Definizione ed implementazione
di una analisi di points-to
per linguaggi C-like

Candidato    Stefano Soffia
Relatore     Roberto Bagnara
Correlatore  Enea Zaffanella

Anno Accademico 2007/2008
Abstract

The points-to problem is the problem of determining the possible run-time targets of pointer variables and is usually considered part of the more general aliasing problem, which consists in establishing whether and when different expressions can refer to the same memory address. Aliasing information is essential to every tool that needs to reason about the semantics of programs. However, due to well-known undecidability results and for all interesting languages that admit aliasing, the exact solution of nontrivial aliasing problems is not generally computable. This dissertation focuses on approximated solutions to this problem by presenting a store-based, flow-sensitive points-to analysis, for applications in the field of automated software verification. In contrast to software testing procedures, which heuristically check the program against a finite set of executions, the methods considered in this dissertation are static analyses, where the computed results are valid for all the possible executions of the analyzed program. We present a simplified programming language and its execution model; then an approximated execution model is developed using the ideas of abstract interpretation theory. Finally, the soundness of the approximation is formally proved. The aim of developing a realistic points-to analysis is pursued by presenting some extensions to the initial simplified model and discussing the correctness of their formulation. This work contains original contributions to the issue of points-to analysis, as it provides a formulation of a filter operation on the points-to abstract domain and a formal proof of the soundness of the defined abstract operations: these, as far as we now, are lacking from the previous literature.
Il problema della points-to è il problema di determinare i possibili valori che le variabili puntatore possono assumere durante l’esecuzione ed è comunemente considerato parte del più generale problema dell’aliasing, che consiste nello stabilire se e quando le diverse espressioni che compaiono nel codice sorgente possono riferirsi alla stessa locazione di memoria. L’informazione di aliasing è essenziale ad ogni strumento che ragiona sulla semantica dei programmi. Tuttavia, a causa dei noti risultati di non decidibilità, per tutti i linguaggi rilevanti ai fini dell’aliasing, una soluzione esatta a questo problema non è in generale calcolabile. Questa dissertazione si concentra sulla ricerca di soluzioni approssimate a questo problema presentando una analisi di points-to store-based e flow-sensitive, per l’applicazione nel campo della verifica automatica del software. Contrariamente alle procedure di testing, le quali in modo euristico controllano il programma rispetto ad un insieme finito di esecuzioni, i metodi descritti in questa dissertazione possono essere classificati come analisi statiche, i cui risultati sono validi rispetto a tutte le possibili esecuzioni del programma analizzato. Nella dissertazione viene presentato un linguaggio di programmazione semplificato ed un relativo modello di esecuzione; quindi viene sviluppata una approssimazione di questo modello usando le idee della teoria dell’interpretazione astratta. In fine, la correttezza dell’approssimazione descritta è formalmente dimostrata. Allo scopo di presentare una analisi di points-to realistica, sono inoltre presentate alcune estensioni al modello semplificato descritto inizialmente e la loro correttezza è discussa. Questo lavoro contiene un contributo originale al problema dell’analisi di points-to in quanto presenta la formulazione di un operazione di filtro su questo tipo di dominio astratto e dimostra formalmente la correttezza delle operazioni astratte definite: elementi che, per quanto in nostra conoscenza, non sono presenti nella letteratura precedente.
## Contents

1 Presentation .................................................. 1
   1.1 Introduction ............................................. 1
      1.1.1 The Aliasing Problem ................................ 1
      1.1.2 A Static Analysis .................................... 4
      1.1.3 Applications .......................................... 7
      1.1.4 Background ........................................... 9
   1.2 Developing a Sound Approximation of the Aliasing Problem ...... 12
      1.2.1 Notation ............................................... 12
      1.2.2 The Execution Model and Its Operations ................. 13
      1.2.3 The Abstract Interpretation Approach .................... 14
      1.2.4 Queries ............................................... 15
      1.2.5 Representation of the Abstract Alias Domain .......... 18
      1.2.6 The Store Based Approach ............................ 21
   1.3 Precision Limits of the Alias Query Representation ............. 24
   1.4 The State of the Art ....................................... 27
   1.5 Thesis Purpose ........................................... 31
   1.6 Thesis Contribution ....................................... 31

2 The Analysis Method ........................................... 32
   2.1 The Domain ................................................. 32
   2.2 The Language ............................................... 33
      2.2.1 Assignment ............................................ 34
      2.2.2 Filter ................................................ 35
   2.3 Examples .................................................. 37
   2.4 Results .................................................... 49
      2.4.1 Notation ................................................. 49
      2.4.2 Concrete Assignment ................................... 50
      2.4.3 Observations on the Domain ........................... 51
      2.4.4 Results of Correctness ................................ 53
      2.4.5 Proofs ................................................ 54
   2.5 Precision Limits ............................................ 76
      2.5.1 Precision of the Points-To Representation .............. 76
      2.5.2 Precision of the Presented Method ....................... 85
      2.5.3 A Final Consideration .................................. 94

3 Extensions .................................................... 96
Organization

Chapter 1. The chapter starts with a brief description of the points-to and alias problems; then an overview of the possible applications, of the available approaches and of the current state of the art is presented.

Chapter 2. A simplified language and a simplified execution model are introduced; the execution model comprehends the memory model and the operations that acts on it. Subsequently, an approximated memory model and the approximated operations are presented. Following the methodology of the abstract interpretation theory, the soundness of the approximated execution model is proved. Finally, some informal considerations about the precision of the abstraction are presented.

Chapter 3. In order to present a realistic points-to analysis, some extensions to the model introduced in the previous chapter are presented and a possible implementation of the approximated memory model is described.

Chapter 4. This chapter draws the conclusions of the thesis and it discusses some of the possible future developments of the present work.
1 Presentation

1.1 Introduction

1.1.1 The Aliasing Problem

In imperative programming languages the concept of memory location is of main importance; it refers to an entity able to keep a finite quantity of information across the subsequent steps of the computation. The concept of variable is then developed as a way to refer to memory locations. In the different languages, different constructs allow for the composition of variable names so as to form expressions (Listing 1.1). From the use of these constructs comes the possibility to refer to the same memory location with different expressions. In the literature, two expressions referring to the same memory location are said to be aliases; the set of pairs of alias expressions is commonly referred to as alias information and the aliasing problem is known as the problem of analyzing the alias information of a program. Due to the many mechanism that can lead to the generation of aliases, the aliasing problem is complex even to characterize. The following paragraphs show how the different constructs of the C language can affect the alias information.

Aliasing From the Use of Arrays

The example presented in Listing 1.2 shows how, through the use of the array’s indexing mechanism, the aliasing problem is influenced by the value of integer variables. As shown by Listing 1.3 also the converse holds — the value of pointer variables, typically considered a alias-related issue, can influence the value of integer variables.

```
1 struct S {
2     struct S *l, *r;
3     int key;
4 } a[10];
5 int i;
6 ...
7 a[i].l->key = ...
```

Listing 1.1: different constructs of the C language can be used to compose variables into expressions. Note at line 7 the use of the dereference operator, of the index and field selectors in the same expression. Many are the available constructs and complex is the problem of analysing all their possible interactions.
int a[10], i, j;
...
if (i == j) {
  ...
  a[i] = a[j];
  ...
}

Listing 1.2: this example shows how the use of arrays may produce aliasing. At line 5 the variables ‘i’ and ‘j’ hold the same value; then the expressions ‘a[i]’ and ‘a[j]’ denote the same memory location, i.e., they are aliases.

int a[10], *p, *q, dist;
...
dist = q - p;

Listing 1.3: the value assigned to the variable ‘dist’ at line 3 depends on the distance between the elements referred to by the pointers ‘p’ and ‘q’.

Aliasing From the Use of Pointers

The simple example in Listing 1.4 shows how the use of pointers can produce aliasing. In the C language the support of pointers is particularly flexible and powerful. For instance, multiple levels of indirections are allowed (Listing 1.5). These characteristics make the development of alias analyses for the C language a challenging problem. The study of the aliasing problem requires also to cover recursive data structures; the use of these can produce particularly complex alias relations (Listing 1.6).

Aliasing Subproblems

Due to the many aspects that must be taken into account in order to provide a complete coverage of the aliasing problem, different area of research have been developed; as a result, in the literature a wide range of analyses is available, which encompasses all the alias subproblems — while a pointer analysis attempts to determine the possible run-time values of pointer variables, a shape analysis focuses on the precise approximation of the

int a, *p;
p = &a;
...

Listing 1.4: after the execution of line 2, the pointer variable ‘p’ contains the address of the variable ‘a’; then the expressions ‘*p’ and ‘a’ are aliases.
Listing 1.5: at line 2 the address of \texttt{a} is assigned to \texttt{p}; as a consequence, \texttt{*p} and \texttt{a} become aliases. At line 3 the address of \texttt{p} is assigned to \texttt{pp}; as a consequence, \texttt{*pp} and \texttt{p} become aliases. Hence, also the expressions \texttt{*pp} and \texttt{*p} are aliases. Finally, by applying the transitive property, it is possible to conclude that \texttt{**pp} and \texttt{a} are aliases too.

Listing 1.6: this example shows how recursive data structures can affect the aliasing problem. After the assignment at line 7, the expressions \texttt{head.next->key}, \texttt{head.next->next->key} —and more generally each expression of the form \texttt{head.(next->)}^\texttt{n} \texttt{key} with \( n \in \mathbb{N} \) — are all aliases of \texttt{head.key}. Even a simple example can produce an infinite set of alias pairs.

1.1 Introduction
aliasing relations produced by recursive data structures; whereas a numerical analysis is required to track the value of array’s indices.

1.1.2 A Static Analysis

The goal of this dissertation is to present an automated method able to prove certain alias properties of programs given in input. In the following we use the term alias analysis to refer to the general and theoretical ideas to approach the alias problem; whereas we use the term alias analyzer to stress the focus on the implementation of an automated analysis. We are interested in defining a static analysis. Commonly, in the context of software analysis, the adjective static referred to the term analysis designates a class of methods that avoid the actual execution of the examined program. In other words, a static analysis can be described as the process of extracting semantic information about a program at compile time. Static analysis techniques are necessary to any software tool that requires compile-time information about the semantics of programs. Consider indeed the following points.

• The termination problem is undecidable; as a consequence any method that requires the execution of the program is not guaranteed to terminate.

• If the execution of the program is performed then the computational complexity of the analysis is bounded from below by the computational complexity of the analyzed program.

• Testing a program on some executions can prove the presence of errors; however, unless all of the possible executions are tried, it cannot prove the absence of errors. More generally, since a program can have an unbounded number of distinct executions, testing can only prove that a property holds on some executions, but it cannot prove that it holds always.

Hence, the existence of analysis methods that avoid the actual execution of the program is motivated by the presence of constraints on the costs of the analysis, the need of predictability of these or the need to verify a property against all of the possible executions. Usually, the results of an alias analysis are only an intermediate step of the computation of a complete static analysis tool; this means that an alias analysis is commonly intended to answer to questions formulated by other automatic analyses. For instance, compilers are the most common tools that exploit the alias information — almost all of the modern compilers include some kind of alias analysis. From the practical perspective, the kind of queries that are posed to the alias analyzer is greatly influenced by the final application; whereas from the theoretical point of view it is useful to assume that the questions posed to the alias analysis are always of the form: does the property $P$ hold on all/some executions of the program?

One Program, Many Executions

Generally, the flow of the execution depends not only on the program’s source code but also on external sources of information, e.g., the user’s input or a random number


```c
int a, b, *p;
p = &a;
if (rand())
    p = &b;
...
```

Listing 1.7: at line 2 the address of ‘a’ is assigned to the variable ‘p’; then at line 3, during all executions, ‘*p’ is an alias of ‘a’. At line 4 the address of ‘b’ is assigned to ‘p’, but this statement is executed only when at line 3 the call to rand() returns a non-zero value. Therefore, it is possible to prove that there exists at least one execution path that reaches line 5 in a state where ‘*p’ is an alias of ‘a’ and also there exists at least one execution that reaches line 5 in a state where the same property is false.

generator; when many executions paths are possible, a property may hold on some but not on all the possible executions (Listing 1.7). In the following we refer to a function declared as ‘int rand()’ as a source of non-determinism; we assume that this function always halts, that it can return zero and not-zero values and that it has no side-effects on the caller.

The Aliasing Problem Is Undecidable

The problem of determining the alias properties of a program is undecidable; it is indeed possible to reduce a problem that is well known to be undecidable, the halting problem, to the aliasing problem. In the sequel, we refer to a function declared as ‘int turing(int n)’; we assume that (1) this function is defined somewhere in the source code and it emulates the execution on the input n of some Turing machine; (2) the result of the execution of the emulated Turing machine is returned to the caller as the return value of the function; (3) calling this function has no side effects on the caller environment.

Listing 1.8 highlights how the aliasing problem is influenced by the halting problem. For this reason the aliasing problem is formulated assuming the reachability as hypothesis. This assumption is not always valid but it is safe, or conservative. In Listing 1.8 it is not possible to tell if line 5 will ever be reached; however, in that case what would happen?

More generally the question is — if the execution reaches the program point p does the property P hold at p? The results of the analysis are then expressed as an implication of the kind — if p is reached then P holds. However, even in this weaker form, the aliasing problem is still undecidable. Consider for instance Listing 1.9 where line 7 is reached if and only if the call ‘turing(K)’ at line 3 halts; in this case the value of ‘p’ is determined

1The idea and the motivations behind this approach are similar to those that drive the development of Hoare’s logic for partial correctness specification, opposed to the total correctness specification, both introduced in [Hoa03]. The concept of Hoare’s triple for partial correctness is introduced — it is a triple \( \{ P \} \) \( C \) \( \{ Q \} \) where \( C \) is a command of a given programming language and \( P \) and \( Q \) are two propositions expressed in some fixed first order logic language. Informally, in Hoare’s logic the triple \( \{ P \} \) \( C \) \( \{ Q \} \) is said to be true if whenever \( C \) is executed in a state satisfying \( P \) and the execution of \( C \) terminates then the resulting output state satisfies \( Q \).
Listing 1.8: at line 4 the call to the function ‘turing’ starts the computation of the Turing machine. Suppose that the call halts; in this case the execution reaches line 5 causing an error due to a dereferenced null pointer. However, the problem of telling whether the execution of a Turing machine will ever halt is undecidable [HMRU00] — there exists no algorithm able to tell for each possible value of ‘K’ if line 5 will ever be reached by the execution; thus if there exists an execution path where a null pointer is dereferenced.

Listing 1.9: an example of the possible interactions of the aliasing problem and other undecidable problems. There exist no algorithm able to tell for every ‘K’ if there exist an execution that reaches line 7 in a state such that ‘p’ points to ‘a’.

by the return value of ‘turing(K)’. As a consequence of Rice’s theorem [HMRU00], also assuming that ‘turing(K)’ halts, there exist no algorithms able to tell for every ‘K’ if the execution reaches line 7 in a state where ‘p’ points to ‘a’.

Summing Up

This section summarizes the various possibilities just presented. Let P be an alias property and ¬P its negation. There exist four possible cases.

1. The property P holds on all of the possible executions or equivalently, ¬P never holds (Listing 1.7).

2. The property P holds on some but not on all of the possible executions; that is, there exists at least one execution in which P holds and also there exists at least one execution in which ¬P holds (Listing 1.7).

3. The property P holds on some executions but it is not known if it holds always; that is there exists at least one execution in which P holds but it is unknown whether there exists an execution in which ¬P holds (Listing 1.10).
int K = ...;
int *p, a;
p = &a;
if (rand())
  if (turing(K))
    p = 0;
*p = 1;

Listing 1.10: line 5 is reached only when the return value of the call to ‘rand()’ evaluates to true. Thus, line 6 is reached only when the execution reaches line 5 and the call ‘turing(K)’ halts and the return value evaluates true. Certainly there exists executions that reach line 7 in a state where ‘p’ points to ‘a’. However, also assuming that ‘turing(K)’ halts, there exists no algorithm able to tell for every K if there exist an execution path that reaches line 7 with ‘p’ equal to null.

void f(int *p) {
  ...
  *p = 0;
  ...
}

Listing 1.11: analyzing this fragment of code without any aliasing information would require a worst-case assumption about the locations pointed by ‘p’, that is, all the possible targets of an ‘int*’ can be modified by the assignment at line 3.

4. It is not known if there exists an execution in which P holds and also it is unknown whether there exists an execution in which ¬P holds (Listing 1.9).

For instance, suppose that P expresses the absence of some kind of error. The first of the listed cases is the optimal case: it has been proved that no errors are possible. The second case is as much positive: it has been proved that there exists at least one erroneous execution, that is the program contains a bug. In the third and the fourth case it is unknown, i.e., the absence of errors cannot be proved. However, assuming the reachability as hypothesis, alias analyses cannot prove the result described in the second case. In other words, every static analysis that assumes the reachability as hypothesis can only prove that P holds always. In this sense, testing procedures are complementary to static analyses techniques.

1.1.3 Applications

The alias information is required by many static analyses; this is due to the following fact: analyzing an indirect assignment, and generally an indirect memory reference, without knowing alias information requires to assume that the assignment may modify almost anything and, under these hypotheses, it is unlikely that the client analysis will be able
to deduce any useful result (Listing 1.11). For what concerns the final application, there are two main areas where the aliasing information is commonly used.

- Optimization and parallelization; used in compilers and interpreters.
- Programs semantics understanding and verification; used in debugging/verifier tools.

These two uses have vastly different requirements on alias analyses. For compiler oriented applications there exist some upper bound on how much precision is useful. There are various studies [HP00, HP01] that state that this upper bound is reached by the current state of the art. For the use in program understanding/verification the picture is different; in this case there is instead a lower bound on precision, below which, alias information is pretty useless. It is commonly believed that the spectrum of techniques currently available does not fully covers the requirements of this kind of use: more research work is necessary.

**Client Analyses**

This section presents a brief list of the most common static analyses that require the aliasing information.

**Mod/Ref analysis:** this analysis determines what variables may be modified/referenced at each program point. This information is subsequently used by other analyses, such as reaching definitions and live variable analysis. Each dereference in the program generates a query of the alias information to determine the referenced objects that are thus classified as modified or referenced depending on the context in which the dereference operator occurs. For example, in assignment statements, the objects referred by the last dereference of the lhs are marked as modified, all other objects referred in the evaluation of the rhs and the lhs are instead marked as read.

**Live variable analysis.** It is common to many imperative languages that the life of a local variable starts at the point of definition and ends at the end of the scope that contains the definition. At the extent of minimizing the memory usage of the compiled program, while keeping unchanged its semantics, it is possible to defer the creation to the point where the variable is first assigned and anticipate its destruction to the last point where its value is used. The live variable analysis tries to compute this information that is useful to compilers for register allocation, detecting the use uninitialized variables and finding dead assignments.

**Reaching definitions analysis:** this analysis determines what variables may reach (in an execution sense) a program point. This informations is useful in computing data dependence among statements, which is an important step for the process of code-motion and parallelization.

---

2Here the term ‘referenced’ means that the value of the object is read.
Interprocedural constant propagation: this analysis tracks the value of constants all over the program and uses this information to statically evaluate conditionals with the goal of detecting if a branch is unreachable; thus allowing the detection of unreachable code.

1.1.4 Background

Probably due to the different areas of application, historically this field of research has treated as separate two fundamental aliasing-related problems: the may alias and the must alias problem. If the general interest of aliasing-related static analyses is the study of how different expressions lead to the same memory location, these two specializations can be characterized as follows.

**May alias:** it tries to find the aliases that occur during some execution of the program.

**Must alias:** find the aliases that occur on all the executions of the program.

Results exist that confirm that the former problem is not recursive\(^3\) while the latter is not recursively enumerable\(^4\) [Lan92]. In recent developments the same concepts are also expressed in terms of possible and definite alias properties. The term definite alias property is used to designate an alias property that holds on every possible execution; whereas a possible alias property \(P\) is such that both \(P\) and \(\neg P\) cannot be proved to be definite. Unfortunately, the mismatch between the naming and the notation used in the published works is not limited to the case just described. For instance, in the literature the names pointer analysis, alias analysis and points-to analysis are often uses interchangeably. As suggested by [Hin01], we prefer to consider the points-to analyses as a proper subset of the alias analyses. An alias analysis attempts to determine when two expressions refer to the same memory location; whereas a points-to analysis [And94, EGH94, HBCC99] is focused in determining what memory locations a pointer can point to. Points-to methods are also characterized by the same representation of the aliasing information. As described in [Hin01], the representation of the alias information is only one of the several parameters that can be used to categorize alias analyses.

