ours can be seen in [11] which is used during the analysis of binaries. The abstract domain is called Bounded Address Tracking(BAT). It can be used to describe abstract memory states while our own memory model can only describe concrete memory states. Binaries do not contain type information thus BAT does not contain type information contrary to our memory model. The memory model contains a configurable bound which determines when information should be abstracted hence the name BAT. Furthermore the memory model is path sensitive which means that there are no merge operations in the analysis when two different paths are joined. This preserve information about the stack. Addresses in our memory model consist of a MemoryBlock and a Offset. Addresses in BAT are structured in a similar way namely MemoryRegion and a Offset. Because there are no variable information available during binary analysis there are two global address spaces in the BAT namely global and stack. In global the static information is saved while stack handles procedure calls by saving the return information on the stack. Additionally address spaces are created in order to deal with dynamic allocation of memory, i. e. malloc operations. Our memory model contains variable information and for each variable we save an address which contains the address space (i.e. a memory block which size depends on the type of the variable) for the variable.

Another CPA $\mathbb{E}$ for explicit analysis of variable values can be seen in [1]. $\mathbb{E}$ does not contain a memory model. Instead the abstract domain consists of a flat lattice where each element e is a function which assigns to the variables V a value e: $V \rightarrow \mathbb{Z} \cup \{\bot_Z, \top_Z\}$. $\top_Z$ indicates that the value of a variable is not known while $\bot_Z$ is used to indicate that no value can be assigned to a variable. $\top$ is the top element of the lattice and indicates that no variable value is known, i.e for each variable v, $\top(v) = \top_Z$. $\bot$ is defined in a similar way and indicates that no value can be assigned to the variables V. $\mathbb{E}$ can only analyze C programs which contain integer variable and no complex data structures or pointer operations while our interpreter CPA is capable of analyzing C programs with complex data structures and operations. $\mathbb{E}$ contains a set of precision Π which contains functions π which assigns to each variable V a bound ($\mathbb{N} \cup \{\infty\}$) which is used to determine how many values for a variable should be tracked during analysis. When for instance

a command $v := exp$ occurs and v has been assigned more than $\pi(v)$ values over the course of the analysis than for the new state e' the value of $e'(v) = \top_Z$. This means for the explicit analysis that certain informations about variables can be abstracted. This feature is currently not supported in our interpreter CPA where every information about the memory is tracked.

A program analysis which is suitable for programs with data structures like lists or trees can be seen in [4]. The analysis uses shape graphs and three valued logic to abstract data structures. A shape graph contains a set of vertices V. Each node $v \in V$ represents either a memory cell or a set of memory cells of a data structure. A shape class $\mathbb{S}$ contains a set of predicates over V. Each shape class represents a data structure class like double linked lists or trees and the set of shape classes S forms a lattice where elements closer to the bottom represent more complex data structures classes. A shape class $\mathbb{S}$ and a set of shape graphs G form a shape region $(\mathbb{S}, G)$. A shape region contains an assignment a for each n-ary predicate p^n in $\mathbb{S}$ with $a_{p^n} : V^n \to \{0, 1, 1/2\}$. Shape regions represent abstract states of the program memory. In order to refine the information about shape regions during program analysis an explicit heap analysis is used in [4] to gather additional information about the heap. This heap information is then used to refine the shape class and consequently the corresponding shape graphs of the shape regions via a shape class generator. Shape graphs could be suitable for our interpreter CPA to allow data structure abstractions and should thus be considered in future work.

A different approach for program analysis is presented in [12]. The tool Pin instruments programs and processes by injecting additional code. This can either be done when starting a new program or during runtime by attaching Pin to the process of the program. It is also possible to detach Pin from a process and continue execution of the process without instrumented code. With Pin it is possible to gather information about program states and execution paths or modify the behaviour of the process/program. Pin can be used to extract variable constraints for static program analysis. It is possible to track the memory or certain memory locations of a process with pin. Pin is operating on a binary level and thus no type information is available to pin when instrumenting a program or process. This means that it is not possible to map certain memory locations to variables without additional informations. Our interpreter CPA does not have these problems as it operates on source code level. Our interpreter CPA is thus better suited for the needs of the coverage analysis of CPA/Tiger.