**Representation.** For the representation of alias information various options are possible.

- **Complete alias pairs:** with this representation all the alias pairs produced by the analysis are stored explicitly.
- **Compact alias pairs:** only a subset of alias pairs is kept explicitly. The complete relation can be derived applying the dereference operator, the transitivity and symmetry properties to the pairs explicitly stored.

---

\(^3\)A problem \(P\) is said to be recursive, or decidable, if there exists an algorithm that terminates after a finite amount of time and correctly decides whether or not a given input belongs to the set of the solutions of \(P\).

\(^4\)A recursively enumerable problem \(P\) is a problem for which there exist an algorithm \(A\) that halts on a given input \(n\) if and only if \(n\) is a solution of \(P\).
Listing 1.12: A program that exposes a simple alias relation.

Points-to pairs: this representation tracks only the relations between the pointers and the pointed objects. The complete alias relation can be derived from the points-to information in a way similar to what done for the compact alias pair representation. This process is informally described in [Ema93].

For instance, the alias relation generated by the sequence of assignments in Listing 1.12 can be represented using the points-to form as

\[
\{ \langle p, i \rangle, \langle q, p \rangle, \langle r, i \rangle \}\.
\]

This corresponds to the complete alias pair set:

\[
\{ \langle *p, i \rangle, \langle *q, p \rangle, \langle **q, i \rangle, \langle *r, i \rangle, \langle r, p \rangle, \langle *r, *p \rangle, \langle r, *q \rangle, \langle *r, **q \rangle \}.
\]

Note that these representations — complete, compact and points-to — are listed in order of decreasing expressive power — the rules of deduction used to infer the complete alias relation from the compact and the points-to format impose a precise structure on the relation. In the next we presents some examples to show how the points-to representation can be less precise than the alias representation (Section 2.5). On the other hand these deduction rules allow to reduce the set of pairs that have to be explicitly represented thus decreasing the cost of the analysis. Note also that, due to recursive data structures, the complete alias relation may contain an infinite number of pairs. If one of the possibilities to overcome this problem is to adopt a compact or a points-to representation, other solutions, specialized in the handling of recursive data structures, exist. These methods use quite different formalism from the ones presented here and they have generated a quite independent field of research that is named shape-analysis. An example of these alternative representations is briefly described in Section 1.2.5.

Flow-sensitivity. The question is whether the control-flow information of the program is used by the analysis. By not considering control-flow information —therefore computing only a conservative summary of it— flow-insensitive analyses compute one solution for either the whole program or for each function [And94, Ste96, HBCC99], whereas a flow-sensitive analysis computes a solution for each program point [EGH94, HBCC99]. Therefore, flow-insensitive methods are generally more efficient but less precise than flow-sensitive ones.

In this case we have omitted to explicitly write the alias pairs that can be obtained by symmetrically closing this relation.

1.1 Introduction
Context-sensitivity. The point is if there is a distinction between the different callers of a function, that is if the caller-context information is used when analyzing a function. If this is not the case, the information can flow from one call site (say caller A) through the called function (the callee) and then back to a different call site (say caller B) thus generating a spurious data flow in the computed solution on the code of the caller B. Whenever a static analysis combines information that reaches a particular program point via different paths some accuracy may be lost. An analysis is context-sensitive to the extent that it separates information originating from different paths of execution. Because programs generally have an unbounded number of potential paths, a static analysis must combine information from different paths — in this sense, the context sensitivity is not a dichotomy but rather a matter of degree.

Heap modeling. The analysis of heap-allocated objects requires different strategies from that of stack-allocated and global memory objects. First because heap objects have a different life-cycle with respect to automatic and globals variables; second, the term heap modeling, is commonly but improperly used to refer to the modelling of recursive data structures as these are usually allocated on the heap. Various trade-offs between the precision and the efficiency exist also for this problem.

- The simpler solution consists in creating a single abstract memory location to model the whole heap [EGH94].
- Another solution distinguishes between heap allocated objects on the basis of the program point in which they are created, that is objects are named by the creating statement (context-insensitive naming.)
- A more precise solution names the objects not only by the program point of the creating statement but with the whole call path (context-sensitive naming.) For example, this means that if the program contains a user defined function for memory allocations (e.g., a wrapper of the ‘malloc’ function) then the analysis is able to discern objects created by different calls of the allocation routine.
- Shape analysis methods adopts a quite different approach to the problem of naming locations, which is based on the expression used to refer to the memory location.

Whole program. Does the analysis method require the whole program or can a sound solution be obtained by analyzing only its components? In the current panorama of software development, component programming and the use of libraries are becoming more and more popular. This trend requires the capability to analyzing fragments of code as the whole program may not be available [LLV].

Language type model. In strongly typed languages, the type information —that can be easily extracted from the source code using common compiler techniques— can be used by the alias analysis to deduce affordable informations about the layout of pointers. This information, joined with other assumptions on the memory model that
usually accompany this kind of languages, can greatly simplify the formulation of
the alias analysis. However, as noted in [WL95], a pointer analysis algorithm cannot
safely rely on high-level type information for C programs. Because of arbitrary type
casts and union types, the defined types can always be overridden. This means
that type information cannot be used to determine which memory locations may
contain pointers. To be safe, an analysis must assume that any memory location
could potentially contain a pointer to any other location. Similarly, any assignment
could modify pointers, even if it is defined to operate on non-pointer types.

Aggregate modeling. This characterization regards how objects of compound types and
arrays are treated. The main question is whether the elements of these aggregate
types are distinguished or collapsed into one object. The choice of the analyzed
language is of main relevance: this task results particularly complex to address in
weakly-typed languages such as C/C++; in these languages the same memory area
can be read using different types. An analysis that aims to precisely track pointers
to fields must then consider the possible overlapping between the memory layouts
of the different types. In strongly typed languages like Java this difficulty does not
exist, as these languages do not allow for reading the memory with a type different
from that used for the allocation.

1.2 Developing a Sound Approximation of the Aliasing
Problem

This section presents informally the approach used for the definition of the points-to
analysis.

1.2.1 Notation

Before proceeding, some clarifications about the used notation are necessary. Let \( A \) and
\( B \) be two sets. We write ‘\( A \rightarrow B \)’ to denote a total function from the set \( A \) to the set
\( B \); we write ‘\( A \rightarrowto B \)’ to denote a partial function from \( A \) to \( B \). We use ‘\( S \triangleq A \rightarrow B \)’
to denote the set of all (total) functions from \( A \) to \( B \); whereas we write ‘\( f : A \rightarrow B \)’ to
mean that \( f \) is a (total) function from \( A \) to \( B \).

We denote as ‘\( \text{Bool} \)’ the set \( \{0, 1\} \) and as ‘\( \text{Bool}^\sharp \)’ the set \( \wp(\{0, 1\}) \); for convenience of
notation we use ‘\( \bot \)’ to refer to the empty element and ‘\( \top \)’ for the \( \{0, 1\} \) element. We refer
to the complete lattice associated to the set \( \text{Bool} \) as the structure \( \langle \text{Bool}, \subseteq, \cup, \cap, \{0, 1\}, \emptyset \rangle \).

Let \( n, m \in \mathbb{N} \), where \( n < m \), we write ‘\( \{n, \cdots, m\} \)’ to denote the set of the naturals
from \( n \) to \( m \), i.e., \( \{i \in \mathbb{N} \mid n \leq i \leq m\} \).

Let \( A, B, C \) be finite sets. We write ‘\# A’ to mean the cardinality of the set \( A \). Let
\( f : A \rightarrow \wp(B) \) and \( g : B \rightarrow C \) and \( a \in A \) be such that \( \# f(a) = 1 \); for convenience of
notation we write ‘\( g(f(a)) \)’ to mean \( g(b) \) where \( f(a) = \{b\} \).
1.2.2 The Execution Model and Its Operations

Though our work is ideally targeted for the C language, we need to introduce some kind of formal execution model. The standard of the C language has indeed many implementation defined issues that every execution model is required to specify in order to provide a working environment for the execution of programs. The literature provides several of such formalizations [BHZ08]; however, for this presentation many of the details would be useless. With the aim of keeping a simple notation, we introduce the following concepts. We denote with ‘Expr’ the set of the expressions of the language. With execution model we mean a formally specified computing device able to execute programs written in the analyzed language. With memory description, or simply memory, we mean a description of the state of the execution model at some step of the computation. Fixed the execution model, we denote with ‘Mem’ the set of the memory descriptions. We make few assumptions about the structure of the memory model; we assume that a memory is composed by a set of memory locations denoted as ‘Loc’. Given a memory description \( m \in \text{Mem} \) and a location \( l \in \text{Loc} \), we denote with \( m[l] \) the information that \( m \) stores at the location \( l \). We also assume the existence of a partial evaluation function

\[ \text{eval} : \text{Mem} \times \text{Expr} \rightarrow \text{Loc}. \]

In the real world, the execution of a program acts in different ways on the memory structure of the computing machine. With the aim of formalizing these interactions, we introduce the concept of operation; an operation is defined as a partial function

\[ \text{op} : \text{Mem} \times \text{Ext} \rightarrow \text{Mem} \]

where ‘Ext’ is an unspecified set that formalizes the use of external information. Note that we have specified \( \text{op} \) as a partial function — this is needed to model the fact that the possible actions that can be performed on the memory structure are not defined on all of the possible states. For example, to process the return statement of a function, the stack of the memory must contain at least one activation frame. By aiming to perform a static analysis, we are interested in determining all the possible memory descriptions that can be generated at a specified program point. To express the transition from a set of memory descriptions to another as a consequence of an operation, we extend the definition of the operation \( \text{op} \) to sets. Let

\[ \text{op} : \wp(\text{Mem}) \times \text{Ext} \rightarrow \wp(\text{Mem}) \]

be defined as follows. Let \( M \subseteq \text{Mem} \) and \( e \in \text{Ext} \), then

\[ \text{op}(M, e) \stackrel{\text{def}}{=} \{ \text{op}(m, e) \mid m \in M \land \text{op}(m, e) \text{ is defined} \}. \]

\[ ^6 \text{In the formalization of Turing machines [HMRE90], an instantaneous description is a complete description of the computing device at one of the steps of the computation; here, with memory description we mean an instantaneous description of the chosen execution model.} \]

\[ ^7 \text{Now we use the term memory location a synonym of memory address. Basically, with location we mean a tag that can be used to identify the information stored in the memory description.} \]
Example 1. Consider the modifications to the memory triggered by the declaration of a local variable. To formalize this event we introduce an operation $\text{new}_s$, which takes the memory description of the execution prior to the declaration, plus some information about the declaration. In this case, the set ‘Ext’ represents the type of the declared variable and, if present, the expression used as initializer. The returned memory describes the properly updated execution state. Now suppose that the set $M \subseteq \text{Mem}$ represents the possible memory configurations at a given program point $p$, which is immediately followed by a local variable declaration. Let $e \in \text{Ext}$ be the information associated to the declaration; then we express the set of all possible memory configurations resulting from the declaration as $\text{new}_s(M, e)$.

1.2.3 The Abstract Interpretation Approach

As shown in the introduction, the aliasing problem is undecidable. Following the approach proposed by the abstract interpretation theory [CC77, CC79, CC92], to overcome this limitation we proceed by developing a computable approximation of the execution model and its operations.

Definition 1. (Concrete domain of the aliasing problem.) We define the concrete domain of the aliasing problem as the complete lattice generated by the powerset of $\text{Mem}$

$$\langle \wp(\text{Mem}), \subseteq, \cup, \cap, \emptyset, \text{Mem} \rangle$$

Then we need to develop an abstract counterpart of the chosen execution model — an abstract domain $\text{Mem}^\sharp$ that provides an approximation of the concrete domain $\wp(\text{Mem})$. We formalize $\text{Mem}^\sharp$ as a complete lattice

$$\langle \text{Mem}^\sharp, \subseteq, \cup, \cap, \bot, \top \rangle.$$

To formally express the semantics of the approximation we provide a concretization function

$$\gamma: \text{Mem}^\sharp \rightarrow \wp(\text{Mem}).$$

We say that a memory description $m \in \text{Mem}$ is approximated, or abstracted, by an element of the abstract domain $m^\sharp \in \text{Mem}^\sharp$ when $m \in \gamma(m^\sharp)$. The formalism also requires the definition of an abstract counterpart $\text{op}^\sharp$ of the concrete operations $\text{op}$

$$\text{op}^\sharp: \text{Mem}^\sharp \times \text{Ext} \rightarrow \text{Mem}^\sharp.$$

To prove the soundness of the proposed abstract model by it is necessary to show that for all $m^\sharp \in \text{Mem}^\sharp$ holds that

$$\text{op}(\gamma(m^\sharp), e) \subseteq \gamma(\text{op}^\sharp(m^\sharp, e));$$

that is, the approximation provided by the abstract operation $\text{op}^\sharp$ is safe with respect to the concrete operation $\text{op}$. Beyond the operations already defined on the concrete
execution model Mem, the formalization of the abstraction requires the definition of other operations that can be described as

$$\text{op}^\sharp: (\text{Mem}^\sharp)^n \times \text{Ext} \rightarrow \text{Mem}^\sharp;$$

along with the corresponding concrete counterpart,

$$\text{op}: \varphi(\text{Mem})^n \times \text{Ext} \rightarrow \varphi(\text{Mem}).$$

The soundness of these operations is expressed in the same way, that is for all $$m_1^\sharp, \ldots, m_n^\sharp \in \text{Mem}^\sharp$$

$$\text{op}(\gamma(m_1^\sharp), \ldots, \gamma(m_n^\sharp), e) \subseteq \gamma(\text{op}^\sharp(m_1^\sharp, \ldots, m_n^\sharp, e)).$$

These additional operations include for instance, the ‘meet’ and ‘join’ operations of the domain. With a slight change of notation, this definition can be accommodated to describe also the requirement of correctness on the partial order ‘$$\subseteq$$’, i.e.,

$$\gamma(m_0^\sharp) \subseteq \gamma(m_1^\sharp) \iff m_0^\sharp \subseteq m_1^\sharp.$$  

1.2.4 Queries

This section introduces the concept of query on a domain. A query defines an interface on the domain, it helps to isolating the relevant information from other uninteresting details. When the analysis process is composed by more abstract domains, the use of queries is useful to formalize the interactions between them. More details on this approach can be found in [CLV94]. In the following we show how queries can also be used to formalize the semantics of the abstraction, that is how the concretization function $$\gamma$$ can be expressed in terms of queries. Fixed the number of arguments $$n$$, we denote with ‘Query’ the space of the concrete query functions and with ‘Query$$^\sharp$$’ the space of the abstract query functions,

$$\text{Query} \overset{\text{def}}{=} (\text{Expr})^n \rightarrow \text{Bool};$$

$$\text{Query}^\sharp \overset{\text{def}}{=} (\text{Expr})^n \rightarrow \text{Bool}^\sharp.$$ 

The concrete query domain is then defined as the complete lattice generated by the powerset of ‘Query’

$$\langle \varphi(\text{Query}), \subseteq, \cap, \cup, \emptyset, \text{Query} \rangle;$$

whereas the abstract query domain is defined as a complete lattice on the set Query$$^\sharp,$$

$$\langle \text{Query}^\sharp, \subseteq, \cap, \cup, \bot, \top \rangle,$$

where ‘$$\subseteq$$’ is the point-wise extension of the ordering of Bool$$^\sharp$$; $$\bot$$ and $$\top$$ are the minimum and maximum elements of Query$$^\sharp$$ with respect to this ordering, respectively; ‘$$\cap$$’ and ‘$$\cup$$’

\* Such as the $$\text{NEW}_*$$, the assignment and all other operations required to define the behaviour of the concrete execution model.

1.2 Developing a Sound Approximation of the Aliasing Problem
are the obvious point-wise extensions of \( \text{Bool}^{\sharp} \)'s operations. Note that ‘Query’ can be seen as subset of ‘Query^{\sharp}'. From this fact, the concretization function

\[ \gamma : \text{Query}^\sharp \rightarrow \wp(\text{Query}), \]

is defined as, for all \( q \in \text{Query} \) and \( q^\sharp \in \text{Query}^\sharp \),

\[ q \in \gamma(q^\sharp) \iff q \subseteq q^\sharp. \]

In order to define the semantics of \( \text{Mem}^{\sharp} \) in terms of queries, it is necessary to describe other two steps of the concretization. First we have to define how the query has to be performed on the concrete domain, that is how to extract the relevant information from a concrete memory. In symbol,

\[ \gamma : \text{Query} \rightarrow \wp(\text{Mem}). \]

Also, we have to define how the query has to be performed on the abstract domain, i.e,

\[ \gamma : \text{Mem}^{\sharp} \rightarrow \wp(\text{Query}^\sharp). \]

The semantics of the abstraction \( \text{Mem}^{\sharp} \) is then defined as the composition of these three steps (Figure 1.1.)

**The Alias Query**

The following definitions present the formal meaning of the statement — \( e_0 \) and \( e_1 \) are aliases in \( m \in \text{Mem} \). Basically, two expressions are considered aliases in a memory description when they evaluate to the same memory location.

---

9Consider indeed the injection \( f : \text{Query} \rightarrow \text{Query}^{\sharp} \) that maps every \( q \in \text{Query} \) to a \( q^\sharp \in \text{Query}^\sharp \) such that, for all \( e \in \text{Expr}^n \), \( q^\sharp(e) = \{ q(e) \} \).
Listing 1.13: in this example the execution can reach line 4 in many different states due to the different values that the variable ‘a’ can assume. However, considering a type based alias analysis—that is assuming that the analysis tracks only the value of pointer variables—each of the possible \( m \in \text{Mem} \) carries the same aliasing information.

Definition 2. (Concrete alias query domain.) Let

\[
\text{AliasQ} \overset{\text{def}}{=} (\text{Expr} \times \text{Expr}) \to \text{Bool}.
\]

We define the concrete alias query domain as the complete lattice generated by the powerset of AliasQ

\[
\langle \varphi(\text{AliasQ}), \subseteq, \cup, \cap, \emptyset, \text{AliasQ} \rangle.
\]

Definition 3. (Concrete alias query semantics.) Let

\[
\gamma : \text{AliasQ} \to \varphi(\text{Mem})
\]

be defined as follows. Let Alias \( \in \) AliasQ and \( m \in \text{Mem} \); then we define \( m \in \gamma(\text{Alias}) \) when, for all \( e, f \in \text{Expr} \) holds that

\[
\text{Alias}(e, f) = \begin{cases} 1, & \text{if } \text{eval}(m, e) = \text{eval}(m, f); \\ 0, & \text{otherwise}. \end{cases}
\]

Given a concrete memory description \( m \in \text{Mem} \), we denote as Alias\(_m\) the concrete alias relation that abstracts \( m \); also we call Alias\(_m\) the alias information of the memory \( m \). As anticipated in Section 1.2.4, the alias query Alias acts as an interface onto \( m \in \text{Mem} \) selecting the interesting details; this idea is shown in Listing 1.13.

Definition 4. (Abstract alias query domain.) Let

\[
\text{AliasQ}^\sharp \overset{\text{def}}{=} (\text{Expr} \times \text{Expr}) \to \text{Bool}^\sharp.
\]

We define the abstract alias query domain as the complete lattice generated by the powerset of AliasQ\(^\sharp\)

\[
\langle \text{AliasQ}^\sharp, \subseteq, \cup, \cap, \bot, \top \rangle.
\]

The semantics of the abstract alias query domain

\[
\gamma : \text{AliasQ}^\sharp \to \varphi(\text{AliasQ})
\]
struct List {
    struct List *n;
    int key;
} *x;

struct Tree {
    struct Tree *l, *r;
    int key;
} *y;

Listing 1.14: In this code, two recursive structures, List and Tree, are defined.

as already specified in Section 1.2.4, is defined as

\[
\text{ALIAS} \in \gamma(\text{ALIAS}^\sharp) \quad \text{def} \quad \text{ALIAS} \subseteq \text{ALIAS}^\sharp.
\]

The last step required in order to complete the definition of the semantics of the abstraction, that is from Mem\(^\sharp\) to \(\wp(\text{ALIAS}^\sharp)\) (Figure 1.1), depends on the details of the chosen approximation method Mem\(^\sharp\). The next section presents some of the available approaches.

1.2.5 Representation of the Abstract Alias Domain

By looking forward to the realization of an alias analyzer, another problem arises. A realistic implementation cannot aim to directly represent abstract alias queries (Definition 4). As demonstrated in Listing 1.6, there can be an infinite number of aliasing pairs making impossible a direct representation. In this sense, the domain Mem\(^\sharp\) introduces an additional layer of abstraction providing a representation suitable for the implementation.

Techniques For Approximating the Alias Information

In the literature, different classes of methods exist. One of these is the class of access-path based methods. A brief description of an access-path based method is reported below. Another class is identified by the name of store based methods; more details on these are presented in Section 1.2.6.

A Notable Example of Access-Path Based Approximation

In the literature, the term access-path is used to design a simplified form of language expressions. A notable example of access-path based method for the approximation of the abstract alias query domain (Definition 4) is presented in the Ph.D. dissertation of A. Deutsch [Deu94]. In this proposal the elements of the abstract domain Mem\(^\sharp\) are formalized as pairs \(m^\sharp = \langle P, C \rangle\) where \(P\) is a set of pairs of symbolic access paths and \(C\) is a set of constraints on \(P\). A symbolic access path is an approximation of a set of

1.2 Developing a Sound Approximation of the Aliasing Problem
Concrete memory $m \in \text{Mem}$

Concrete alias query $\text{ALIAS} \in \text{AliasQ}$

Abstract alias query $\text{ALIAS}^\sharp \in \text{AliasQ}^\sharp$

Access-path based abstraction (Deutsch)

Points-to based abstraction.

Abstract memory $m^\sharp \in \text{Mem}^\sharp$

Figure 1.2: A representation of the abstraction relations under discussion. Arrows should be read as ‘is abstracted by.’
expressions\textsuperscript{10} the concretization of a symbolic access path is defined using a mechanism similar to regular expressions. Consider for instance the code presented in Listing 1.14, and let $\langle P, C \rangle \in \text{Mem}^2$ be an abstract memory description of the program at line 11, such that $C = \{ i = j \}$. The set of constraints $C$ has the set of solutions

$$\{ (n, n) \}_{n \in \mathbb{N}}$$

in the variables $(i, j)$. Let $p \in P$ be the pair of symbolic access paths

$$p \overset{\text{def}}{=} \langle x(\cdot \to n)^i \to \text{key}, y(\cdot \to l, \cdot \to r)^j \to \text{key} \rangle.$$  

The semantics of $p$ is a set of pairs of concrete expressions and it can be computed by replacing the occurrences of the variables $i$ and $j$ found in the symbolic access paths of $p$, with the values given by the solutions of $C$. For instance, by replacing the occurrences of the index ‘$j$’ with the integer 2 in the symbolic access path

$$y(\cdot \to l, \cdot \to r)^j \to \text{key},$$

we obtain the regular expression

$$y(\cdot \to l, \cdot \to r)^2 \to \text{key},$$

that can be finally translated into the following set of expressions

$$\{ y(\cdot \to l) \to \text{key}, y(\cdot \to l) \to \text{key}, y(\cdot \to l) \to \text{key}, y(\cdot \to l) \to \text{key} \}.$$ 

Depending on the considered solution of $C$, the pair $p$ approximates different sets of alias pairs. For instance, using the solution $(0, 0)$ we have

$$\langle x(\cdot \to n)^0 \to \text{key}, y(\cdot \to l, \cdot \to r)^0 \to \text{key} \rangle = \{ (x \to \text{key}, y \to \text{key}) \}.$$ 

With the solution $(1, 1)$ we have

$$\langle x(\cdot \to n)^1 \to \text{key}, y(\cdot \to l, \cdot \to r)^1 \to \text{key} \rangle = \{ (x \to n \to \text{key}, y \to l \to \text{key}), (x \to n \to \text{key}, y \to r \to \text{key}) \}.$$ 

Using the solution $(2, 2)$ we obtain

$$\langle x(\cdot \to n)^2 \to \text{key}, y(\cdot \to l, \cdot \to r)^2 \to \text{key} \rangle = \{ (x \to n \to n \to \text{key}, y \to l \to l \to \text{key}), (x \to n \to n \to \text{key}, y \to r \to l \to \text{key}),$$

$$\langle x \to n \to n \to \text{key}, y \to l \to r \to \text{key} \rangle, (x \to n \to n \to \text{key}, y \to r \to r \to \text{key}) \}.$$ 

Generally, $C$ is a set of constraints on a tuple of indices $I = \{ i_1, \ldots, i_n \}$. The indices of $I$ also occur in the symbolic access paths of $P$. To each solution $S: I \to \mathbb{N}$ of $C$ corresponds

\textsuperscript{10}The term \textit{symbolic access path} comes from the original paper [Deu94] and it actually means an abstraction of the concept of \textit{expression}. With our notation, the term \textit{abstract expression} would be probably used instead.
a different alias query expressed as a set of pairs of (concrete) expressions. Given a solution $S$ to $C$, the corresponding alias query, say $P(S)$, can be obtained from $P$ by replacing every occurrence of the index $i_k$ in $P$ with the solution $S(i_k)$, for each index $i_k$ of $I$. As shown above, this replacement yields a set of pairs of no-longer-symbolic access paths. Seen as regular expressions, these no-longer-symbolic access paths are transformed in a set of pairs of expressions. The semantics of Mem$^\#$ can be finally expressed in terms of queries as follows. Let $\text{alias}^\# \in \text{AliasQ}^\#$ and $(P, C) \in \text{Mem}^\#$. We say that $\text{alias}^\# \in \gamma((P, S))$ when

$$
\exists S \text{ solution of } C. \forall e, f \in \text{Expr} : \text{alias}^\#(e, f) \in \{\top, 1\} \implies (e, f) \in P(S).
$$

Note that this has two main consequences.

- This formulation is unable to represent definite alias properties, that in terms of abstract alias queries correspond to the answer ‘1’; the approximation provided by this method is indeed also called may-alias information. For example, at line 3 of Listing 1.7 in all of the possible executions, the expression ‘*p’ is an alias of ‘a’. However, this method is only able to tell that ‘*p’ is possibly an alias of ‘a’, that in terms of abstract alias query corresponds to the outcome $\top$.

- Every solution of $C$ corresponds to a different abstract alias query, whereas, as we will show in Chapter 2, the concretization of a points-to abstraction consists of only one abstract alias query. As a consequence, this representation of the alias information is able capture relational information, whereas points-to methods cannot.

To represent the set of integer constraints $C$ different options exist. The literature on this field provides a wide choice of numeric lattices offering different trade-off between accuracy and efficiency; from non relational domains —like arithmetic intervals and arithmetic congruences— up to relational domains [BHZ08]. The alias analysis just described is completely parametric with respect to the chosen numeric domain and —due to the large availability of numeric domains— this is a point of strength of the method.

### 1.2.6 The Store Based Approach

This section introduces some concepts that are useful to understand the approach of store based methods. Points-to analyses are special cases of stored-based methods. The idea common to all store based methods is the explicit introduction of formal entities to represent memory locations. As in the concrete situation we use the notation ‘Loc’ to represent the set of the memory locations; now we introduce the notation Loc$^\#$ to denote the set of the abstract locations. Store based information usually consist of some sort of compact representation of a binary relation ‘$P$’ on the set of the abstract locations. To bind the concept of location to the concept of expression an environment function is provided. Basically, the environment function is needed to resolve identifiers into abstract locations. Denoting with ‘Identifiers’ the set of identifiers, an environment function can
be described as

\[ \text{id: Identifiers } \rightarrow \text{Loc}^\sharp \]

Since identifiers are the base case for the definition of the Expr set, from the elements \( \langle P, \text{id} \rangle \) it is possible to build the abstract evaluation function

\[ \text{EVAL: (Mem}^\sharp \times \text{Expr}) \rightarrow \emptyset(\text{Loc}^\sharp). \]

The ‘EVAL’ function is defined inductively following the inductive definition of the ‘Expr’ set. The details depend on the chosen language and intermediate representation; a complete definition is presented in the following chapter. The ‘EVAL’ function is then used to define the semantics of Mem\(^\sharp\) in terms of abstract alias queries; for instance, a possible definition is the following. Let \( m^\sharp \in \text{Mem}^\sharp \) and Alias\(^\sharp\) \in \text{AliasQ}^\sharp\), we say that Alias\(^\sharp\) \in \gamma(m^\sharp)\) when, for all \( e, f \in \text{Expr} \) holds that

\[ \text{Alias}^\sharp(e, f) = \begin{cases} 0, & \text{if } \text{EVAL}(m^\sharp, e) \cap \text{EVAL}(m^\sharp, f) = \emptyset; \\ \top, & \text{otherwise}. \end{cases} \]

This is an oversimplified definition, presented only to give an idea of how a store based approximation can answer to alias queries; note indeed that we have omitted to consider definite alias informations. Due to the introduction of the set of abstract locations Loc\(^\sharp\), the semantics of the abstract domain Mem\(^\sharp\) can also be expressed in terms of the value of locations; we have an abstraction function

\[ \alpha: \text{Loc} \rightarrow \text{Loc}^\sharp; \]

where, for each \( l \in \text{Loc} \), \( \alpha(l) \) denotes the abstract location that approximates \( l \). Given \( l \in \text{Loc}^\sharp \) and an abstraction \( m^\sharp \in \text{Mem}^\sharp \), we denote as \( m^\sharp[l] \) the value of the abstract location \( l \) in the abstract memory description \( m^\sharp \). Now, let \( m \in \text{Mem} \) and \( m^\sharp \in \text{Mem}^\sharp \); then we have

\[ m \in \gamma(m^\sharp) \ \overset{\text{def}}{\iff} \left( \forall l \in L : m[l] \text{ is defined } \Rightarrow m[l] \in \gamma\left( m^\sharp[\alpha(l)] \right) \right). \]

This formulation of the semantics of Mem\(^\sharp\) can be applied to points-to methods only, but it has the advantage that it can be generalized to the case where the points-to domain is coupled with some other abstract domain, provided that its semantics can be expressed in the same way. Moreover, the concretization function expressed in terms of locations is more similar to the algorithms actually implemented as client analyses are more likely to reason in terms of “pointed locations” than in terms of “aliased expressions”.

**Practical Considerations on Store Based Methods**

Despite the commonalities of store based methods described in the previous section, from the implementation perspective many different options exist. For example Emami et al.
Listing 1.15: in this code the assignments at line 7 and line 8 contain expressions the dereference operator occurs more than once.

Listing 1.16: this is the simplified version of line 7 and line 8 from Listing 1.15. Note the use of the additional variables tmp0, tmp1 and tmp2. Note that all expressions contain at most one occurrence of the dereference operator.

[Ema93, EGH94] and also [Ghi95] do not define a complete abstract evaluation function EVAL. Instead, they prefer to work on a simplified version of the code. To accomplish this they introduce a simplification phase to be performed before the actual analysis. Basically, this phase breaks the occurrences of “complex” expressions into a simpler form by means of the introduction of auxiliar variables and assignments. For example, in the simplified code all the expressions contain at most one occurrence of the dereference operator. Listing 1.16 presents the result of the simplification phase applied to the code in Listing 1.15. Having reduced all the expressions to a base form, the definition of the evaluation function EVAL is greatly simplified. However, the simplification phase has also other side effects. First, assuming to have already proved the correctness of the analysis, its results are valid on the code resulting from the simplification phase; to obtain any formal result on the original code it must be proved that the applied simplification does not change the semantics of the code. From the point of view of the efficiency, it is unclear whether or not a simpler evaluation function EVAL allows a more efficient analysis. In both cases the same steps of evaluation must be made; the difference is that in one case temporaries are made explicit. In our approach we have chosen to avoid the simplification phase as we believe that enabling the analyzer to see complete expressions can improve the precision.

Example 2. Assume that at line 3 of Listing 1.17 holds the following points-to information.
Listing 1.17: an example of ‘complex’ expression occurring in the condition of an if statement.

Listing 1.18: the result of the simplification of Listing 1.17.

\[
P(pp) = \{q, p\}, \\
P(p) = \{a\}, \\
P(q) = \{b\}.
\]

Looking at the condition of the if statement at line 3, it is possible to refine the points to information of line 4; that is, inside the ‘then’ branch, ‘pp’ points only to ‘p’. However, on the simplified code (Listing 1.18), looking only at the simplified condition of the if statement, it is not possible to infer any useful information about ‘pp’, as it occurs no more in the expression. It is possible to prove that ‘temp’ points only to ‘a’, but this information is useless as ‘temp’ is a auxiliary variable introduced by the simplification phase and thus it is not used elsewhere.

### 1.3 Precision Limits of the Alias Query Representation

This section presents an example that highlights the limitations of the alias query representation; alias queries (Section 1.2.4) fail to represent relational information. For instance, the code presented in Listings 1.19 and 1.20 induce the same abstract alias query; in particular in Listing 1.19 the alias representation is unable to express that, at line 4, if ‘p’ points to ‘a’ then ‘q’ points to ‘c’. This situation is illustrated in Figures 1.3 and 1.4.
Below an extract of the concrete alias query \( \text{ALIAS}_{m_0} \) induced by the concrete memory description \( m_0 \in \text{Mem} \). Above a graphical representation of the points-to information associated to the same memory.

As above, on the concrete memory description \( m_1 \).

The abstract alias query

\[
\text{ALIAS}^\downarrow \in \text{AliasQ}^\downarrow
\]

is defined as \( \text{ALIAS}_{m_0} \sqcup \text{ALIAS}_{m_1} \) and it is the most precise abstract alias query that abstracts the set of concrete alias queries \( \{ \text{ALIAS}_{m_0}, \text{ALIAS}_{m_1} \} \).

Figure 1.3: a representation of the alias query induced by the code in Listing 1.19.
Listing 1.19: in this code only two possible executions exist. At line 4, knowing the value of one of the two pointers ‘p’ and ‘q’, it is possible to determine the value of the other.

Listing 1.20: in this code four executions are possible; at line 7, also knowing the value of one of the two pointers ‘p’ and ‘q’, it is not possible to determine the value of the other.

An example of a spurious element of the concretization of \(\text{ALIAS}^2\). We have that

\[
\text{ALIAS}_{m_2} \in \gamma(\text{ALIAS}^2).
\]

However, \(\text{ALIAS}_{m_2}\) can not be generated by the program.

Another spurious element of the concretization of \(\text{ALIAS}^2\). Note that:

\[
\gamma(\text{ALIAS}^2) = \gamma(\text{ALIAS}_{m_0} \uplus \text{ALIAS}_{m_1}) = \{ \text{ALIAS}_{m_0}, \text{ALIAS}_{m_1}, \text{ALIAS}_{m_2}, \text{ALIAS}_{m_3} \}.
\]
1.4 The State of the Art

Static analysis originally concentrated on Fortran and it was predominately confined to a single procedure (intra-procedural analysis). Since the emergence of the C language, static analysis of programs with dynamic storage and recursive data structures has become a field of active research producing methods of ever increasing sophistication. In [Hin01] it is noted that, during the past two decades, over seventy-five papers and nine Ph.D. theses have been published on alias analysis, leading the author to the question — given the tomes of work on this topic, haven’t we solved this problem yet? The answer is that though many interesting results have been obtained, still many “open questions” remain. As shown in the introduction, also limited to the analysis of pointers, the aliasing problem is still undecidable [Lan92]; therefore, the main question that arise approaching it is about the desired trade-off between the efficiency of the algorithm and the precision of the approximated solution computed. A wide range of worst-case time complexities is available: from almost linear [Ste96] to exponential [Deu94]. The current research effort is proceeding in at least two distinct directions: improving the efficiency of the analyses while keeping the actual precision and increasing the precision of the approximation while keeping a reasonable computational costs.

Improving the Efficiency

Again in [Hin01], the problem of scalability is listed among the “open questions”. About this topic two distinct efforts are currently active and both proceed toward the goal analysing programs of ever increasing size. Today, flow-insensitive analyses [Ste96, LLV] can quickly analyze million-line programs. It is commonly believed that the precision provided by these fast methods is sufficient to satisfy ordinary compiler-oriented client analyses [Hin01]; but definitely they do not suffice for verifier-oriented applications [OR06, WMD08]. On the other side various works [HBCC99] have increased the efficiency of the more precise but slower flow-sensitive methods with respect to the initially proposed methods [EGH94]. It must be noted that some studies [EGH94, HP00, HP01] show that client analyses improved in efficiency as the pointer information was made more precise because the input size to the client analysis becomes smaller; on average, this reduction outweighed the initial cost of the pointer analysis. However, these studies focused on typical compiler oriented analyses — no data is available for the field of program understanding/verification.

Improving Precision

Another goal of the current research effort is to improve the precision without sacrificing the scalability. As for the scalability issue, nowadays there are two main directions in which researchers are investigating to improve the current state of the art. The first area of investigation tries to reconsider the notion of safety by loosening the soundness constraints on the analysis. The other direction of investigation tries to recognize the areas of the source code that needs to be analyzed with greater accuracy; the idea is to perform a quick alias analysis on the whole program and then refine the first results only in those regions of the code where more precision is needed. In other fields of the static
analysis research this idea has yield to the formalization of the concept of demand-driven analysis [OR06, WMD08]. Demand-driven methods can avoid the costly computation of exhaustive solutions: given an initial query, the analysis contains the logic to detect what other information are needed to answer it and then it proceeds by recursively formulating a new set of queries. It is still an open question whether the precise alias analyses currently available -that is flow- and context-sensitive analyses and shape analyses- can be reformulated in a demand-driven fashion [Hin01].

Different Notions of Safety

A reading of the literature available for the field reveals that there exist two slightly different notions of safety, which are determined by the different areas of application. Compiler targeted analyses are required to produce a safe approximation of the alias information for every standard-compliant program, allowing thereby the analyzer to assume that the analyzed program is standard-compliant\footnote{For some notion of standard-compliant; there exists different possible language standards, hence different notions of standard-compliance.} From [WL95]

The possibility of non-pointer values [stored inside pointer variables] is not always important. For example, when a location is dereferenced, we can assume that it always contains a pointer value, since otherwise the program would be erroneous.

On the other hand, for software verification tools, the conformance of the analyzed program to the standard is not an hypothesis but one of the theses that need to be proved. For example, a desirable feature for a verifier tool would be to signal if a dereferenced pointer may hold an undefined or a null value. For analyses that cannot simply ignore the possibility of errors, the approach called \(\theta\)-soundness is usually applied [CDNB08]: when the analysis detects the possibility of an error, then the program point is marked with a warning and the analysis proceeds assuming that the condition that led to the error is not verified. For example, if we have that to the pointer ‘\(p\)’ corresponds the points-to set \{\(a,\)null\} —i.e, ‘\(p\)’ may point to the variable ‘\(a\)’ or be null— then the analysis of the statement ‘\(*p\)’ would produce a warning for a possible dereferenced null pointer and the execution will continue assuming that ‘\(p\)’ points only to ‘\(a\)’. Verifier targeted analyses are not allowed to assume the absence of errors; in this sense, the notion of safety required by compiler targeted analyses is weaker. However, practical considerations softens the requirements on verifier’s analyses. If compilers are required to expose a well-defined behaviour on all conforming programs, verification tools often assume stricter rules than those dictated by the standard of the programming language with the result of restricting the class of analyzable programs to a set of well-behaved ones. For instance, assuming the absence of some kind of casts [Act06], it is possible to simplify the analysis and also improve its precision. For those programs that do not belong to this restricted set, the analysis produce some false positives\footnote{A false positive is an error reported by the analyzer which however cannot occur in any of the possible execution paths.} and the process of \(\theta\)-soundness will erroneously
remove from the abstraction some of the possible executions yielding to a non-safe result. As noted in [Hin01] this can be acceptable in many areas:

I was told the users actually liked the false-positives in my analysis because they claimed when my analysis got confused it was a good indication that the code was poorly written and likely to have other problems. This came as a complete surprise. While additional study is needed to claim these observations to be valid in a broader sense, they lead me to conclude that the notion of safety should be reconsidered for many applications of static analysis.