CHAPTER 5

Experiments

In this Chapter we conduct a number of experiments to assess the performance of the new interpreter CPA. In Section 5.1 we compare the performance of our new interpreter CPA with the existing interpreter CPA. In Section 5.2 we test the performance of the interpreter CPA on programs with complex data structures. Programs analyzed by the interpreter CPA are preprocessed with CIL [13]. All experiments are performed on a computer with a Intel(R) Core(TM) i3-2100 processor which has 2 cores and uses 4 threads which run at 3.10 GHz. The memory size of the computer is 4GB.

5.1 Experiments for Integer Programs

For a set of integer programs we compare the time performance of the two interpreter CPAs. More exactly we measure for a given integer program the time for each interpreter CPA. For each integer program we measure the time needed for a fixed test case by the interpreters 1000 times. Each time we execute the interpreter twice and measure the second run. We do this to mitigate the effects of class loading in JUnit in order to get more accurate results. With this method is also the consecutive execution of the interpreter CPA is also taken into consideration. We then obtain for each integer program the average, the sample variance [10], and for each interpreter the maximum and minimum time needed in 1000 runs. We calculate the average and sample variance in the following way:

$$avg = \frac{1}{1000}\sum_{i=1}^{1000} d_i$$

and

$$var = \frac{1}{999}\sum_{i=1}^{1000} (d_i - avg)^2$$

where d_i is the time needed by an interpreter for an integer program.

	new interpreter [t in ms]				old interpreter [t in ms]			
name	avg	var	max	min	avg	var	max	min
kbfiltr_simpl1	21.366	64.214	67	13	19.347	40.755	50	11
kbfiltr_simpl2	20.662	98.927	153	8	15.179	62.211	64	8
floppy_simpl3	18.833	73.981	63	9	14.878	74.596	60	6
floppy_simpl4	13.749	47.467	60	6	11.641	39.377	52	5
cdaudio_simpl1	10.876	98.189	89	5	7.676	56.57	93	4
diskperf_simpl1	8.594	20.856	73	4	5.658	18.091	99	3
s3_clnt_1	23.877	23.187	54	16	20.298	33.903	59	12
s3_clnt_2	16.797	5.011	36	11	14.4	13.89	46	9
s3_clnt_3	16.594	4.608	33	10	11.764	6.335	29	7
s3_clnt_4	14.404	11.719	72	7	10.723	7.225	35	6
s3_srvr_1	67.617	175.434	121	31	63.499	127.165	125	25
s3_srvr_2	25.491	83.353	91	8	18.88	55.421	58	6
s3_srvr_3	14.881	20.051	35	7	10.006	6.951	30	6
s3_srvr_4	11.657	10.348	31	7	8.505	6.508	25	5
s3_srvr_6	12.368	14.497	41	6	9.599	9.398	27	5
s3_srvr_7	13.471	18.145	39	7	9.942	10.641	28	5
s3_srvr_8	11.679	14.701	33	6	10.184	11.131	39	6

Table 5.1: Comparison of the two interpreters for integer programs

The results of our experiments can then be seen in Table 5.1. We can observe that for the majority of the programs there is no significant[1] difference between the *avg* times of the two interpreters. In our test cases the new interpreter CPA is from 6% to 51% slower than the old interpreter CPA. There are integer programs (instance diskperf_simpl1) where the new interpreter CPA *avg* time is significantly worse than the old interpreter CPA.

5.2 Experiments for Complex Programs

We perform a series of experiments to investigate the performance of our new interpreter CPA when run on programs which contain complex data structures. In Table 5.2 we show the results of experiments which measured the time consumption in the same way as the experiment for integer programs. Again we measure the average time, the sample variance, the maximum, and minimal time consumption of our complex programs. We perform these measurements on an optimized version of the interpreter CPA and on an unoptimized version of the interpreter CPA. We do this to compare the time performance of the two versions. The optimized version minimizes the number of iterations over the version tree when the method getScope is called several times in PersMemory over the course of a transfer relation call (i.e. the interpreter element remains the same). We do this by saving a reference to the current scope in a variable and use this reference as long as the interpreter element does not change and no function is called or exited.