Measuring the Alias Analyses

It is a quite accepted fact that in the alias analysis field, the independent verification of the published results is a considerably difficult task. The first consequence of this is the absence of a clear and complete comparison between the existing methods. The difficulty of reproducing the publicly available results can be explained by the intrinsic difficulty of defining a valuable metric for the problem as a great number of parameters must be taken into account: as the chosen intermediate representation, the benchmark suite used for the testing phase and, more generally, all the details of the infrastructure where the analysis is put to work. For instance, some analyses [EGH94, HP00] work on an intermediate representation of the code that results from a simplification phase, which reduces all expressions to a normal form with the goal of limiting the complexity of the implementation as less cases need to be considered; however, it also introduces temporary variables and intermediate assignments to emulate step by step the evaluation of the original expressions. Since many of the used metrics depend on the number of variables, this transformation makes harder, if not impossible at all, any comparison between these methods with other methods that do not perform the simplification. Moreover, alias information is not useful on its own, but it is needed by other client analyses. Thus, the definition of what is a good trade-off between the cost of the analysis and the precision of the computed solution inevitably depends on the client applications; it is indeed a common opinion among the researchers that each area of application requires an ad-hoc method or an adaptation of one described in the literature. The result is that a single metric that gives an absolute measure of the value of a method does not exist. However, to help implementors of aliasing analyses to determine which pointer analysis is appropriate for their application and to help researchers to identify which algorithms should be used as basis for future advances, some partial metrics have been proposed [HP01]; the idea is that since all these metrics have their strengths and weaknesses, a combination should be used. A first popular metric records for each pointer variable the number of pointed objects; the idea is that a lower number of referenced objects would mean a more precise alias information. Although this metric is quite simple to measure, it presents some flaws.

- Due to local variables in recursive functions and the possibility of dynamically allocating memory (heap-allocated objects), an alias analysis should be able to model an unbounded number of objects. To have a finite representation of the
set of the possible memory objects, each method defines a finitely representable approximation. For example in [EGH94] the whole heap is modeled as a single object; in this case the metric will count only one for all the referenced heap-allocated objects with the effect of incorrectly suggesting a precise analysis.

- As anticipated, alias information is used by other client analyses, then its real effectiveness can only be measured on the results of whole process. But there are no straightforward relations between the results of this metric and the precision of the client analyses; For example, the removal of a single alias pair would allow for the client analysis to prove the absence of a run-time error otherwise not provable.

The above metric is usually named direct as it refers to a quantity that is a direct result of the analysis. To address the flaws just highlighted, some indirect metrics have been developed.

1. A first kind of indirect metric measures the relative improvement to the precision of the aliasing information with respect to the worst-case assumption. This kind of metric is reported to be particularly useful on strongly-typed languages where the worst-case assumptions are not as bad as in other weakly-typed languages like C [Hin01].

2. A second kind of indirect metric requires to implement a client of the alias information and then it measures the variation of the precision of the results of the client analysis at the varying of the precision of the supplied aliasing information. The main weakness of this metric is that its results cannot be generalized to other client analyses.

Comparisons are difficult also for what concerns performances. The careful engineering of a points to analysis, particularly for flow-sensitive analyses\footnote{This is probably due to the greater complexity of flow-sensitive analyses with respect to a flow-insensitive one. In a more complex method there are more opportunities to improve.}, can dramatically improve its performance [Hin01]. The worst-case complexities often do not reflect the mean cost of the algorithm, which is greatly influenced by heuristics developed over the default algorithm, which however require a great effort of fine tuning for the specific target application. However, as criticized in [Hin01], even today most published papers about new analysis methods seldom present a complete quantitative evaluation using these guidelines; also, for those works that provide experimental data, too often the independent verification is missing and the acceptance of the proposed results becomes a matter of faith.

Notes on the Analysis of the Java Language

The Java language has emerged as a popular alternative to other mainstream languages in many areas. Java presents a clean and simple memory model where conceptually all objects are allocated in a garbage-collected heap. While useful to the programmer, this model comes with a cost. In many cases it would be more efficient to allocate objects on the stack, eliminating the dynamic memory management overhead for that...
object. Aliasing analysis allows to detect those cases in which it is possible to perform this simplification. Another characteristic of the Java language is the availability of synchronized methods that ensure that the body of the function is executed atomically by acquiring and releasing a lock in the receiver object. But the lock overhead is wasted when only one thread can access the object; the lock is required only when there is multiple threads may attempt to access the same object simultaneously. Also in this case, alias analysis allows to detect which threads can access an object and thus possibly allowing the removal of the code for the locking. Studies have shown [WR99] that it is possible to eliminate a significant number of heap allocations (in the tests between 22% and 95%) and synchronization operations (in the tests between 24% and 64%). For what concerns the realization of alias analyses, the Java language —while adding new features like virtual functions and exception handling— may still be much easier to analyze than the C language [WL95] because of its strong type system without type casts and pointer arithmetics, the type information given by the static type system of the language can be used to deduce affordable alias information. Another feature of Java simplifies the analysis algorithm: it does not support pointers into the middle of an object — an object reference in Java can point only to the beginning of an object. This means that two pointers may either point to exactly the same location or not; they cannot point to different offsets within one allocated block of memory as it is possible in the C language.

1.5 Thesis Purpose

The presented method is targeted for application in the context of software verification. Compiler-targeted applications require relatively imprecise alias information, thus they can rely on fast algorithms for its computation. However, as empirical studies have evidenced [HP01, Hin01], for software verification there is a lower bound of precision below which the points-to information is pretty useless. For these reasons, our aim is to develop a points-to analysis that, though less efficient than other methods based on the same representation, computes a more precise approximation of stack-allocated objects and that is also suitable for integration with the precise inter-procedural techniques already present in the literature [Ema93, WL95].

1.6 Thesis Contribution

The present dissertation describes a store-based, flow-sensitive and intra-procedural points-to analysis working on a relatively high-level intermediate representation of the source code, which also makes no assumptions about the inter-procedural analysis model. In particular, beyond the assignment operation — which is the most essential operation of a points-to analysis and thus it is omnipresent in all the papers on the topic — we describe a filter operation that enables the analysis to increase the precision of the computed solution by exploiting the expressions used in branching statements. Moreover, a formal proof of the soundness of the presented operations is developed.

14The same consideration holds for all other strongly typed languages.
2 The Analysis Method

This chapter is meant to be as much self-contained as possible. The aim of this sections is to present few simple but formal definitions of a simplified but general memory model and, on these, build the algorithms and prove their correctness.

2.1 The Domain

Let \( \mathcal{L} \) be a given set that we call the locations set and whose elements are called locations.

**Definition 5. (Abstract and concrete domains.)** We call support set of the concrete domain the set \( \mathcal{C} \) of the total functions from \( \mathcal{L} \) to \( \mathcal{L} \); we call support set of the abstract domain the set \( \mathcal{A} \) of the binary relations on the set \( \mathcal{L} \)

\[
\mathcal{C} \overset{\text{def}}{=} \mathcal{L} \rightarrow \mathcal{L}; \\
\mathcal{A} \overset{\text{def}}{=} \wp(\mathcal{L} \times \mathcal{L}).
\]

We define the concrete domain as the complete lattice generated by the powerset of \( \mathcal{C} \)

\[
\langle \wp(\mathcal{C}), \subseteq, \cup, \cap, \emptyset, \mathcal{C} \rangle.
\]

We define the abstract domain as the complete lattice

\[
\langle \mathcal{A}, \subseteq, \cup, \cap, \emptyset, \mathcal{L} \times \mathcal{L} \rangle.
\]

Note that from the above definition we have that \( \mathcal{C} \subseteq \mathcal{A} \). Though we use the same notation for the operations of the two lattices they obviously have different definitions. For the abstract domain the partial order ‘\( \subseteq \)’, the operations ‘\( \cup \)’ and ‘\( \cap \)’ are referred to sets of pairs of locations; whereas for the concrete domain they are referred to sets of functions \( \mathcal{L} \rightarrow \mathcal{L} \). The semantics of the abstract domain is defined using the fact that \( \mathcal{C} \subseteq \mathcal{A} \) and the partial order ‘\( \subseteq \)’ on sets of pairs of locations.

**Definition 6. (Concretization function.)** Let

\[
\gamma : \mathcal{A} \rightarrow \wp(\mathcal{C})
\]

be defined, for all \( A \in \mathcal{A} \), as

\[
\gamma(A) \overset{\text{def}}{=} \{ C \in \mathcal{C} \mid C \subseteq A \}.
\]

Now we present some definitions useful to define how we navigate the pointsto graph.
Definition 7. (The prev and post functions.) Let
\[ \text{PREV, POST} : \mathcal{A} \times \mathcal{L} \rightarrow \wp(\mathcal{L}) \]
be defined, for all \( A \in \mathcal{A} \) and \( l \in \mathcal{L} \), as
\[
\begin{align*}
\text{PREV}(A, l) & \triangleq \{ m \in \mathcal{L} \mid (m, l) \in A \}; \\
\text{POST}(A, l) & \triangleq \{ m \in \mathcal{L} \mid (l, m) \in A \}.
\end{align*}
\]

For convenience we generalize the definition of the POST and PREV functions to sets of locations.

Definition 8. (Extended prev and post functions.) Let
\[ \text{PREV, POST} : \mathcal{A} \times \wp(\mathcal{L}) \rightarrow \wp(\mathcal{L}) \]
be defined, for all \( A \in \mathcal{A} \) and \( L \subseteq \mathcal{L} \), as
\[
\begin{align*}
\text{PREV}(A, L) & \triangleq \bigcup \{ \text{PREV}(A, l) \mid l \in A \}; \\
\text{POST}(A, L) & \triangleq \bigcup \{ \text{POST}(A, l) \mid l \in A \}.
\end{align*}
\]

2.2 The Language

In this section we present a simple language to model the points-to problem.

Definition 9. (Expressions.) We define the set \( \text{Expr} \) as the language generated by the grammar
\[
e ::= l | * e
\]
where \( l \in \mathcal{L} \) and \( * \notin \mathcal{L} \) is a terminal symbol.

Definition 10. (Evaluation of expressions.) Let
\[ \text{EVAL} : \mathcal{A} \times \text{Expr} \rightarrow \wp(\mathcal{L}) \]
be defined inductively on \( \text{Expr} \) (Definition 9). Let \( A \in \mathcal{A} \), \( l \in \mathcal{L} \) and \( e \in \text{Expr} \); then we define
\[
\begin{align*}
\text{EVAL}(A, l) & \triangleq \{ l \}; \\
\text{EVAL}(A, * e) & \triangleq \text{POST}(A, \text{EVAL}(A, e)).
\end{align*}
\]

Not necessary for the goal of this chapter, for completeness we report the concretization of the points-to abstract domain in terms of abstract alias queries.
Definition 11. (Induced alias relation.) We define
\[ \gamma : A \rightarrow \text{AliasQ}^\sharp \]
as follows. Let \( A \in A \), then let \( \gamma(A) \overset{\text{def}}{=} \text{ALIAS}^\sharp \), where, forall \( e, f \in \text{Expr} \), we have
\[ E \overset{\text{def}}{=} \text{EVAL}(A, e); \]
\[ F \overset{\text{def}}{=} \text{EVAL}(A, f); \]
\[ \text{ALIAS}^\sharp(e, f) \overset{\text{def}}{=} \begin{cases} 
0, & \text{if } E \cap F = \emptyset; \\
1, & \text{if } E = F \land \# E = 1; \\
\top, & \text{otherwise.}
\end{cases} \]

Definition 12. (Conditions.) We define the set of conditions as the set
\[ \text{Cond} \overset{\text{def}}{=} \{\text{eq, neq}\} \times \text{Expr} \times \text{Expr} \]

Definition 13. (Value of conditions.) Let
\[ \text{TrueCond} \subseteq C \times \text{Cond} \]
be a set defined, for all \( C \in C \) and \( e, f \in \text{Expr} \), as
\[ (C, (\text{eq}, e, f)) \in \text{TrueCond} \overset{\text{def}}{=} \text{EVAL}(C, e) = \text{EVAL}(C, f); \]
\[ (C, (\text{neq}, e, f)) \in \text{TrueCond} \overset{\text{def}}{=} (C, (\text{eq}, e, f)) \notin \text{TrueCond}. \]

Let \( C \in C \) and let \( c \in \text{Cond} \), for convenience of notation we write \( C \models c \) when \( (C, c) \in \text{TrueCond} \). We also introduce the function
\[ \text{MODELSET} : \text{Cond} \rightarrow 2^C, \]
defined, for all \( c \in \text{Cond} \), as
\[ \text{MODELSET}(c) \overset{\text{def}}{=} \{ C \in C \mid C \models c \}. \]
In other words, ‘\( \text{MODELSET}(c) \)’ is the set of the concrete memory descriptions where the condition \( c \) is true.

2.2.1 Assignment

Definition 14. (Assignment evaluation.) We define the set of assignments as
\[ \text{Assignments} \overset{\text{def}}{=} \text{Expr} \times \text{Expr} \]

Let
\[ \text{ASSIGN} : A \times \text{Assignments} \rightarrow A \]
be defined as follows. For all $A \in \mathcal{A}$ and $e, f \in \text{Expr}$, let

$$\text{ASSIGN}(A, (e, f)) \overset{\text{def}}{=} \text{EVAL}(A, e) \times \text{EVAL}(A, f) \cup (A \setminus K)$$

where, the \textit{kill set} $K$ is defined as

$$K \overset{\text{def}}{=} \begin{cases} \text{EVAL}(A, e) \times \mathcal{L}, & \text{if } \#\text{EVAL}(A, e) = 1; \\ \emptyset, & \text{otherwise.} \end{cases}$$

The following lemma shows that the \textit{‘assign’} function just described, defines also the concrete semantics of the assign operation, i.e. performing an assignment on an element of the concrete domain yields another element of the concrete domain.

\textbf{Lemma 1. (Restriction of the assignment to the concrete domain.)} The set $\mathcal{C}$ is closed with respect to the function assign, that is, for all $C \in \mathcal{C}$ and $a \in \text{Assignments}$ holds that $\text{ASSIGN}(C, a) \in \mathcal{C}$.

Therefore, the function assign restricted to $\mathcal{C}$ can be written as

$$\text{ASSIGN} : \mathcal{C} \times \text{Assignments} \rightarrow \mathcal{C}.$$
Definition 17. (Extended eval function.) Let
\[
\text{eval} : \mathcal{A} \times \text{Expr} \times \mathbb{N} \to \wp(\mathcal{L})
\]
be inductively defined as follows. Let \( A \in \mathcal{A}, l \in \mathcal{L}, e \in \text{Expr} \) and \( i \in \mathbb{N} \); then we define
\[
\begin{align*}
\text{eval}(A, e, 0) & \overset{\text{def}}{=} \text{eval}(A, e); \\
\text{eval}(A, l, i + 1) & \overset{\text{def}}{=} \emptyset; \\
\text{eval}(A, * e, i + 1) & \overset{\text{def}}{=} \text{eval}(A, e, i).
\end{align*}
\]

Definition 18. (Target function.) We define the function
\[
\text{targ} : \mathcal{A} \times \wp(\mathcal{L}) \times \text{Expr} \times \mathbb{N} \to \wp(\mathcal{L})
\]
inductively as follows. Let \( A \in \mathcal{A}, M \subseteq \mathcal{L}, e \in \text{Expr} \) and \( i \in \mathbb{N} \); then we define
\[
\begin{align*}
\text{targ}(A, M, e, 0) & \overset{\text{def}}{=} \text{eval}(A, e) \cap M; \\
\text{targ}(A, M, e, i + 1) & \overset{\text{def}}{=} \text{eval}(A, e, i + 1) \cap \text{prev}(A, \text{targ}(A, e, i)).
\end{align*}
\]

Definition 19. (Filter 1.) Let
\[
\phi : \mathcal{A} \times \wp(\mathcal{L}) \times \text{Expr} \times \mathbb{N} \to \mathcal{A}
\]
be defined as follows. Let \( A \in \mathcal{A}, M \subseteq \mathcal{L}, e \in \text{Expr} \) and \( i \in \mathbb{N} \). For convenience of notation let \( x = \langle A, M, e \rangle \); then we define
\[
\begin{align*}
\phi(x, 0) & \overset{\text{def}}{=} A; \\
T & \overset{\text{def}}{=} \text{targ}(x, i + 1); \\
\phi(x, i + 1) & \overset{\text{def}}{=} \phi(x, i) \setminus \begin{cases} T \times (\mathcal{L} \setminus \text{targ}(x, i)), & \text{if } \#T = 1; \\
\emptyset, & \text{otherwise.}
\end{cases}
\end{align*}
\]

Definition 20. (Filter 2.) Let
\[
\phi : \mathcal{A} \times \wp(\mathcal{L}) \times \text{Expr} \to \mathcal{A}
\]
be defined, for all \( A \in \mathcal{A}, M \subseteq \mathcal{L} \) and \( l \in \mathcal{L} \), as
\[
\begin{align*}
\phi(A, M, l) & \overset{\text{def}}{=} \begin{cases} A, & \text{if } l \in M; \\
\bot, & \text{otherwise;}
\end{cases} \\
\phi(A, M, * e) & \overset{\text{def}}{=} \bigcap_{i \in \mathbb{N}} \phi(A, M, * e, i).
\end{align*}
\]
int **pp, *q, *p, *r, a, b, c;

if (...) pp = &p;
else    pp = &q;
        // EVAL(*pp) = \{p, q\}

if (...) r = &a;
else    r = &c;
        // EVAL(*r) = \{a, c\}

p = &a;
// eval(*p) = \{a\}
q = &b;
// eval(*q) = \{b\}

**pp = r;
// eval(**pp) = eval(*q) = \{a, b, c\}
// eval(*p) = \{a, c\}

Listing 2.1: an example of application of the assignment operation.

Definition 21. (Filter 3.) Let

\( \phi: A \times \text{Cond} \rightarrow A \)

be defined as follows. Let \( e, f \in \text{Expr} \), and let

\[ I \overset{\text{def}}{=} \text{EVAL}(A, e) \cap \text{EVAL}(A, f); \]
\[ E \overset{\text{def}}{=} \text{EVAL}(A, e) \setminus \text{EVAL}(A, f); \]
\[ F \overset{\text{def}}{=} \text{EVAL}(A, f) \setminus \text{EVAL}(A, e). \]

Then, for all \( A \in \mathcal{A} \), we define

\[ \phi(A, (\text{eq}, e, f)) \overset{\text{def}}{=} \phi(A, I, e) \cap \phi(A, I, f), \]
\[ \phi(A, (\text{neq}, e, f)) \overset{\text{def}}{=} \begin{cases} \phi(A, E, e) \cup \phi(A, F, f), & \text{if } \#I = 1; \\ A, & \text{otherwise}. \end{cases} \]

2.3 Examples

This section presents some examples to illustrated how the model just presented works.

Example 3. This example is about the abstract assignment operation. Consider the code in Listing 2.1. Note that the C assignment ‘*pp = r’ in our simplified language is
expressed as the pair \((\ast pp, \ast r)\). Assume to reach line 15 with the approximated points-to information \(A \in \mathcal{A}\)
\[
\begin{align*}
\text{EVAL}(A, \ast pp) &= \{p, q\}, \\
\text{EVAL}(A, \ast p) &= \{a\}, \\
\text{EVAL}(A, \ast q) &= \{b\}, \\
\text{EVAL}(A, \ast r) &= \{a, c\};
\end{align*}
\]
then
\[
\text{EVAL}(A, \ast pp) \times \text{EVAL}(A, \ast r) = \{(p, a), (p, c), (q, a), (q, c)\}.
\]
The result of the evaluation of the rhs of the assignment, \(\ast pp\), contains more than one locations, \(p\) and \(q\); then from the definition of the assignment operation (Definition 14) we have that the kill set \(K\) is empty, then the result of the assignment can be expressed as
\[
\text{ASSIGN}(A, (\ast pp, \ast r)) = A \cup \text{EVAL}(A, \ast pp) \times \text{EVAL}(A, \ast r) \\
= A \cup \{(p, a), (p, c), (q, a), (q, b)\}.
\]
Note that after the execution of the assignment (Figure 2.1), the old values of the variables ‘\(p\)’ and ‘\(q\)’ are not overwritten, i.e.,
\[
\{(p, a), (q, b)\} \subseteq \text{ASSIGN}(A, (\ast pp, \ast r)).
\]

**Example 4.** This is another example of the application of the abstract assignment operation. Consider the code in Listing 2.2. Again, the C assignment ‘\(*pp = r\)’ in our simplified language is expressed as the pair \((\ast pp, \ast r)\). Assume to reach line 9 with the approximated points-to information \(A \in \mathcal{A}\) such that
\[
\begin{align*}
\text{EVAL}(A, \ast pp) &= \{p\}, \\
\text{EVAL}(A, \ast p) &= \{c\}, \\
\text{EVAL}(A, \ast r) &= \{a, b\};
\end{align*}
\]
Listing 2.2: another example of application of the assignment operation.