[1] i.e difference below 50% from the old interpreter time value

We can see that this optimization leads to a better time performance (the unoptimized version is between 2.6% and 9.5% slower than the optimized version of the interpreter CPA). We also add how often our transfer relation is called (i.e Transfer Relation Count - TRC) and the number of lines for each program (i.e. Lines of Code - LoC). The transfer relation count also measures the number of CFA edges that are traversed by the interpreter CPA. We can observe that diskperf which is 9828 lines of code long has the highest average time with 347.605ms and calls the transfer relation function 1894 times. The ratio of avg time to transfer relation count is thus 0.183ms per transfer-relation-call. The program kbfiltr has a similar time performance with 316.74ms but a much better ratio of 0.085ms per transfer relation call. The program s3_clnt_blast.01 has a much better time performance with 41.585ms and a better ratio with 0.007ms per transfer relation call. Finally the program drecusive which contains recursive function calls has the smallest time consumption with 5.135ms and a ratio with 0.05ms per transfer relation call.

In the next experiment we analyze the access behaviour of our fat node data structures. For this we define the *access depth* which is for a field the number of nodes of the version tree which need to be traversed - from the node representing the current state of the interpreter - to retrieve the current value of the field. With this we can create for all relevant fields of the objects of the memory model diagrams where then x-coordinate is the access depth for the field and the y-coordinate the number of accesses of the field. So we get the number of accesses with access depth for a given field.

Figures 5.1, 5.3, 5.4, 5.5, 5.6, are the diagrams for the complex program diskperf. We can see that the Figure 5.1 has the highest access depth and a uniform distribution of accesses of number 1 along the access depth axes. The cause for the high access depth is that during the initialization of the gobal variables only the global scope is accessed. The version history contains only one node with a reference to the scope, which was created when the scope itself was created. This means that each time the version tree grows by one node, i. e. which each statement during the initialization phase, the access depth grows by one. One method to mitigate this effect would be to make a new reference to the scope in a the current node of the version tree if the access depth succeeds a certain height. Another effect that can be seen in Figure 5.1 is a gap in the access depth between roughly 370 and 420. This gap is the effect of forward and extern function declarations. Whenever such a declaration occurs during the global initialization phase, there is no memory access but the version tree adds a new node.

When we compare Figure 5.1 which represents PersMemory for the optimized interpreter CPA with Figure 5.2 which is PersMemory for the unoptimized interpreter CPA we can see that the quantity for the data points in the unoptimized version is generally higher than in the optimized version of the interpreter CPA. For the Object Scope with Figure 5.3 we can see that the access depth is generally lower that for the object PersMemory with exception of one access which has an access depth around 370. In scope our interpreter CPA is accessing variables fields. Because variables are only created at the start of a new function the access depth for the variable grows with the length of the function. Also function calls have to be taken into consideration. This effect would explain the isolated data point in the Scope diagram. A variable is allocated at the

	optimized [t in ms]				unoptimized [t in ms]					
name	avg	var	max	min	avg	var	max	min	TRC	LoC
diskperf	347.6	12074.3	756	75	368.12	12672.7	806	88	1894	9828
kbfiltr	316.74	12529.2	688	41	346.9	12771.3	679	48	3700	6434
s3_clnt.blast.01	41.5	30.797	66	33	42.8	48.1	85	34	5350	3535
drecursive	5.135	2.8	14	3	5.27	2.57	14	4	87	58

Table 5.2: Time evaluation of new interpreter CPA for complex programs

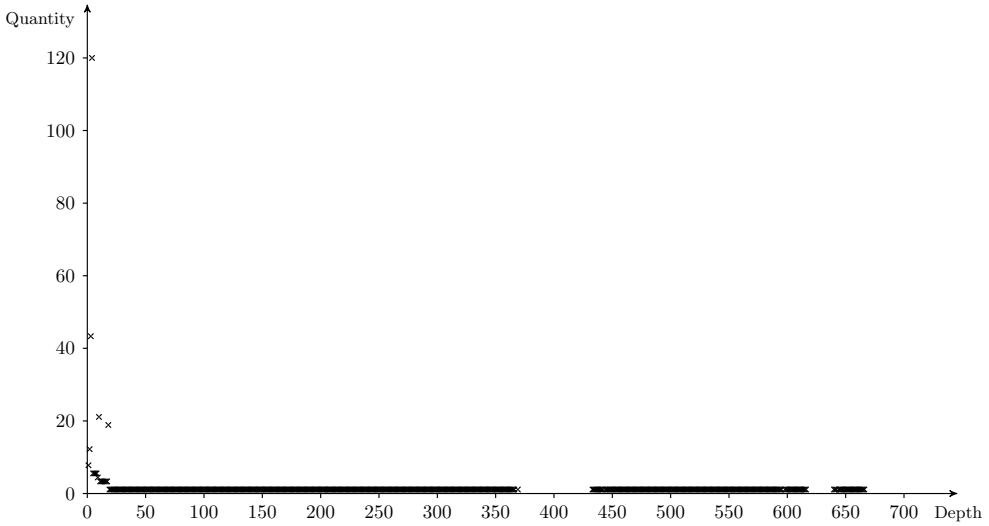


Figure 5.1: Diagram of objects PersMemory when calling method getcurrentscope

beginning of a function and then accessed after approximately 370 state transitions of our CFA. We can see that there are more accesses with low access depth than in PersMemory. The situations of the MemoryBlock, AddressMemoryCell, DataMemoryCell, and FunctionMemoryCell Objects is similar. All have low[2] access depths. We only have one data point for the FunctionMemoryCell Objects with access depth 1 and quantity 90. The general rule is the longer a certain element of a field has not been accessed since creation the longer the elements access depth will be when being accessed.

[2] i.e access depth below 100

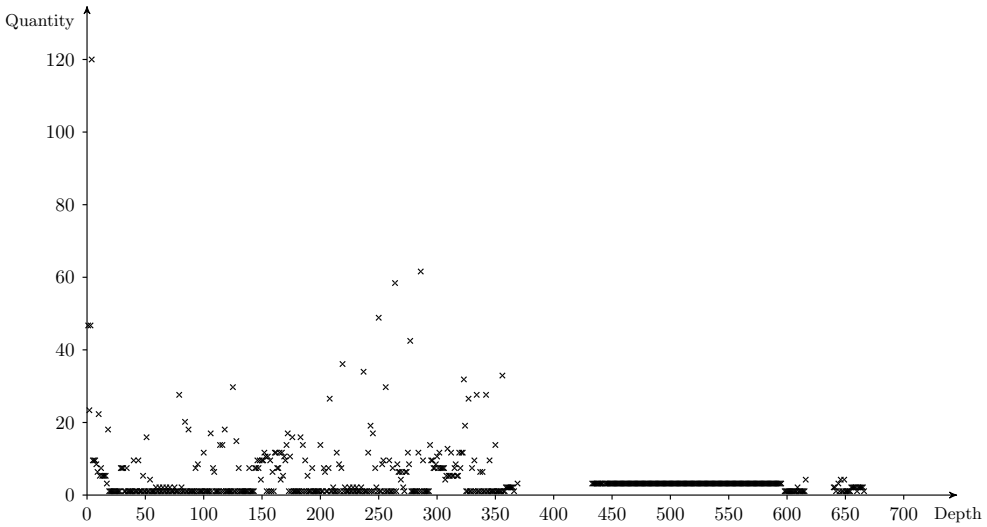


Figure 5.2: Diagram of objects PersMemory when calling method getcurrentscope (unoptimized)