Figure 2.2: a representation of points-to information before and after the execution of the
assignment operation at line 10 of Listing 2.2
int *p, *q, a, b, c;

if (...) p = &a;
else    p = &b;
// EVAL(*p) = \{a, b\}

if (...) q = &b;
else    q = &c;
// EVAL(*q) = \{b, c\}

if (p == q) {
    // EVAL(*p) = EVAL(*q) = \{b\}
}

Listing 2.3: an example of application of the filter operation.

then

\text{EVAL}(A, *pp) \times \text{EVAL}(A, *r) = \{(p, a), (p, b)\}.

But this time the evaluation of the rhs of the assignment, *pp, contains only one location, p. From (Definition 14) we have

\[ K = \text{EVAL}(A, *pp) \times \mathcal{L} = \{(p, c)\}, \]

and then (Figure 2.1)

\[ \text{ASSIGN}(A, (*pp, *r)) = (A \setminus K) \cup \text{EVAL}(A, *pp) \times \text{EVAL}(A, *r) \]

\[ = \left( A \setminus \{(p, c)\} \right) \cup \{(p, a), (p, b)\}. \]

Note that, in this case, the assignment deletes the old value of the variable ‘p’, i.e.,

\[ (p, c) \not\in \text{ASSIGN}(A, (*pp, *r)). \]

\textbf{Example 5.} Consider the example program in Listing 2.3. As anticipated in the annotations of the presented code, the filter operation, acting on the condition ‘p == q’, is able to detect that inside the body of the \textbf{if} statement at line 12 both ‘p’ and ‘q’ definitely point to ‘b’. Now we want to show step by step how this result is obtained from the given definitions. Since the situation for ‘p’ and ‘q’ is symmetrical, we show only how it can be derived that ‘p’ definitely points to ‘b’. Recall that the boolean expression of the C language ‘q == p’ corresponds, in our simplified language, to the triple (eq, *p, *q). Assume now that line 10 is reached with the following approximated points-to information

\[ A \in \mathcal{A} \ (\text{Figure 2.3}) \]

\[ \text{EVAL}(A, *p) = \{a, b\}, \]

\[ \text{EVAL}(A, *q) = \{b, c\}. \]
Figure 2.3: A representation of the points-to information before and after the execution of the filter operation on the condition of the if statement at line 11 of Listing 2.3.

Figure 2.4: A representation of computation of the filter operation for the example in Listing 2.3.

2.3 Examples
From the definition of the abstract filter operation (Definition 21) we have

\[ I = \text{eval}(A, \ast p) \cap \text{eval}(A, \ast q), \]
\[ \phi(A, (eq, \ast p, \ast q)) = \phi(A, I, \ast p) \cap \phi(A, I, \ast q). \]

The evaluation of the expressions is illustrated by the following table.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \text{eval}(A, \ast p, i) )</th>
<th>( \text{eval}(A, \ast q, i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{p}</td>
<td>{q}</td>
</tr>
<tr>
<td>0</td>
<td>{a,b}</td>
<td>{b,c}</td>
</tr>
</tbody>
</table>

For \( i = 0 \), the target set of the filter (Definition 20) is then defined as

\[ I = \text{eval}(A, \ast p) \cap \text{eval}(A, \ast q) \]
\[ = \text{eval}(A, \ast p, 0) \cap \text{eval}(A, \ast q, 0) \]
\[ = \{a, b\} \cap \{b, c\} = \{b\}. \]

Then, recalling from Definition 19 that

\[ \text{Targ}(A, I, e, i + 1) = \text{eval}(A, e, i + 1) \cap \text{prev}(A, \text{Targ}(A, I, e, i)), \]

we compute backward the sequence of target sets for the expression \( \ast p \) as

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \text{Targ}(A, {b}, \ast p, i) )</th>
<th>Removed arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{p}</td>
<td>{(p, a)}</td>
</tr>
<tr>
<td>0</td>
<td>{b}</td>
<td>{}</td>
</tr>
</tbody>
</table>

Since the target set for \( i = 1 \) consists of the only element \( p \) and the node \( a \) is not part of the target set for \( i = 0 \), then the filter removes the arc \((p, a)\) from the points-to information. See Figure 2.4 for a graphical representation of the described situation.

**Example 6.** Now we present a similar situation to show that when the abstract filter operation cuts some arcs (Definition 19) what matters is the cardinality of the set of the “pointers” and not cardinality of the set of the “pointed” objects. Consider the code in Listing 2.4 the points-to information \( A \in \mathcal{A} \) at line 14, is presented in Figure 2.5. In this case the evaluation of the two expressions \( \ast p \) and \( \ast q \) proceeds as follows

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \text{eval}(A, \ast p, i) )</th>
<th>( \text{eval}(A, \ast q, i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{p}</td>
<td>{q}</td>
</tr>
<tr>
<td>0</td>
<td>{a,b,c,d}</td>
<td>{b,c}</td>
</tr>
</tbody>
</table>

For \( i = 0 \), we have the target set

\[ \text{eval}(A, \ast p) \cap \text{eval}(A, \ast q) = \text{eval}(A, \ast p, 0) \cap \text{eval}(A, \ast q, 0) \]
\[ = \{a, b, c, d\} \cap \{b, c\} = \{b, c\}. \]

The computation of the filter on the expression ‘p’ proceeds as follows (Figure 2.6)
int *p, *q, a, b, c, d;

if (...) p = &a;
else p = &b;
if (...) p = &c;
else p = &d;
// EVAL(*p) = \{a, b, c, d\}

if (...) q = &b;
else q = &c;
// EVAL(*q) = \{b, c\}

if (p == q) {
// EVAL(*p) = EVAL(*q) = \{b, c\}
}

Listing 2.4: an example of application of the filter operation.

Figure 2.5: a representation of the points-to information before and after the execution of the filter operation on the condition of the if statement at line 15 of Listing 2.4.

2.3 Examples
2.3 Examples

Figure 2.6: a representation of the computation of the filter operation for the example in Listing 2.4.

Figure 2.7: a representation of the points-to information $A \in \mathcal{A}$ before and after the execution of the filter operation on the condition $(\text{eq, } ** pp, a)$. 
Figure 2.8: a representation of computation of the filter operation for the example in Figure 2.7.

### Example 7
Consider the points-to approximation \( A \in \mathcal{A} \) described in Figure 2.7. In this case there are two levels of indirection. Assume to filter the points-to approximation \( A \) with respect to the condition \((eq, **pp,a)\). The evaluation of the lhs and the rhs of the condition proceeds as follows

\[
\begin{array}{lll}
i & viewed(A, \{b, c\}, p, i) & Removed arcs \\
1 & \{p\} & \{(p,a), (p,d)\} \\
0 & \{b, c\} & \emptyset \\
\end{array}
\]

Then, for \( i = 0 \), we have the target set

\[
\text{eval}(A, **p) \cap \text{eval}(A, a) = \text{eval}(A, **p, 0) \cap \text{eval}(A, a, 0) \\
= \{a, b, c\} \cap \{a\} = \{a\}.
\]

The computation of the filter on the lhs proceeds as

\[
\begin{array}{lll}
i & viewed(A, \{a\}, **p, i) & Removed arcs \\
2 & \{pp\} & \{(pp,q)\} \\
1 & \{p\} & \{(p,b)\} \\
0 & \{a\} & \emptyset \\
\end{array}
\]

2.3 Examples
Figure 2.9: on the left a representation of the initial points-to information $A \in \mathcal{A}$, in the middle the information resulting by filtering the initial information $A$ on the condition $(\text{eq}, **pp, a)$; finally, on the right, the points-to information resulting from filtering the approximation $\hat{A}$ on the condition $(\text{neq}, **pp, a)$.

Figure 2.10: a representation of computation of the filter operation for the example in Figure 2.9 on the condition $(\text{eq}, **pp, a)$.
Figure 2.11: a representation of computation of the filter operation for the example in Figure [2.9] on the condition \((\text{neq}, **pp, a)\).

Figure [2.8] depicts the computation just described.

**Example 8.** Consider the points-to approximation \(A \in \mathcal{A}\) described in Figure [2.9]. The evaluation of the expression \(**pp\) follows the steps

<table>
<thead>
<tr>
<th>(i)</th>
<th>EVAL((A, **p, i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{pp}</td>
</tr>
<tr>
<td>1</td>
<td>{p, q, r}</td>
</tr>
<tr>
<td>0</td>
<td>{a, b}</td>
</tr>
</tbody>
</table>

Assume to filter the points-to approximation \(A\) on the condition \((\text{eq}, **pp, a)\) and also on the opposite condition \((\text{neq}, **pp, a)\). For \(i = 0\), for the equality and the inequality conditions we have the target sets

\[
\text{EVAL}(A, **pp) \cap \text{EVAL}(A, a) = \{a, b\} \cap \{a\} = \{a\},
\]
\[
\text{EVAL}(A, **pp) \setminus \text{EVAL}(A, a) = \{a, b\} \setminus \{a\} = \{b\},
\]

respectively. The computation of the filter on the lhs proceeds as

<table>
<thead>
<tr>
<th>(i)</th>
<th>TARG((A, {a}, **pp, i))</th>
<th>Removed arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{pp}</td>
<td>{(pp, r)}</td>
</tr>
<tr>
<td>1</td>
<td>{p, q}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>0</td>
<td>{a}</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(i)</th>
<th>TARG((A, {b}, **pp, i))</th>
<th>Removed arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{pp}</td>
<td>{(pp, p), (pp, q)}</td>
</tr>
<tr>
<td>1</td>
<td>{r}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>0</td>
<td>{b}</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

2.3 Examples
Figure 2.12: on the left a representation of the points-to approximation $A \in \mathcal{A}$, on the right a representation of the approximation resulting from the application of the filter on the condition $(eq, \ast\ast\ast ppp, a)$. The arcs $\{(ppp, rr), (p, b)\}$ have been removed.

Figure 2.10 and Figure 2.11 depict the filter computation just described.

**Example 9.** Now consider the points-to approximation $A \in \mathcal{A}$ described in Figure 2.12. The evaluation of the expression $\ast\ast\ast ppp$ follows the steps

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{EVAL}(A, \ast\ast p, i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>${ppp}$</td>
</tr>
<tr>
<td>2</td>
<td>${pp, qq, rr}$</td>
</tr>
<tr>
<td>1</td>
<td>${p, r}$</td>
</tr>
<tr>
<td>0</td>
<td>${a, b, c}$</td>
</tr>
</tbody>
</table>

Assume to filter the points-to approximation $A$ on the condition $(eq, \ast\ast\ast ppp, a)$. For $i = 0$, for the equality condition we have the target set

$$\text{EVAL}(A, \ast\ast\ast ppp) \cap \text{EVAL}(A, a) = \{a, b, c\} \cap \{a\} = \{a\}.$$  

The computation of the filter on the lhs proceeds as follows

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{TARG}(A, {a}, \ast\ast\ast ppp, i)$</th>
<th>Removed arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>${ppp}$</td>
<td>${(ppp, rr)}$</td>
</tr>
<tr>
<td>2</td>
<td>${pp, qq}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>1</td>
<td>${p, qq}$</td>
<td>${(p, b)}$</td>
</tr>
<tr>
<td>0</td>
<td>${a}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Figure 2.13 depicts the filter computation just described.
2.4 Results

2.4.1 Notation

The proofs are organized as sequences of deductions, for convenience of notation presented inside tables. Each table is organized in three columns: the first column contains the tag used to name the step; the second column contains the statement and the third column contains a list of tags that represents the list of statements used to infer the current row. There are three kinds of tags. The first kind of tag, denoted as ‘TS’, is used to mark the thesis, which, if explicitly presented, occurs always in the top row. The second kind of tag is used to describe the hypotheses, marked as ‘H₀’, . . . , ‘Hₙ’. Among the hypotheses we improperly list the lemmas used in the proof. The third kind of tag is used to describe deductions, displayed as ‘D₀’, . . . , ‘Dₙ’, with the exception of the last deduction step, which is tagged with the symbol ‘✠’. Within the table, the hypotheses are displayed below the thesis and deductions below the hypotheses. To stress the separation of the thesis from the hypotheses and of the hypotheses from the deductions horizontal line are used. Deductions, between themselves, are sorted in topological order, such that, if the deduction ‘Dₘ’ requires the deduction ‘Dₙ’, then m > n. When the proof consists of more cases, then multiple tables are used; in this case, an initial table containing the hypotheses common to all cases may be present. Cases are marked as ‘C₁’, . . . , ‘Cₙ’; if an hypothesis comes from considering the case ‘Cₙ’, then the tag ‘Cₙ’ is also reported in the third column of the corresponding row. In inductive proofs, the inductive hypothesis
is marked with a ‘(ind. hyp.)’.

2.4.2 Concrete Assignment

We start by showing that the assignment operation is closed with respect to the set of the concrete memory descriptions \( C \).

**Lemma 2. (Eval cardinality on the concrete domain.)** Let \( C \in C \) and \( e \in \text{Expr} \), then \( \# \text{eval}(C, e) = 1 \).

**Proof.** Let \( C \in C \). We proceed by induction on the definition of the set \( \text{Expr} \) (Definition 9).

\[
\begin{array}{c|c}
\text{TS} & \# \text{eval}(C, e) = 1 \\
\hline
\text{H0} & \text{Definition 10, the eval function.}
\end{array}
\]

For the base case let \( e = l \in L \).

\[
\begin{array}{c|c}
\text{TS} & \# \text{eval}(C, l) = 1 \\
\hline
\text{D0} & \text{eval}(C, l) = \{l\} \quad (\text{H0}) \\
\end{array}
\]

For the inductive case let \( e \in \text{Expr} \).

\[
\begin{array}{c|c}
\text{TS} & \# \text{eval}(C, *e) = 1 \\
\hline
\text{H1} & \# \text{eval}(C, e) = 1 \quad (\text{ind. hyp.}) \\
\text{H2} & \text{Definition 5, the concrete domain.} \\
\text{H3} & \text{Definition 7, the post function.} \\
\hline
\text{D0} & \text{eval}(C, *e) = \text{post}(C, \text{eval}(C, e)) \quad (\text{H0}) \\
\text{D1} & \forall l \in C : \#\{(l, m) \in C\} = 1 \quad (\text{H2}) \\
\text{D2} & \#\{(l, m) \in C \mid l \in \text{eval}(C, e)\} = 1 \quad (\text{H1, D1}) \\
\text{D3} & \# \text{post}(C, \text{eval}(C, e)) = 1 \quad (\text{D2, H3}) \\
\xmark & \# \text{eval}(C, *e) = 1 \quad (\text{D3, D0})
\end{array}
\]

**Lemma 3. (Assignment on the concrete domain.)** Let \( C \in C \) and \( e, f \in \text{Expr} \). For convenience of notation, let \( a \in \text{Assignments} \) such that \( a = (e, f) \). Let

\[
\begin{align*}
\text{eval}(C, e) &= \{l\}; \\
\text{post}(C, l) &= \{n\}; \\
\text{eval}(C, f) &= \{m\};
\end{align*}
\]

then

\[
\text{assign}(C, a) = \left( C \setminus \{l, n\} \right) \cup \{(l, m)\}.
\]
Proof. Let $C \in \mathcal{C}$. First note that from the definition of the concrete domain (Definition 5) and Lemma 2, $\# \text{eval}(C, e) = \# \text{eval}(C, f) = 1$. From the definition of the post function (Definition 7) also $\# \text{post}(C, l) = 1$. Thus the above statement is well formed.

TS
\[
\text{assign}(C, a) = \left( A \setminus \{(l, n)\} \right) \cup \{(l, m)\}
\]

H0 $\{l\} = \text{eval}(C, e)$
H1 $\{n\} = \text{post}(C, l)$
H2 $\{m\} = \text{eval}(C, f)$
H3 Definition 14, the assignment evaluation.

D0 $\# \text{eval}(C, e) = 1$ (H0)
D1 $\text{assign}(C, a) = \text{eval}(C, e) \times \text{eval}(C, f) \cup \left( C \setminus \text{eval}(C, e) \times \mathcal{L} \right)$ (D0, H3)
D2 $\text{eval}(C, e) \times \text{eval}(C, f) = \{(l, m)\}$ (H0, H2)
D3 $C \cap \text{eval}(C, e) \times \mathcal{L} = \{(l, n)\}$ (H1)
D4 $C \setminus \text{eval}(C, e) \times \mathcal{L} = C \setminus \{(l, n)\}$ (D3)
✠ $\text{assign}(C, a) = \left( C \setminus \{(l, n)\} \right) \cup \{(l, m)\}$ (D4, D2, D1)

Proof. (Restriction of the assignment to the concrete domain, Lemma 1.) This result is a simple corollary of Lemma 3.

2.4.3 Observations on the Domain

First we present the following simple result about the monotonicity of the concretization function.

Lemma 4. (Monotonicity of the concretization function.) Let $A, B \in \mathcal{A}$, then

$A \subseteq B \implies \gamma(A) \subseteq \gamma(B)$.

Proof. Let $A, B \in \mathcal{A}$. If $\gamma(A) = \emptyset$ then the thesis is trivially verified. Otherwise let $C \in \gamma(A)$, we have to show that $C \in \gamma(B)$ too.

TH $C \in \gamma(B)$
H0 $C \in \gamma(A)$
H1 Definition 6, the concretization function.
H2 $A \subseteq B$

D0 $C \subseteq A$ (H0, H1)
D1 $C \in \mathcal{C}$ (H0, H1)
D2 $C \subseteq B$ (D0, H2)
✠ $C \in \gamma(B)$ (D2, D1, H1)

From the definition of the concrete and of the abstract domain (Definition 5) and the definition of the concretization function (Definition 6) we complete the description of the abstraction by presenting the abstraction function.

2.4 Results
Definition 22. (Abstraction function.) Let
\[ \alpha : \wp(C) \to \mathcal{A} \]
be defined, for all \( C \subseteq C \), as
\[ \alpha(C) \overset{\text{def}}{=} \bigcup_{D \in C} D. \]

It is possible to show that \( \langle \wp(C), \alpha, \mathcal{A}, \gamma \rangle \) is a Galois connection, that is, for all \( C \in \mathcal{C} \) and \( A \in \mathcal{A} \), holds that:
\[ \alpha(C) \subseteq A \iff C \subseteq \gamma(A). \]
Indeed, given \( C \subseteq C \) and \( A \in \mathcal{A} \) the following steps are all equivalent
\[ \alpha(C) \subseteq A, \]
\[ \bigcup_{D \in C} D \subseteq A, \]
\[ \forall D \in C : D \subseteq A, \]
\[ \forall D \in C : D \in \gamma(A), \]
\[ C \subseteq \gamma(A). \]

On the presented abstraction holds also the following result. The following lemma shows that given a non-bottom abstraction \( a \in \mathcal{A} \), then for each arc \((l,m) \in A\) there is a concrete memory \( C \) abstracted by \( A \) that contains the arc \((l,m)\).

Lemma 5. (Concrete coverage.) Let \( A \in \mathcal{A} \), then
\[ \gamma(A) \neq \emptyset \implies \forall (l, m) \in A : \exists C \in \gamma(A) \cdot (l, m) \in C. \]

Proof. Let \( A \in \mathcal{A} \) such that \( \gamma(A) \neq \emptyset \) and let \((l, m) \in A\). Let \( C \in \gamma(A) \), let \( n \in \mathcal{L} \).