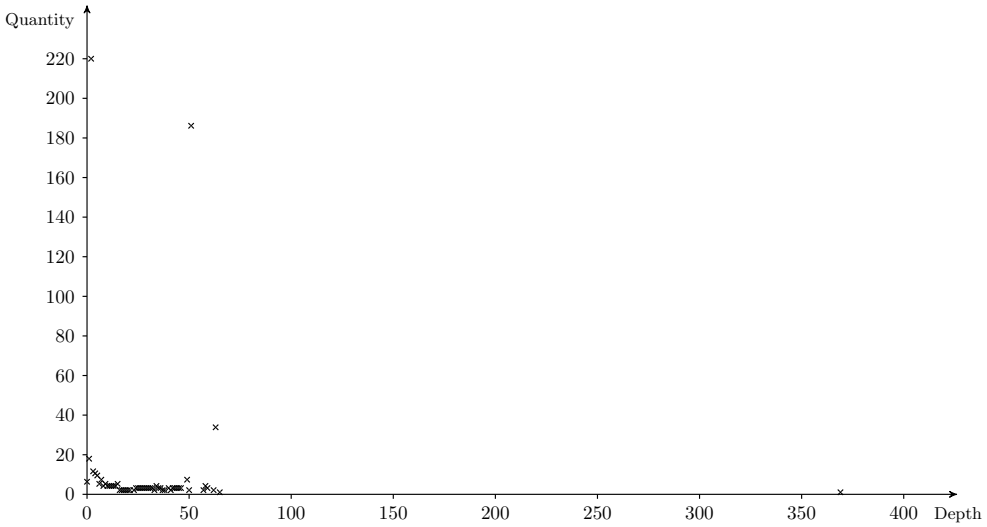


Figure 5.3: Diagram of objects Scope when calling method getVariable

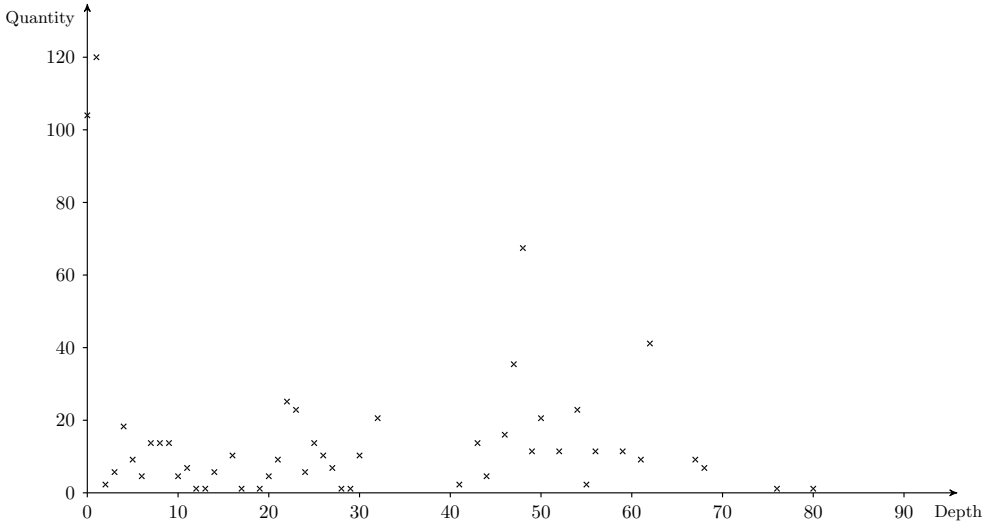


Figure 5.4: Diagram of objects MemoryBlock when calling method getMemoryBlock

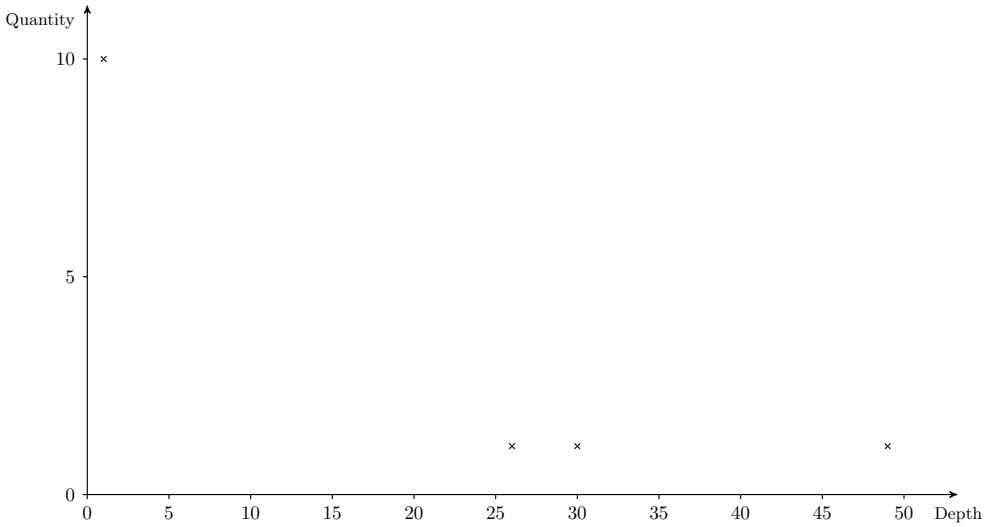


Figure 5.5: Diagram of objects AddressMemoryCell when calling method getAddress

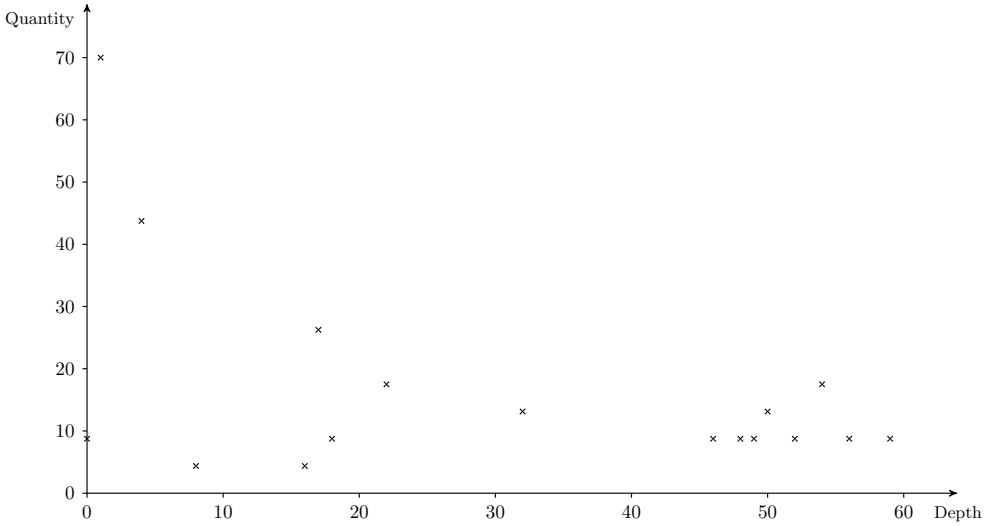


Figure 5.6: Diagram of objects DataMemoryCell when calling method getData

CHAPTER 6

Conclusion and Future Work

We implemented an interpreter CPA which contains a bit precise memory model. This enables CPAchecker to analyze test suites for programs with complex data structures. Our design of the memory model enables pointer arithmetic and prohibits out-of-memory pointer calculations. Our interpreter CPA can analyze programs which contain complex program constructs like recursions and function pointers. A future goal is to integrate the interpreter CPA into CPA/Tiger. In Chapter 3 we gave an overview over the memory model and the formal definition of our interpreter CPA. The interpreter CPA was not capable to analyze programs of reasonable[1] length due to the memory consumption of the initial memory model. Thus we adapted the Fat Node [7] data structure to minimize the memory consumption of our memory model. We conducted a series of experiments to evaluate the runtime of our new interpreter CPA. In our tests the new interpreter CPA is between 6% and 51% slower than the old interpreter CPA which is only able to analyze integer programs.

We introduced the notion of the access depth which is the number of iterations over a version history (which is a path in the version tree, i.e. fat node) to retrieve the value of a field. The Experiments indicate that certain objects like for instance PersMemory have a higher access depth than others. In order to minimize the access depth we suggest the following approach: Whenever we obtain the value of a field we measure the access depth. If the access depth exceeds a certain value we save a new reference for the value in the current node of the version history. Figure 6.1 illustrates the new approach where we add a new reference if the access depth is larger or equal to 3 when retrieving a field value. For field x we have for view v the access depth of 4 because it takes 4 