<table>
<thead>
<tr>
<th>TS</th>
<th>\exists D \in \gamma(A) \cdot (l, m) \in D</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0</td>
<td>((l, m) \in A)</td>
</tr>
<tr>
<td>H1</td>
<td>(C \in \gamma(A))</td>
</tr>
<tr>
<td>H2</td>
<td>{n} \in \text{post}(C, l)</td>
</tr>
<tr>
<td>H3</td>
<td>Definition 7, concretization function.</td>
</tr>
<tr>
<td>H4</td>
<td>Definition 7, the post function.</td>
</tr>
<tr>
<td>D0</td>
<td>(C \subseteq A)</td>
</tr>
<tr>
<td>D1</td>
<td>(C \setminus {(l, n)} \subseteq A)</td>
</tr>
<tr>
<td>D2</td>
<td>((l, n) \in C)</td>
</tr>
<tr>
<td>D3</td>
<td>((C \setminus {(l, n)}) \cup {(l, m)} \in \mathcal{C})</td>
</tr>
<tr>
<td>D4</td>
<td>((C \setminus {(l, n)}) \cup {(l, m)} \subseteq A)</td>
</tr>
<tr>
<td>D5</td>
<td>((C \setminus {(l, n)}) \cup {(l, m)} \in \gamma(A))</td>
</tr>
<tr>
<td>*</td>
<td>(\exists D \in \gamma(A) \cdot (l, m) \in C)</td>
</tr>
</tbody>
</table>

2.4 Results
Lemma 6. (Abstraction effect.) \( \text{Let } A \in \mathcal{A}, \text{ then} \)
\[
\alpha(\gamma(A)) \subseteq A;
\]
moreover
\[
\gamma(A) \neq \emptyset \implies \alpha(\gamma(A)) = A.
\]

Proof. Let \( A \in \mathcal{A}. \) Consider that
\[
\begin{align*}
\text{H0} & \quad \text{Definition \ref{def:abstraction-function}} \text{ the abstraction function.} \\
\text{D0} & \quad \alpha(\gamma(A)) = \bigcup_{C \in \gamma(A)} C \\
\end{align*}
\]
We proceed by showing the two inclusions separately. For the first inclusion let \((l, m) \in \alpha(\gamma(A)); \) then we have
\[
\begin{align*}
\text{TS} & \quad (l, m) \in A \\
\text{H1} & \quad (l, m) \in \alpha(\gamma(A)) \\
\text{H2} & \quad \text{Definition \ref{def:concretization-function} the concretization function.} \\
\text{D1} & \quad \exists C \in \gamma(A) : (l, m) \in C \\
\text{D2} & \quad \forall C \in \gamma(A) : C \subseteq A \\
\text{X} & \quad (l, m) \in A \\
\end{align*}
\]
For the second inclusion assume that \( \gamma(A) \neq \emptyset \) and let \((l, m) \in A; \) then we have
\[
\begin{align*}
\text{TS} & \quad (l, m) \in \alpha(\gamma(A)) \\
\text{H1} & \quad \gamma(A) \neq \emptyset \\
\text{H2} & \quad (l, m) \in A \\
\text{H3} & \quad \text{Lemma \ref{lemma:concrete-coverage} concrete coverage.} \\
\text{D1} & \quad \exists C \in \gamma(A) : (l, m) \in C \\
\text{X} & \quad (l, m) \in \alpha(\gamma(A)) \\
\end{align*}
\]

\[\square\]

2.4.4 Results of Correctness

We formalize the requirement of correctness of the abstract operations presented —the expression evaluation, the assignment and the filter operations— with the following theorems.

Theorem 7. (Correctness of expression evaluation.) \( \text{Let } A \in \mathcal{A} \text{ and } e \in \text{Expr;} \) then
\[
\bigcup_{C \in \gamma(A)} \text{EVAL}(C, e) \subseteq \text{EVAL}(A, e).
\]

\[\square\]

2.4 Results
Theorem 8. (Correctness of the assignment.) Let $A \in \mathcal{A}$ and $a \in \text{Assignments}$; then

$$\text{ASSIGN}(\gamma(C), a) \subseteq \gamma(\text{ASSIGN}(A, a)).$$

Theorem 9. (Correctness of the filter.) Let $A \in \mathcal{A}$ and $c \in \text{Cond};$ then

$$\phi(\gamma(A), c) \subseteq \gamma(\phi(A, c)).$$

2.4.5 Proofs

We present some technical lemmas that will lead to the proof of the correctness theorems.

Lemma 10. (Monotonicity of post.) Let $A, B \in \mathcal{A}$ and $l \in \mathcal{L};$ then

$$A \subseteq B \implies \text{POST}(A, l) \subseteq \text{POST}(B, l)$$

Proof. Let $A, B \in \mathcal{A}$ such that $A \subseteq B.$ Let $l, m \in \mathcal{L}.$

<table>
<thead>
<tr>
<th>TS</th>
<th>$m \in \text{POST}(B, l)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0</td>
<td>$m \in \text{POST}(A, l)$</td>
</tr>
<tr>
<td>H1</td>
<td>$A \subseteq B$</td>
</tr>
<tr>
<td>H2</td>
<td>Definition 7, the post function.</td>
</tr>
<tr>
<td>D0</td>
<td>$(l, m) \in A$</td>
</tr>
<tr>
<td>D1</td>
<td>$(l, m) \in B$</td>
</tr>
<tr>
<td>✠</td>
<td>$m \in \text{POST}(B, l)$</td>
</tr>
</tbody>
</table>

Lemma 11. (Monotonicity of the extended post function 1.) Let $A, B \in \mathcal{A}$ and $L \subseteq \mathcal{L};$ then

$$A \subseteq B \implies \text{POST}(A, L) \subseteq \text{POST}(B, L).$$

Proof. Let $A, B \in \mathcal{A}$ such that $A \subseteq B$ and let $L \subseteq \mathcal{L}.$ Let $m \in \mathcal{L}.$

<table>
<thead>
<tr>
<th>TS</th>
<th>$m \in \text{POST}(B, L)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0</td>
<td>$m \in \text{POST}(A, L)$</td>
</tr>
<tr>
<td>H1</td>
<td>$A \subseteq B$</td>
</tr>
<tr>
<td>H2</td>
<td>Definition 8, the extended post function.</td>
</tr>
<tr>
<td>H3</td>
<td>Lemma 10, monotonicity of the post function.</td>
</tr>
<tr>
<td>D0</td>
<td>$\exists l \in L. \ m \in \text{POST}(A, l)$</td>
</tr>
<tr>
<td>D1</td>
<td>$\exists l \in L. \ m \in \text{POST}(B, l)$</td>
</tr>
<tr>
<td>✠</td>
<td>$m \in \text{POST}(B, L)$</td>
</tr>
</tbody>
</table>

2.4 Results
Lemma 12. (Monotonicity of the extended post function 2.) Let \( A \in \mathcal{A} \) and \( L, M \subseteq \mathcal{L} \); then

\[ L \subseteq M \implies \text{post}(A, L) \subseteq \text{post}(A, M). \]

Proof. Let \( A \in \mathcal{A} \) an let \( L \subseteq M \subseteq \mathcal{L} \). Let \( m \in \mathcal{L} \).

\[
\begin{array}{ll}
\text{TS} & m \in \text{post}(A, M) \\
\text{H0} & m \in \text{post}(A, L) \\
\text{H1} & L \subseteq M \\
\text{H2} & \text{Definition 8, the extended post function.} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{D0} & \exists l \in L. m \in \text{post}(A, l) \quad \text{(H0, H2)} \\
\text{D1} & \exists l \in M. m \in \text{post}(A, l) \quad \text{(D0, H1)} \\
\text{✠} & m \in \text{post}(A, M) \quad \text{(D1, H2)} \\
\end{array}
\]

Lemma 13. (Monotonicity of the eval function.) Let \( A, B \in \mathcal{A} \) and \( e \in \text{Expr} \); then

\( A \subseteq B \implies \text{eval}(A, e) \subseteq \text{eval}(B, e) \).

Proof. Let \( A, B \in \mathcal{A} \) such that \( A \subseteq B \). We proceed inductively on the definition of the eval function. For the base case let \( l \in \mathcal{L} \).

\[
\begin{array}{ll}
\text{TS} & \text{eval}(A, l) \subseteq \text{eval}(B, l) \\
\text{H0} & \text{Definition 10, the eval function.} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{D0} & \text{eval}(A, l) = \{l\} \quad \text{(H0)} \\
\text{D1} & \text{eval}(B, l) = \{l\} \quad \text{(H0)} \\
\text{✠} & \text{eval}(A, l) \subseteq \text{eval}(B, l) \quad \text{(D1, D0)} \\
\end{array}
\]

For the inductive case let \( e \in \text{Expr} \).

\[
\begin{array}{ll}
\text{TS} & \text{eval}(A, \ast e) \subseteq \text{eval}(B, \ast e) \\
\text{H0} & A \subseteq B \\
\text{H1} & \text{Definition 10, the eval function.} \\
\text{H2} & \text{Lemma 11, monotonicity of the ext. post 1.} \\
\text{H3} & \text{Lemma 12, monotonicity of the ext. post 2.} \\
\text{H4} & \text{eval}(A, e) \subseteq \text{eval}(B, e) \quad \text{(ind. hyp.)} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{D0} & \text{eval}(A, \ast e) = \text{post}(A, \text{eval}(A, e)) \quad \text{(H1)} \\
\text{D1} & \text{eval}(B, \ast e) = \text{post}(B, \text{eval}(B, e)) \quad \text{(H1)} \\
\text{D2} & \text{post}(A, \text{eval}(A, e)) \subseteq \text{post}(A, \text{eval}(B, e)) \quad \text{(H4, H3)} \\
\text{D3} & \text{post}(A, \text{eval}(B, e)) \subseteq \text{post}(B, \text{eval}(B, e)) \quad \text{(H0, H2)} \\
\text{D4} & \text{post}(A, \text{eval}(A, e)) \subseteq \text{post}(B, \text{eval}(B, e)) \quad \text{(D2, D3)} \\
\text{✠} & \text{eval}(A, \ast e) \subseteq \text{eval}(B, \ast e) \quad \text{(D4, D1, D0)} \\
\end{array}
\]

2.4 Results
Lemma 14. (Monotonicity of the extended eval 1.) Let $A, B \in \mathcal{A}$, $e \in \text{Expr}$ and $i \in \mathbb{N}$; then

$$A \subseteq B \implies \text{eval}(A, e, i) \subseteq \text{eval}(B, e, i).$$

Proof. Let $A, B \in \mathcal{A}$ such that $A \subseteq B$. We proceed inductively on the definition of the extended eval function.

For the first base case let $l \in \mathcal{L}$ and let $i \in \mathbb{N}$.

For the second base case let $e \in \text{expressions}.

For the inductive step let $e \in \text{Expr}$ and let $i \in \mathbb{N}$.

Lemma 15. (Eval cardinality on the abstract domain.) Let $A \in \mathcal{A}$ and $e \in \text{Expr}$; then

$$\gamma(A) \neq \emptyset \implies \# \text{eval}(A, e) > 0.$$ 

Proof. Let $A \in \mathcal{A}$ such that $\gamma(A) \neq \emptyset$. Let $e \in \text{Expr}.$

2.4 Results
Lemma 16. (Extended eval cardinality on the abstract domain.) Let $A, B \in \mathcal{A}$, $e \in \text{Expr}$ and $i \in \mathbb{N}$, then

$$(\gamma(A) \neq \emptyset \land A \subseteq B \land \# \text{eval}(B, e, i) > 0) \implies \# \text{eval}(A, e, i) > 0.$$  

Proof. Let $A, B \in \mathcal{A}$, let $e \in \text{Expr}$ and let $i \in \mathbb{N}$.  

We proceed by induction on $i$ and on $e$ (Definition 9). For the base case let $i = 0$.  

Let $i > 0$. For the second base case let $e = l \in \mathcal{L}$.  

For the inductive case let $e = * f$ where $f \in \text{Expr}$.

2.4 Results 57
Proof. (Correctness of the expression evaluation, Theorem 7.) Let \( A \in \mathcal{A} \) and let \( e \in \text{Expr} \). We distinguish two cases. First case: \( \gamma(A) = \emptyset \) then the thesis is trivially verified. For the second case, \( \gamma(A) \neq \emptyset \), let \( C \in \gamma(A) \). Then we have

\[
\begin{array}{c|l}
\text{TS} & \text{EVAL}(C, e) \subseteq \text{EVAL}(A, e) \\
\text{H0} & C \in \gamma(A) \\
\text{H1} & \text{Definition 14, the concretization function.} \\
\text{H2} & \text{Lemma 13, monotonicity of the eval function.} \\
\text{D0} & C \subseteq A \quad \text{(H0, H1)} \\
\# & \text{EVAL}(C, e) \subseteq \text{EVAL}(A, e) \quad \text{(D0, H2)}
\end{array}
\]

\[\Box\]

Lemma 17. (Effects of the assignment.) Let \( A \in \mathcal{A} \), \( l \in \mathcal{L} \) and \( e, f \in \text{Expr} \); for convenience of notation let \( a \in \text{Assignments} \) be such that \( a = (e, f) \) and \( E = \text{EVAL}(A, e) \). Then

\[
\begin{align*}
\text{POST}(\text{ASSIGN}(A, a), l) & \equiv \\
\text{if } l \notin E; & \text{POST}(A, l), \\
\text{if } E = \{l\}; & \text{EVAL}(A, f), \\
\text{if } l \in E \land \# E > 1. & \text{POST}(A, l) \cup \text{EVAL}(A, f),
\end{align*}
\]

Proof. Let \( A \in \mathcal{A} \), let \( (e, f) = a \in \text{Assignments} \) and \( l \in \mathcal{L} \). We proceed case by case. These are our initial hypotheses.

\[
\begin{array}{c}
\text{H0} & \text{Definition 14, assignment definition.} \\
\text{H1} & \text{Definition 7, the post function.}
\end{array}
\]

We consider separately the three cases of the lemma

\[
\begin{align*}
l \notin \text{EVAL}(A, e); & \quad \text{(C1)} \\
\{l\} = \text{EVAL}(A, e); & \quad \text{(C2)} \\
l \in \text{EVAL}(A, e) \land \# \text{EVAL}(A, e) > 1. & \quad \text{(C3)}
\end{align*}
\]

First case.

\[
\begin{align*}
\text{TS} & \text{ POST}(\text{ASSIGN}(A, a), l) = \text{POST}(A, l) \\
\text{H2} & l \notin \text{EVAL}(A, e) \quad \text{(C1)}
\end{align*}
\]

To prove TS we prove the two inclusions.

\[
\begin{align*}
\text{POST}(A, l) & \subseteq \text{POST}(\text{ASSIGN}(A, a), l); \quad \text{(C1.1)} \\
\text{POST}(\text{ASSIGN}(A, a), l) & \subseteq \text{POST}(A, l). \quad \text{(C1.2)}
\end{align*}
\]

Let \( m \in \mathcal{L} \). For the first sub-case we have

2.4 Results 58
For the second sub-case:

\[
\begin{align*}
\text{TS} & \quad m \in \text{POST}(A,l) \\
H3 & \quad m \in \text{POST}(\text{ASSIGN}(A,a),l) \\
D0 & \quad (l,m) \in A \quad (H3, H1) \\
D1 & \quad (l,m) \notin \text{EVAL}(A,e) \times \mathcal{L} \quad (H2) \\
D3 & \quad (l,m) \in \text{ASSIGN}(A,a) \quad (D0, D1, H0) \\
\text{x} & \quad m \in \text{POST}(\text{ASSIGN}(A,a),l) \quad (D3, H1)
\end{align*}
\]

For the second and third cases we have to prove an intermediate result.

\[
l \in \text{EVAL}(A,e) \implies \text{EVAL}(A,f) \subseteq \text{POST}(\text{ASSIGN}(A,a),l).
\]

Let \( m \in \mathcal{L} \).

\[
\begin{align*}
\text{TS} & \quad m \in \text{POST}(\text{ASSIGN}(A,a),l) \\
H2 & \quad l \in \text{EVAL}(A,e) \quad (C2) \\
H3 & \quad m \in \text{EVAL}(A,f) \quad (C2) \\
D0 & \quad \text{EVAL}(A,e) \times \text{EVAL}(A,f) \subseteq \text{ASSIGN}(A,a) \quad (H0) \\
D1 & \quad (l,m) \in \text{EVAL}(A,e) \times \text{EVAL}(A,f) \quad (H2, H3) \\
D2 & \quad (l,m) \in \text{ASSIGN}(A,a) \quad (D1, D0) \\
\text{x} & \quad m \in \text{POST}(\text{ASSIGN}(A,a),l) \quad (D2, H1)
\end{align*}
\]

Note that for both the second and third case we assume that \( l \in \text{EVAL}(A,e) \) thus \# \text{EVAL}(A,e) > 0 \) so we will check only the cases \# \text{EVAL}(A,e) = 1 (2nd case) and \# \text{EVAL}(A,e) > 1 (3rd case). Now the second case.

\[
\begin{align*}
\text{TS} & \quad \text{POST}(\text{ASSIGN}(A,a),l) = \text{EVAL}(A,f) \\
H2 & \quad l \in \text{EVAL}(A,e) \\
H3 & \quad \# \text{EVAL}(A,e) = 1 \\
D0 & \quad \text{ASSIGN}(A,a) = \{l\} \times \text{EVAL}(A,f) \cup \left(A \setminus \{l\} \times \mathcal{L}\right) \quad (H3, H2, H0)
\end{align*}
\]

Also in this case, to prove TS we prove the two inclusions

\[
\begin{align*}
\text{POST}(\text{ASSIGN}(A,a),l) & \subseteq \text{EVAL}(A,f) \quad (C2.1) \\
\text{POST}(\text{ASSIGN}(A,a),l) & \supseteq \text{EVAL}(A,f) \quad (C2.2)
\end{align*}
\]

One inclusion (C2.2) comes by modus ponens by applying the hypothesis H2 to the previous intermediate result. For the other inclusion (C2.1), let \( m \in \mathcal{L} \); then we have

\[
2.4 \text{ Results} \quad 59
\]
Now the third case.

<table>
<thead>
<tr>
<th>TS</th>
<th>$m \in \text{eval}(A, f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>H4</td>
<td>$m \in \text{post}(\text{assign}(A, a), l)$</td>
</tr>
<tr>
<td>D1</td>
<td>$(l, m) \in \text{assign}(A, a)$</td>
</tr>
<tr>
<td>D2</td>
<td>$(l, m) \not\in A \setminus {l} \times \mathcal{L}$</td>
</tr>
<tr>
<td>D3</td>
<td>$(l, m) \in {l} \times \text{eval}(A, f)$</td>
</tr>
<tr>
<td>$\blacklozenge$</td>
<td>$m \in \text{post}(A, a)$</td>
</tr>
</tbody>
</table>

Again, we prove separately the two inclusions.

$$\text{post}(\text{assign}(A, a), l) \subseteq \text{post}(A, l) \cup \text{eval}(A, f);$$  \hspace{1cm} (C3.1)

$$\text{post}(\text{assign}(A, a), l) \supseteq \text{post}(A, l) \cup \text{eval}(A, f).$$  \hspace{1cm} (C3.2)

For the inclusion (C3.2), applying the modus ponens to the hypothesis H2 and to the above intermediate result we have that

$$\text{eval}(A, f) \subseteq \text{post}(\text{assign}(A, a), l).$$

For the other part

<table>
<thead>
<tr>
<th>TS</th>
<th>$m \in \text{post}(\text{assign}(A, a), l) \supseteq \text{post}(A, l)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>H4</td>
<td>Lemma 13, monotonicity of eval.</td>
</tr>
<tr>
<td>D1</td>
<td>$\text{assign}(A, a) \supseteq A$</td>
</tr>
<tr>
<td>$\blacklozenge$</td>
<td>$\text{post}(\text{assign}(A, a), l) \supseteq \text{post}(A, l)$</td>
</tr>
</tbody>
</table>

For the remaining inclusion (C3.1), let $m \in \mathcal{L}$ so that $m \in \text{post}(\text{assign}(A, a), l)$. We need to show that $m \in \text{eval}(A, f) \cup \text{post}(A, l)$ too: to do this we show that $m \not\in \text{eval}(A, f) \implies m \in \text{post}(A, l)$.