iterations to retrieve the value for $x = 33$ at the root of the version tree. In the next step of our interpreter CPA the field x is again accessed. Without optimization we will then need 5 iterations for view v' to retrieve the value for x leading to an access depth of 5. With our suggested approach we only need 1 iteration to retrieve the value for x for view v'. With this

[1] i.e. more than 1000 lines of code

approach we can minimize for the accesses to the field scope their access depths in PersMemory.

Predicate analysis with adjustable block encoding(ABE) has been introduced in [6]. In predicate analysis with ABE refinements are not computed after every operation but rather after blocks of program statements where the the length of the block is adjustable (hence the name ABE). This improvements results in a better program analysis performance. The idea of performing certain operations after blocks of program statements can be also applied to the memory model of our interpreter CPA. Currently the interpreter CPA adds a new version node to the fat node after a program statement is interpreted by the interpreter CPA. We alter this approach so that a new version node is created only after a block of program statements has been interpreted by the interpreter CPA. The size of the block can either be configured in the interpreter CPA or determined by other CPAs if the interpreter CPA is used in a Composite CPA. Communication between the CPAs is realized via the strengthening operator. This new approach will decrease the memory consumption of the interpreter CPA and enables it to be used in conjunction with other CPAs like predicate abstraction. An example of this new method can be seen in Figure 6.2. For a given program we see a subset of its CFA. Our interpreter CPA is used in a composite CPA together with a predicate CPA. Nodes are colored gray if a new version node is added to the version tree otherwise white. We can see that in our initial memory model all nodes of the CFA are colored gray because after each operation a new version node is created. The updated memory model on the other hand gets information when to introduce new version nodes from the predicate CPA via the strengthening operator. So new nodes are only introduced at program locations 1, 4 and 9 leading to a smaller memory consumption.

When performing predicate analysis with ABE we are interested in the set of reachable program states S (esp. of an error state). Predicate analysis will return an over-approximation P of the set S, $S \subseteq [\![P]\!]$. Our interpreter analysis yields an under-approximation I (only one program path is investigated) of the set S, $[\![I]\!] \subseteq S$. The standard approach for combing these two analysis is to use them in a composite CPA. It is also possible to separate the two analysis and run the CPA algorithm for interpreter CPA and the predicate CPA in isolation. We would first perform the predicate analysis. When for a state an abstraction should be performed in the predicate analysis this new state is not added to the waitlist of the CPA algorithm. Thus our algorithm would return the set of abstract states where an abstraction should occur in the predicate analysis. Next we perform our interpreter analysis. Where the predicate analysis reached an abstraction point a new version node is introduced at the same program location. We would then get a set of concrete states for the program locations where abstractions should occur in the predicate analysis. In the final step we combine these two informations. This approach enables us to use the interpreter analysis to configure/control the predicate analysis and vice versa. It is for instance then possible to use our under-approximation (interpreter CPA) to compute the abstraction of our over-approximation (predicate CPA). This would have the effect of a more accurate over-approximation.

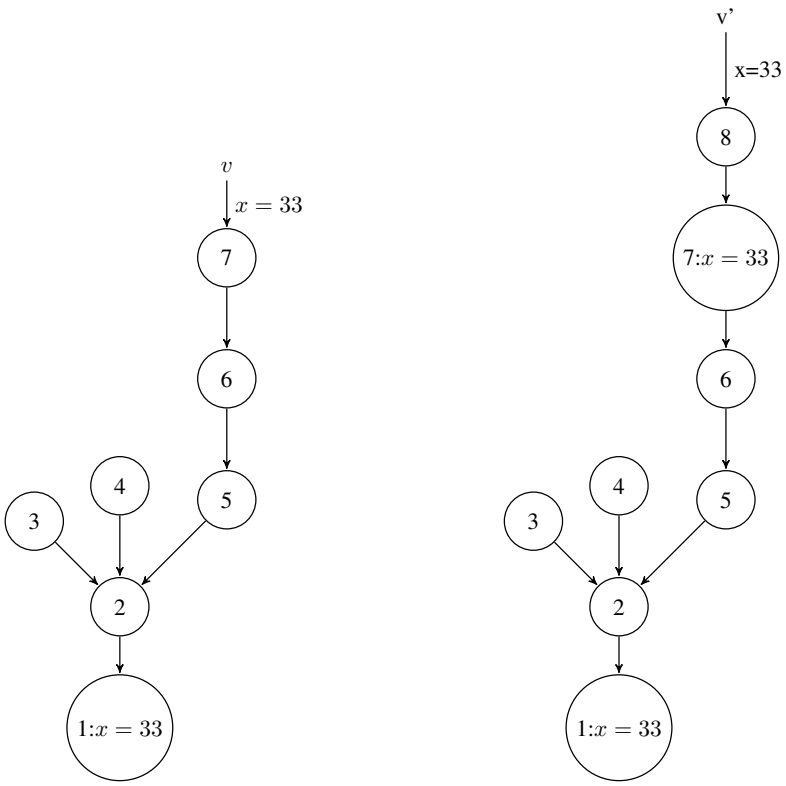


Figure 6.1: Example for new fat node approach that minimizes the access depth

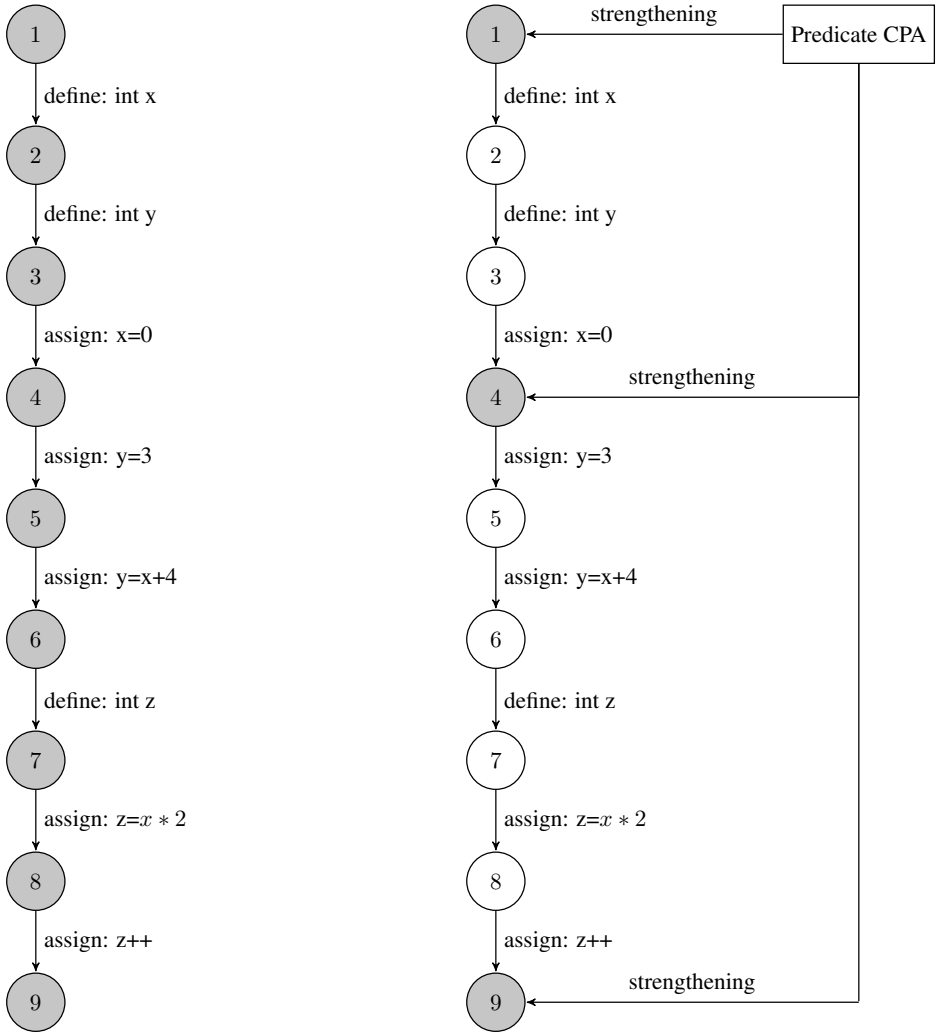


Figure 6.2: Example for adjustable-block-memory-model

Bibliography