<table>
<thead>
<tr>
<th>TS</th>
<th>$m \in \text{post}(A, l)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>H4</td>
<td>$m \not\in \text{eval}(A, f)$</td>
</tr>
<tr>
<td>H5</td>
<td>$m \in \text{post}(\text{assign}(A, a), l)$</td>
</tr>
<tr>
<td>D1</td>
<td>$(l, m) \not\in \text{eval}(A, e) \times \text{eval}(A, f)$</td>
</tr>
<tr>
<td>D2</td>
<td>$(l, m) \in \text{assign}(A, a)$</td>
</tr>
<tr>
<td>D3</td>
<td>$(l, m) \in A$</td>
</tr>
<tr>
<td>$\blacklozenge$</td>
<td>$m \in \text{post}(A, l)$</td>
</tr>
</tbody>
</table>

\[ \square \]
Lemma 18. (Monotonicity of the assignment.) Let $A, B \in \mathcal{A}$ and $a \in \text{Assignments}$; then

$$(A \subseteq B \land \gamma(A) \neq \emptyset) \implies \text{assign}(A, a) \subseteq \text{assign}(B, a).$$

Proof. Let $A, B \in \mathcal{A}$ be such that $A \subseteq B$ and $\gamma(A) \neq \emptyset$. Let $(e, f) = a \in \text{Assignments}$. We have to prove that $\text{assign}(A, a) \subseteq \text{assign}(B, a)$. Then let $l, m \in \mathcal{L}$ be such that $(l, m) \in \text{assign}(A, a)$. To prove this lemma we have to prove that $(l, m) \in \text{assign}(B, a)$ too. Thus we have

<table>
<thead>
<tr>
<th>TS</th>
<th>$(l, m) \in \text{assign}(B, a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0</td>
<td>$A \subseteq B$</td>
</tr>
<tr>
<td>H1</td>
<td>$\gamma(A) \neq \emptyset$</td>
</tr>
<tr>
<td>H2</td>
<td>$(l, m) \in \text{assign}(A, a)$</td>
</tr>
<tr>
<td>H3</td>
<td>Lemma 17, effects of the assignment.</td>
</tr>
<tr>
<td>H4</td>
<td>Lemma 13, monotonicity of eval.</td>
</tr>
<tr>
<td>H5</td>
<td>Definition 7, the post function.</td>
</tr>
<tr>
<td>H6</td>
<td>Lemma 11, monotonicity of the extended post 1.</td>
</tr>
<tr>
<td>D0</td>
<td>$\text{eval}(A, e) \subseteq \text{eval}(B, e)$ (H0, H4)</td>
</tr>
<tr>
<td>D1</td>
<td>$\text{eval}(A, f) \subseteq \text{eval}(B, f)$ (H0, H4)</td>
</tr>
<tr>
<td>D2</td>
<td>$\text{post}(A, l) \subseteq \text{post}(B, l)$ (H0, H6)</td>
</tr>
</tbody>
</table>

We distinguish two cases.

1. $l \in \text{eval}(A, e)$;
2. $l \not\in \text{eval}(A, e)$.

For the first case.

| H7   | $l \in \text{eval}(A, e)$ (C1) |
| D3   | $l \in \text{eval}(B, e)$ (H7, D0) |
| D4   | $m \in \text{post}(\text{assign}(A, a), l)$ (H2, H5) |
| D5   | $\text{eval}(B, f) \subseteq \text{post}(\text{assign}(B, a), l)$ (D3, H3) |

Note that from H7 follows that $\# \text{eval}(A, e) \geq 1$ and now we distinguish the two sub-cases

1. $\# \text{eval}(A, e) = 1$; (C1.1)
2. $\# \text{eval}(A, e) > 1$; (C1.2)

which cover all the possibilities. Now the first sub-case.

| H8   | $\# \text{eval}(A, e) = 1$ (C1.1) |
| D6   | $\text{post}(\text{assign}(A, a), l) = \text{eval}(A, f)$ (H7, H8, H3) |
| D7   | $m \in \text{eval}(A, f)$ (D4, D6) |
| D8   | $m \in \text{eval}(B, f)$ (D7, D1) |
| D9   | $m \in \text{post}(\text{assign}(B, a), l)$ (D8, D5) |
| ★    | $(l, m) \in \text{assign}(B, a)$ (D9, H5) |
For the other sub-case

\[
\begin{align*}
H8 & \quad \# \text{eval}(A, e) > 1 \quad \text{(C1.2)} \\
D6 & \quad \text{post} (\text{assign}(A, a), l) = \text{eval}(A, f) \cup \text{post}(A, l) \quad \text{(H7, H8, H3)} \\
D7 & \quad \# \text{eval}(B, a) > 1 \quad \text{(H8, D0)} \\
D8 & \quad \text{post} (\text{assign}(B, a), l) = \text{eval}(B, f) \cup \text{post}(B, l) \quad \text{(D3, D7, H3)} \\
D9 & \quad \text{post} (\text{assign}(A, a), l) \subseteq \text{post} (\text{assign}(B, a), l) \quad \text{(D6, D8, D0, D2)} \\
D10 & \quad m \in \text{post}(\text{assign}(B, e), l) \quad \text{(D4, D9)} \\
\star & \quad (l, m) \in \text{assign}(B, a) \quad \text{(D10, H5)}
\end{align*}
\]

This completes the first case (C1). Now the second case (C2).

\[
\begin{align*}
H7 & \quad l \notin \text{eval}(A, e) \quad \text{(C2)} \\
D3 & \quad \text{post} (\text{assign}(A, a), l) = \text{post}(A, l) \quad \text{(H7, H3)} \\
D4 & \quad m \in \text{post}(\text{assign}(A, a), l) \quad \text{(H2, H5)} \\
D5 & \quad m \in \text{post}(A, l) \quad \text{(D4, D3)} \\
D6 & \quad m \in \text{post}(B, l) \quad \text{(D5, D2)}
\end{align*}
\]

Also in the second case we distinguish two sub-cases.

\[
\begin{align*}
l \notin \text{eval}(B, e); & \quad \text{(C2.1)} \\
l \in \text{eval}(B, e). & \quad \text{(C2.2)}
\end{align*}
\]

Now the first sub-case.

\[
\begin{align*}
H8 & \quad l \notin \text{eval}(B, e) \quad \text{(C2.1)} \\
D7 & \quad \text{post} (\text{assign}(B, a), l) = \text{post}(B, l) \quad \text{(H8, H3)} \\
D8 & \quad m \in \text{post}(\text{assign}(B, a), l) \quad \text{(D6, D7)} \\
\star & \quad (l, m) \in \text{assign}(B, a) \quad \text{(D8, H5)}
\end{align*}
\]

Now the other second sub-case

\[
\begin{align*}
H8 & \quad l \in \text{eval}(B, e) \quad \text{(C2.2)} \\
H9 & \quad \text{Lemma 15, eval cardinality on the abstract domain.} \\
D7 & \quad \# \text{eval}(A, e) > 0 \quad \text{(H1, H9)} \\
D8 & \quad \# \text{eval}(A, e) \leq \# \text{eval}(B, e) \quad \text{(D0)} \\
D9 & \quad \# \text{eval}(A, e) < \# \text{eval}(B, e) \quad \text{(D8, H7, H8)} \\
D10 & \quad \# \text{eval}(B, e) > 1 \quad \text{(D9, D7)} \\
D11 & \quad B \subseteq \text{assign}(B, a) \quad \text{(D10, H3)} \\
D12 & \quad (l, m) \in B \quad \text{(D6, H5)} \\
\star & \quad (l, m) \in \text{assign}(B, a) \quad \text{(D12, D11)}
\end{align*}
\]

\[\square\]

It is worth stressing that \(\gamma(A) \neq \emptyset\) is a necessary hypothesis of Lemma 18. Consider indeed the following example: \(L = \{l, m, n\}\) and \(A, B \in A\) such that \(A = \{(m, n)\}\) and \(B = \{(l, m), (m, n)\}\). We have obviously that \(A \subseteq B\). Consider what happens to the arc \((m, n)\) when the assignment \(a = (\ast l, l)\) is performed: \(\text{eval}(A, \ast l) = \emptyset\) while

2.4 Results
EVAL(B, l) = {m} resulting in ASSIGN(A, a) = A and ASSIGN(B, a) = {(l, m), (m, l)}. Thus ASSIGN(A, a) ∉ ASSIGN(B, a).

Proof. (Correctness of the assignment, Theorem 8.) Let A ∈ A, let C ∈ γ(A) and let a ∈ Assignments.

\[
\begin{array}{c|c}
D0 & C \subseteq A \\
D1 & C \in \gamma(C) \\
D2 & \gamma(C) \neq \emptyset \\
D3 & ASSIGN(C, a) \subseteq ASSIGN(A, a) \\
D4 & ASSIGN(C, a) \in \gamma(A(A, a)) \\
\end{array}
\]

To proceed in the proof of the correctness of the filter abstract operation, now we reformulate all the previous lemmas on the post function on the prev function.

Definition 23. (Transposed abstract domain.) Let

\[
\text{TRAN} : A \rightarrow A
\]

be defined, for all A ∈ A, as

\[
\text{TRAN}(A) = \{ (m, l) \mid (l, m) \in A \}.
\]

Lemma 19. (Transpose is idempotent.) Let A ∈ A, then

\[
\text{TRAN(TRAN(A))} = A.
\]

Proof. This result can be easily derived from the definition of the transpose function (Definition 23).

Lemma 20. (Duality of prev and post.) Let A ∈ A and l ∈ L; then

\[
\begin{align*}
\text{POST}(A, l) &= \text{PREV(TRAN(A), l);} \\
\text{POST(TRAN(A), l)} &= \text{PREV(A, l)}.
\end{align*}
\]
Proof. Let $A \in \mathcal{A}$ and let $l \in \mathcal{L}$.

\[
\begin{array}{ll}
\text{TS} & \text{post}(A, l) = \text{prev}(\text{tran}(A), l) \\
\text{H0} & \text{Definition 7, the prev function.} \\
\text{H1} & \text{Definition 7, the post function.} \\
\text{H2} & \text{Definition 23, transposed abstract domain.}
\end{array}
\]

We proceed by proving separately the two inclusions

\[
\begin{align*}
\text{post}(A, l) & \subseteq \text{prev}(\text{tran}(A), l); \quad \text{(C1)} \\
\text{post}(A, l) & \supseteq \text{prev}(\text{tran}(A), l). \quad \text{(C2)}
\end{align*}
\]

Let $m \in \mathcal{L}$. For the first inclusion (C1)

\[
\begin{array}{ll}
\text{TS} & m \in \text{prev}(\text{tran}(A), l) \\
\text{H3} & m \in \text{post}(A, l) \quad \text{(C1)} \\
\text{D0} & (l, m) \in A \quad \text{(H3, H1)} \\
\text{D1} & (m, l) \in \text{tran}(A) \quad \text{(D0, H2)} \\
\text{(*)} & m \in \text{prev}(\text{tran}(A), l) \quad \text{(D1, H0)}
\end{array}
\]

For the second inclusion (C2)

\[
\begin{array}{ll}
\text{TS} & m \in \text{post}(A, l) \\
\text{H3} & m \in \text{prev}(\text{tran}(A), l) \quad \text{(C2)} \\
\text{D0} & (m, l) \in \text{tran}(A) \quad \text{(H3, H0)} \\
\text{D1} & (l, m) \in A \quad \text{(D0, H2)} \\
\text{(*)} & m \in \text{post}(A, l) \quad \text{(D1, H1)}
\end{array}
\]

The other half of this lemma can be proved observing that the transpose function is idempotent and applying this result to the first part of the lemma.

\[
\begin{array}{ll}
\text{TS} & \text{post}(\text{tran}(A), l) = \text{prev}(A, l) \\
\text{H0} & \text{post}(A, l) = \text{prev}(\text{tran}(A), l) \quad \text{(H1)} \\
\text{H1} & \text{Lemma 19, transpose is idempotent.} \\
\text{D0} & \text{prev}(A, l) = \text{prev}\big(\text{tran}\big(\text{tran}(A)\big), l\big) \quad \text{(H1)} \\
\text{D1} & \text{prev}\big(\text{tran}\big(\text{tran}(A)\big), l\big) = \text{post}(\text{tran}(A), l) \quad \text{(H0)} \\
\text{(*)} & \text{post}(\text{tran}(A), l) = \text{prev}(A, l) \quad \text{(D1, D0)}
\end{array}
\]

\[\square\]

Lemma 21. (Monotonicity of prev.) Let $A, B \in \mathcal{A}$ and $l \in \mathcal{L}$; then

\[A \subseteq B \implies \text{prev}(A, l) \subseteq \text{prev}(B, l).\]

Proof. Let $A, B \in \mathcal{A}$ such that $A \subseteq B$ and let $l \in \mathcal{L}$.
Lemma 22. (Duality of extended prev and post.) Let $A \in \mathcal{A}$ and $L \subseteq \mathcal{L}$; then

$$\text{post}(A, L) = \text{prev}(\text{tran}(A), L);$$

$$\text{post}(\text{tran}(A), L) = \text{prev}(A, L).$$

Proof. This result comes easily from the definition of the extended prev and post functions (Definition 8) applying the result of duality of prev and post (Lemma 20).

Lemma 23. (Monotonicity of the extended prev 1.) Let $A, B \in \mathcal{A}$ and $L \subseteq \mathcal{L}$; then

$$A \subseteq B \implies \text{prev}(A, L) \subseteq \text{prev}(B, L).$$

Proof. Let $A, B \in \mathcal{A}$ such that $A \subseteq B$ and let $L \subseteq \mathcal{L}$.

Lemma 24. (Monotonicity of the extended prev 2.) Let $A \in \mathcal{A}$ and $L, M \subseteq \mathcal{L}$; then

$$L \subseteq M \implies \text{prev}(A, L) \subseteq \text{prev}(A, M).$$

Proof. Let $A \in \mathcal{A}$ and let $L \subseteq M \subseteq \mathcal{L}$. 

2.4 Results
Lemma 25. (Location closure.) Let $A \in \mathcal{A}$ and $l \in \mathcal{L}$; then

$$\text{post}(A, l) \neq \emptyset \implies l \in \text{prev}(A, \text{post}(A, l)).$$

Proof. Let $A \in \mathcal{A}$ and let $l \in \mathcal{L}$ such that $\text{post}(A, l) \neq \emptyset$. Let then $m \in \text{post}(A, l)$.

\begin{center}
\begin{tabular}{ll}
TS & $l \in \text{prev}(A, \text{post}(A, l))$ \\
H0 & $m \in \text{post}(A, l)$ \\
H1 & Lemma 24, monotonicity of extended prev 2. \\
H2 & Definition 7, the prev function. \\
H3 & Definition 7, the post function. \\
D0 & $\text{prev}(A, \{m\}) \subseteq \text{prev}(A, \text{post}(A, l))$ (H0, H1) \\
D1 & $(l, m) \in A$ (H0, H3) \\
D2 & $l \in \text{prev}(A, \{m\})$ (D1, H2) \\
\checkmark & $l \in \text{prev}(A, \text{post}(A, l))$ (D2, D0) \\
\end{tabular}
\end{center}

Lemma 26. (Extended location closure.) Let $A \in \mathcal{A}$ and $L \subseteq \mathcal{L}$; then

$$\gamma(A) \neq \emptyset \implies L \subseteq \text{prev}(A, \text{post}(A, L)).$$

Proof. Let $A \in \mathcal{A}$ such that $\gamma(L) \neq \emptyset$ and let $L \subseteq \mathcal{L}$. If $L = \emptyset$ then the thesis is trivially verified. Otherwise, let $l \in L$. 

\[2.4 \text{ Results}\]
Lemma 27. (Monotonicity of extended eval 3.) Let $A \in \mathcal{A}$, $e \in \text{Expr}$ and $i \in \mathbb{N}$; then

$$\gamma(A) \neq \emptyset \implies \text{eval}(A, e, i + 1) \subseteq \text{prev}(A, \text{eval}(A, e, i)).$$

Proof. Let $A \in \mathcal{A}$ such that $\gamma(A) \neq \emptyset$, let $e \in \text{Expr}$ and let $i \in \mathbb{N}$. We proceed by induction on the definition of the extended eval function.

For every $i$ let $e = l \in \mathcal{L}$.

Let $e = \ast f$ with $f \in \text{Expr}$. For $i = 0$

<table>
<thead>
<tr>
<th>Step</th>
<th>Premise</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS</td>
<td>$\text{eval}(A, l, i + 1) \subseteq \text{prev}(A, \text{eval}(A, l, i))$</td>
<td></td>
</tr>
<tr>
<td>H0</td>
<td>$\text{eval}(A, l, i + 1) = \emptyset$</td>
<td>(H0)</td>
</tr>
<tr>
<td>$\times$</td>
<td>$\text{eval}(A, l, i + 1) \subseteq \text{prev}(A, \text{eval}(A, l, i))$</td>
<td>(D0)</td>
</tr>
</tbody>
</table>
For $i > 0$ for convenience of notation let $i = j + 1$.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TS</td>
<td>$\text{EVAL}(A, * f, j + 2) \subseteq \text{PREV}(A, \text{EVAL}(A, * f, j + 1))$</td>
</tr>
<tr>
<td>H2</td>
<td>$\text{EVAL}(A, f, j + 1) \subseteq \text{PREV}(A, \text{EVAL}(A, f, j))$ (ind. hyp.)</td>
</tr>
<tr>
<td>D0</td>
<td>$\text{EVAL}(A, * f, j + 2) = \text{EVAL}(A, f, j + 1)$ (H0)</td>
</tr>
<tr>
<td>D1</td>
<td>$\text{EVAL}(A, * f, j + 1) = \text{EVAL}(A, f, j)$ (H0)</td>
</tr>
<tr>
<td>$\times$</td>
<td>$\text{EVAL}(A, * f, j + 2) \subseteq \text{PREV}(A, \text{EVAL}(A, * f, j + 1))$ (D1, D0, H2)</td>
</tr>
</tbody>
</table>

Lemma 28. (Monotonicity of extended eval 3b.) Let $A \in \mathcal{A}$, $e \in \text{Expr}$ and $i \in \mathbb{N}$; then

$\gamma(A) \neq \emptyset \implies \text{POST}(A, \text{EVAL}(A, e, i + 1)) \subseteq \text{EVAL}(A, e, i)$.