[1] D. Beyer, T. A. Henzinger, and G. Theoduloz. Program analysis with dynamic precision adjustment. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 29–38, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[3] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007, Berlin, Germany, July 3-7)*, Lecture Notes in Computer Science, pages 504–518. Springer, 2007.

[4] Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz, and Damien Zufferey. Shape refinement through explicit heap analysis. In *Proceedings of the 13th international conference on Fundamental Approaches to Software Engineering*, FASE'10, pages 263–277, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] Dirk Beyer, Andreas Holzer, Michael Tautschnig, and Helmut Veith. Multi-goal reachability analysis. In *Proceedings of 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*, Lecture Notes in Computer Science. Springer, April 2012. Submitted.

[6] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 189–198, Austin, TX, 2010. FMCAD Inc.

[7] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent, 1989.

[8] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. How did you specify your test suite. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 407–416, New York, NY, USA, 2010. ACM.

[9] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. An introduction to test specification in FQL. In Sharon Barner, Ian G. Harris, Daniel Kroening, and Orna Raz, editors, *Proceedings of 6th International Haifa Verification Conference (HVC 2010)*, volume 6504 of *Lecture Notes in Computer Science*, pages 9–22. Springer, October 2010.

[10] James Lee Johnson. *Probability and Statistics for Computer Science*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[11] Johannes Kinder. *Static Analysis of x86 Executables*. PhD thesis, Technische Universitaet Darmstadt, 2010.

[12] Chi-keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[13] George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *In International Conference on Compiler Construction*, pages 213–228, 2002.