Proof. Let $A \in \mathcal{A}$ such that $\gamma(A) \neq \emptyset$, let $e \in \text{Expr}$ and let $i \in \mathbb{N}$. We proceed again by induction on the definition of the extended eval function.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TS</td>
<td>$\text{POST}(A, \text{EVAL}(A, e, i + 1)) \subseteq \text{EVAL}(A, e, i)$</td>
</tr>
<tr>
<td>H0</td>
<td>Definition[17] the extended eval function.</td>
</tr>
<tr>
<td>H1</td>
<td>$\gamma(A) \neq \emptyset$</td>
</tr>
</tbody>
</table>

For every $i$ let $e = l \in L$.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TS</td>
<td>$\text{POST}(A, \text{EVAL}(A, l, i + 1)) \subseteq \text{EVAL}(A, l, i)$</td>
</tr>
<tr>
<td>D0</td>
<td>$\text{EVAL}(A, l, i + 1) = \emptyset$ (H0)</td>
</tr>
<tr>
<td>D1</td>
<td>$\text{POST}(A, \text{EVAL}(A, l, i + 1)) = \emptyset$ (D0, H2)</td>
</tr>
<tr>
<td>$\times$</td>
<td>$\text{POST}(A, \text{EVAL}(A, l, i + 1)) \subseteq \text{EVAL}(A, l, i)$ (D1)</td>
</tr>
</tbody>
</table>

Let $e = * f$ with $f \in \text{Expr}$. For $i = 0$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TS</td>
<td>$\text{POST}(A, \text{EVAL}(A, * f, 1)) \subseteq \text{EVAL}(A, * f, 0)$</td>
</tr>
<tr>
<td>H2</td>
<td>Definition[10] the eval function.</td>
</tr>
<tr>
<td>H3</td>
<td>Lemma[26] the extended location closure.</td>
</tr>
<tr>
<td>D0</td>
<td>$\text{EVAL}(A, * f, 1) = \text{EVAL}(A, f)$ (H0)</td>
</tr>
<tr>
<td>D1</td>
<td>$\text{POST}(A, \text{EVAL}(A, * f, 1)) = \text{POST}(A, \text{EVAL}(A, f))$ (D0)</td>
</tr>
<tr>
<td>D2</td>
<td>$\text{POST}(A, \text{EVAL}(A, * f, 1)) = \text{EVAL}(A, * f)$ (D1, H2)</td>
</tr>
<tr>
<td>$\times$</td>
<td>$\text{POST}(A, \text{EVAL}(A, * f, 1)) = \text{EVAL}(A, * f, 0)$ (D2, H0)</td>
</tr>
</tbody>
</table>

For $i > 0$ for convenience of notation let $i = j + 1$.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TS</td>
<td>$\text{POST}(A, \text{EVAL}(A, * f, j + 2)) \subseteq \text{EVAL}(A, * f, j + 1)$</td>
</tr>
<tr>
<td>H2</td>
<td>$\text{POST}(A, \text{EVAL}(A, * f, j + 1)) \subseteq \text{EVAL}(A, * f, j)$ (ind. hyp.)</td>
</tr>
<tr>
<td>D0</td>
<td>$\text{EVAL}(A, * f, j + 2) = \text{EVAL}(A, f, j + 1)$ (H0)</td>
</tr>
<tr>
<td>D1</td>
<td>$\text{EVAL}(A, * f, j + 1) = \text{EVAL}(A, f, j)$ (H0)</td>
</tr>
<tr>
<td>$\times$</td>
<td>$\text{POST}(A, \text{EVAL}(A, * f, j + 2)) \subseteq \text{EVAL}(A, * f, j + 1)$ (D1, D0, H2)</td>
</tr>
</tbody>
</table>

2.4 Results
Lemma 29. (Monotonicity of target.) Let \( A, B \in A, e \in \text{Expr}, L \subseteq \mathcal{L} \) and \( i, j \in \mathbb{N} \); then

\[
( A \subseteq B \land i \leq j \land \gamma(A) \neq \emptyset ) \implies ( \text{eval}(A, e, i) \subseteq \text{targ}(B, L, e, i) \implies \text{eval}(A, e, j) \subseteq \text{targ}(B, L, e, j) )
\]

*Proof.* Note that if \( i = j \) then the consequent of the implication in the above statement is always true thus the thesis is trivially verified. For the case \( i < j \) we will prove that

\[
( A \subseteq B \land \gamma(A) \neq \emptyset ) \implies ( \text{eval}(A, e, i) \subseteq \text{targ}(B, L, e, i) \implies \text{eval}(A, e, i + 1) \subseteq \text{targ}(B, L, e, i + 1) )
\]
as this implies, by a trivial induction on \( i \), the original result. Let \( A, B \in A \) such that \( A \subseteq B \) and \( \gamma(A) \neq \emptyset \), let \( i \in \mathbb{N}, e \in \text{Expr} \) and \( L \subseteq \mathcal{L} \).

\[
\text{TS } \text{eval}(A, e, i + 1) \subseteq \text{targ}(B, L, e, i + 1)
\]

<table>
<thead>
<tr>
<th>\text{H0}</th>
<th>A \subseteq B</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{H1}</td>
<td>\text{eval}(A, e, i) \subseteq \text{targ}(B, L, e, i)</td>
</tr>
<tr>
<td>\text{H2}</td>
<td>\gamma(A) \neq \emptyset</td>
</tr>
<tr>
<td>\text{H3}</td>
<td>Definition 18, the target function.</td>
</tr>
<tr>
<td>\text{H4}</td>
<td>Lemma 27, monotonicity of the extended eval 3.</td>
</tr>
<tr>
<td>\text{H5}</td>
<td>Lemma 23, monotonicity of the extended prev 1.</td>
</tr>
<tr>
<td>\text{H6}</td>
<td>Lemma 24, monotonicity of the extended prev 2.</td>
</tr>
<tr>
<td>\text{H7}</td>
<td>Lemma 14, monotonicity of the extended eval 1.</td>
</tr>
</tbody>
</table>

| \text{D0} | \text{targ}(B, L, e, i + 1) = \text{eval}(B, e, i + 1) \cap \text{prev}(B, \text{targ}(B, L, e, i)) |
| \text{D1} | \text{eval}(A, e, i + 1) \subseteq \text{eval}(B, e, i + 1) |
| \text{D2} | \text{eval}(A, e, i + 1) \subseteq \text{prev}(A, \text{eval}(A, e, i)) |
| \text{D3} | \text{eval}(A, e, i + 1) \subseteq \text{prev}(B, \text{eval}(A, e, i)) |
| \text{D4} | \text{eval}(A, e, i + 1) \subseteq \text{prev}(B, \text{targ}(B, L, e, i)) |
| \text{D5} | \text{eval}(A, e, i + 1) \subseteq \text{prev}(B, \text{targ}(B, L, e, i)) |

\[
\text{eval}(A, e, i + 1) \subseteq \text{targ}(B, L, e, i + 1)
\]

Lemma 30. (Generalized correctness of filter 2.) Let \( A, B \in A, L \subseteq \mathcal{L} \) and \( e \in \text{Expr} \); then

\[
( A \subseteq B \land \gamma(A) \neq \emptyset \land \text{eval}(A, e) \subseteq L ) \implies A \subseteq \phi(B, L, e).
\]

*Proof.* Let \( A, B \in A \), let \( L \subseteq \mathcal{L} \) and let \( e \in \text{Expr} \). We distinguish two cases

\[
e = l \in \mathcal{L}; \quad \text{C1}
\]
\[
e = * f \in \text{Expr} \setminus \mathcal{L}. \quad \text{C2}
\]

For the first case (C1) let \( l \in \mathcal{L} \). We have

\section*{2.4 Results}
Now the second case (C2).

To prove TS we will prove the equivalent result T2. Let \((l, m) \in A\) and let \(i \in \mathbb{N}\).

We proceed by induction on \(i\). For the base case let \(i = 0\).

For the inductive case let \(i > 0\). For convenience of notation let \(j \in \mathbb{N}\) such that \(i = j + 1\)

We distinguish two cases depending on the cardinality of the target set

\[
\# \text{TARG}(B, L, e, j + 1) \neq 1; \quad \text{(C2.1)}
\]

\[
\# \text{TARG}(B, L, e, j + 1) = 1. \quad \text{(C2.2)}
\]

For the first case (C2.1), we have

\[T2 \quad (l, m) \in \phi(B, L, e) \]

We distinguish two cases depending on the cardinality of the target set

\[
\# \text{TARG}(B, L, e, j + 1) \neq 1; \quad \text{(C2.1)}
\]

\[
\# \text{TARG}(B, L, e, j + 1) = 1. \quad \text{(C2.2)}
\]

For the first case (C2.1), we have
For the second case (C2.2), assume that

\[
\begin{align*}
\text{H7} & \quad \# \text{TARG}(B, L, e, j + 1) \neq 1 \quad \text{(C2.1)} \\
\text{D0} & \quad \phi(B, L, e, j + 1) = \phi(B, L, e, j) \quad \text{(H7, H4)} \\
\text{\textbullet} & \quad (l, m) \in \phi(B, L, e, j + 1) \quad \text{(D0, H6)}
\end{align*}
\]

H7 \# TARG(B, L, e, j + 1) \neq 1 (C2.1)

D0 \phi(B, L, e, j + 1) = \phi(B, L, e, j) (H7, H4)

\text{\textbullet} (l, m) \in \phi(B, L, e, j + 1) (D0, H6)

For the second case (C2.2), assume that

\[
\begin{align*}
\text{H7} & \quad \# \text{TARG}(B, L, e, j + 1) = 1 \quad \text{(C2.2)} \\
\text{D0} & \quad \phi(B, L, e, j + 1) = \phi(B, L, e, j) \\
& \quad \setminus \left( \text{TARG}(B, L, e, j + 1) \times (L \setminus \text{TARG}(B, L, e, j)) \right) \quad \text{(H7, H4)}
\end{align*}
\]

We distinguish two sub-cases

\[
\begin{align*}
l \notin \text{TARG}(B, L, e, j + 1); & \quad \text{(C2.2.1)} \\
l \in \text{TARG}(B, L, e, j + 1). & \quad \text{(C2.2.2)}
\end{align*}
\]

For the first sub-case (C2.2.1) assume that

\[
\begin{align*}
\text{H8} & \quad l \notin \text{TARG}(B, L, e, j + 1) \quad \text{(C2.2.1)} \\
\text{D1} & \quad (l, m) \notin \text{TARG}(B, L, e, j + 1) \times (L \setminus \text{TARG}(B, L, e, j)) \quad \text{(H8)} \\
\text{\textbullet} & \quad (l, m) \in \phi(B, L, e, j + 1) \quad \text{(D1, D0, H6)}
\end{align*}
\]

For the second sub-case (C2.2.2) we have

2.4 Results
Lemma 31. (Correctness of filter 2.) Let $A \in \mathcal{A}$, $L \subseteq \mathcal{L}$ and $e \in \text{Expr}$; then

$$\forall C \in \gamma(A) : \text{eval}(C, e) \subseteq L \implies C \in \gamma(\phi(A, L, e)).$$

Proof. This is a simple corollary of Lemma 30. Let $A \in \mathcal{A}$ and let $C \in \gamma(A)$. Let $e \in \text{Expr}$ and let $L \subseteq \mathcal{L}$ such that $\text{eval}(C, e) \subseteq L$.

2.4 Results
Lemma 32. (Equality target.) Let \( A \in \mathcal{A} \) and \( e, f \in \text{Expr} \). For convenience of notation let \( c \in \text{Cond} \) be such that \( c = (\text{eq}, e, f) \). Finally, let \( C \in \phi(\gamma(A), c) \). Then
\[
\text{eval}(C, e) \subseteq \text{eval}(A, e) \cap \text{eval}(A, f).
\]

Proof. Let \( A \in \mathcal{A} \), let \( (\text{eq}, e, f) = c \in \text{Cond} \) and let \( C \in \mathcal{C} \) such that \( C \in \phi(\gamma(A), c) \). Note that from the definition of the concrete semantics of the filter operation (Definition 16) we have
\[
\phi(\gamma(A), c) = \gamma(A) \cap \text{modelset}(c).
\]
Thus, \( C \in \text{modelset}(c) \) and \( C \in \gamma(A) \).

Lemma 33. (Inequality target.) Let \( A \in \mathcal{A} \) and \( e, f \in \text{Expr} \). For convenience of notation let \( c \in \text{Cond} \) be such that \( c = (\text{neq}, e, f) \). Let \( C \in \phi(\gamma(A), c) \) and let
\[
I = \text{eval}(A, e) \cap \text{eval}(A, f),
\]
\[
E = \text{eval}(A, e) \setminus \text{eval}(A, f),
\]
\[
F = \text{eval}(A, f) \setminus \text{eval}(A, e).
\]

2.4 Results 73
Then
\[ \# I = 1 \implies \text{eval}(C, e) \subseteq E \lor \text{eval}(C, f) \subseteq F. \]

**Proof.** Let \( A \in \mathcal{A} \), let \((\text{neq}, e, f) = c \in \text{Cond}\) and let \( C \in \mathcal{C}\). To show the thesis we assume that \( \# I = 1 \) and \( \text{eval}(C, e) \not\subseteq E \) and then we show that \( \text{eval}(C, f) \subseteq F \).

\[
\begin{array}{ll}
\text{TS} & \text{eval}(C, f) \subseteq \text{eval}(A, f) \setminus \text{eval}(A, e) \\
H0 & C \in \gamma(A) \\
H1 & C \models c \\
H2 & \#(\text{eval}(A, e) \cap \text{eval}(A, f)) = 1 \\
H3 & \neg(\text{eval}(C, e) \subseteq \text{eval}(A, e) \setminus \text{eval}(A, f)) \\
H4 & \text{Definition 6, the concretization function.} \\
H5 & \text{Lemma 13, monotonicity of eval.} \\
H6 & \text{Definition 13, value of conditions.} \\
H7 & \text{Lemma 2, eval cardinality on } C. \\
D0 & \neg(\text{eval}(C, e) \subseteq \text{eval}(A, e) \land \text{eval}(C, e) \not\subseteq \text{eval}(A, f)) \quad (H3) \\
D1 & \text{eval}(C, e) \not\subseteq \text{eval}(A, e) \lor \text{eval}(C, e) \subseteq \text{eval}(A, f) \quad (D0) \\
D2 & C \subseteq A \quad (H0, H4) \\
D3 & \text{eval}(C, e) \subseteq \text{eval}(A, e) \quad (D2, H5) \\
D4 & \text{eval}(C, e) \subseteq \text{eval}(A, e) \land \text{eval}(C, e) \subseteq \text{eval}(A, f) \quad (D3, D1) \\
D5 & \text{eval}(C, e) \subseteq \text{eval}(A, e) \cap \text{eval}(A, f) \quad (D4) \\
D6 & \# \text{eval}(C, e) = 1 \quad (H7) \\
D7 & \text{eval}(C, e) = \text{eval}(A, e) \cap \text{eval}(A, f) \quad (H2, D6, D5) \\
D8 & \text{eval}(C, e) \neq \text{eval}(C, f) \quad (H1, H6) \\
D9 & \text{eval}(C, f) \neq \text{eval}(A, e) \cap \text{eval}(A, f) \quad (D6, D5, H2) \\
D10 & \# \text{eval}(C, f) = 1 \quad (H7) \\
D11 & \text{eval}(C, f) \not\subseteq \text{eval}(A, e) \cap \text{eval}(A, f) \quad (D10, D9) \\
D12 & \text{eval}(C, f) \not\subseteq \text{eval}(A, e) \lor \text{eval}(C, f) \not\subseteq \text{eval}(A, f) \quad (D11) \\
D13 & \text{eval}(C, f) \subseteq \text{eval}(A, f) \quad (D2, H5) \\
D14 & \text{eval}(C, f) \subseteq \text{eval}(A, f) \land \text{eval}(C, f) \not\subseteq \text{eval}(A, e) \quad (D13, D12) \\
\checkmark & \text{eval}(C, f) \subseteq \text{eval}(A, f) \setminus \text{eval}(A, e) \quad (D14)
\end{array}
\]

\( \square \)

**Proof.** (Correctness of the filter, Theorem 9) Let \( A \in \mathcal{A} \), let \( c \in \text{Cond}\) and let \( C \in \mathcal{C} \). For convenience of notation let \( I = \text{eval}(A, e) \cap \text{eval}(A, f) \).

\[
\begin{array}{ll}
\text{TS} & C \in \gamma(\phi(A, c)) \\
H0 & C \models c \\
H1 & C \in \gamma(A) \\
H2 & \text{Lemma 31, correctness of filter 2.} \\
H3 & \text{Definition 21, filter 3.} \\
H4 & \text{Definition 6, concretization function.}
\end{array}
\]

2.4 Results
We distinguish two cases

\[ c = (eq, e, f); \quad (C1) \]
\[ c = (neq, e, f). \quad (C2) \]

For the first case (C1) let \( c = (eq, e, f) \).

<table>
<thead>
<tr>
<th>H5</th>
<th>Lemma [32], the equality target.</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>( \text{eval}(C, e) \subseteq I ) (H5, H1, H0)</td>
</tr>
<tr>
<td>D1</td>
<td>( \text{eval}(C, f) \subseteq I ) (H5, H1, H0)</td>
</tr>
<tr>
<td>D2</td>
<td>( C \in \gamma(\phi(A, I, e)) ) (D0, H1, H2)</td>
</tr>
<tr>
<td>D3</td>
<td>( C \in \gamma(\phi(A, I, f)) ) (D1, H1, H2)</td>
</tr>
<tr>
<td>D4</td>
<td>( C \subseteq \phi(A, I, e) ) (D2, H4)</td>
</tr>
<tr>
<td>D5</td>
<td>( C \subseteq \phi(A, I, f) ) (D3, H4)</td>
</tr>
<tr>
<td>D6</td>
<td>( \phi(A, c) = \phi(A, I, e) \cap \phi(A, I, f) ) (H3)</td>
</tr>
<tr>
<td>D7</td>
<td>( C \subseteq \phi(A, I, e) \cap \phi(A, I, f) ) (D5, D4)</td>
</tr>
<tr>
<td>D8</td>
<td>( C \subseteq \phi(A, c) ) (D7, D6)</td>
</tr>
<tr>
<td>✩</td>
<td>( C \in \gamma(\phi(A, c)) ) (D8, D4)</td>
</tr>
</tbody>
</table>

For the second case (C2) let \( c = (neq, e, f) \). We distinguish two sub-cases.

\[ \# I \neq 1; \quad (C2.1) \]
\[ \# I = 1. \quad (C2.2) \]

For the first sub-case (C2.1)

<table>
<thead>
<tr>
<th>H5</th>
<th># I \neq 1 (C2.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>( \phi(A, c) = A ) (H3, H5)</td>
</tr>
<tr>
<td>✩</td>
<td>( C \in \gamma(\phi(A, c)) ) (D0, H1)</td>
</tr>
</tbody>
</table>

In the second sub-case (C2.2) for convenience of notation let \( E, F \subseteq \mathcal{L} \) be defined as

\[ E = \text{eval}(A, e) \setminus \text{eval}(A, f), \]
\[ F = \text{eval}(A, f) \setminus \text{eval}(A, e). \]

<table>
<thead>
<tr>
<th>H5</th>
<th># I = 1 (C2.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>( \phi(A, c) = \phi(A, E, e) \cup \phi(A, F, f) ) (H3, H5)</td>
</tr>
<tr>
<td>D1</td>
<td>( \text{eval}(C, e) \subseteq E \lor \text{eval}(C, f) \subseteq F ) (H6, H5, H1, H0)</td>
</tr>
<tr>
<td>D2</td>
<td>( \text{eval}(C, e) \subseteq E \iff C \in \gamma(\phi(A, E, e)) ) (H2, H1)</td>
</tr>
<tr>
<td>D3</td>
<td>( \text{eval}(C, f) \subseteq F \iff C \in \gamma(\phi(A, F, f)) ) (H2, H1)</td>
</tr>
<tr>
<td>D4</td>
<td>( C \in \gamma(\phi(A, E, e)) \lor C \in \gamma(\phi(A, F, f)) ) (D1, D2, D3)</td>
</tr>
<tr>
<td>D5</td>
<td>( C \subseteq \phi(A, E, e) \lor C \subseteq \phi(A, F, f) ) (D4, H4)</td>
</tr>
<tr>
<td>D6</td>
<td>( C \subseteq \phi(A, E, e) \cup \phi(A, F, f) ) (D5)</td>
</tr>
<tr>
<td>D7</td>
<td>( C \subseteq \phi(A, c) ) (D6, D0)</td>
</tr>
<tr>
<td>✩</td>
<td>( C \in \gamma(\phi(A, c)) ) (D7, H4)</td>
</tr>
</tbody>
</table>

\[ \square \]
2.5 Precision Limits

This section presents some considerations about the precision of the analysis; starting from questions that regard the points-to representation, that is, common to all points-to methods; to questions about the specific method presented.

2.5.1 Precision of the Points-To Representation

Reconsider now the correctness results presented in Theorem 8 and 9. Let \(A \in \mathcal{A}\) and \(a \in \text{Assignments}\), the correctness of the assignment

\[\gamma(\text{assign}(A, a)) \supseteq \text{assign}(\gamma(A), a),\]

using the definition of the abstraction function (Definition 6) and Lemma 6 implies that

\[\text{assign}(A, a) \supseteq \alpha\left(\text{assign}(\gamma(A), c)\right)\]

\[= \bigcup \{C \mid C \in \text{assign}(\gamma(A), c)\}\]

\[= \bigcup \{\text{assign}(C, a) \mid C \in \gamma(A)\}.\]

Moreover, given \(c \in \text{Cond}\), the correctness of the filter

\[\gamma(\phi(A, c)) \supseteq \phi(\gamma(A), c),\]

using Lemma 6 implies that

\[\phi(A, c) \supseteq \alpha(\phi(\gamma(A), c))\]

\[= \bigcup \{C \mid C \in \phi(\gamma(A), c)\}\]

\[= \bigcup \{C \mid C \in \gamma(A) \cap \text{modelset}(c)\}.\]

Expressed in this form, the correctness results highlight the attribute independent nature of the points-to abstract domain; in this sense these results provide a limit to the precision attainable. Note that these limits,

\[\bigcup \{\text{assign}(C, a) \mid C \in \gamma(A)\},\]

\[\bigcup \{C \mid C \in \gamma(A) \cap \text{modelset}(c)\},\]

do not depend in any way on the definition of the abstract operations but only on the characteristics of the abstract and concrete domains (Definition 5), their semantics (Definition 6) and on the concrete semantics of the operations (Definition 15 and 16). In other words, these are limitations of the points-to representation and are thus common to any method based on it. In Section 1.3 we have presented an example of the limitations of the alias query representation; now we show some examples of the limitations of the points-to representation, which is strictly less powerful.
Below, an extract of the concrete alias query $\text{ALIAS}_{m_0}$ induced by the concrete memory description $m_0 \in \text{Mem}$. Above, a graphical representation of the points-to information $C_0 \in C$ associated to the same memory.

<table>
<thead>
<tr>
<th>ALIAS$_{m_0}$</th>
<th>a</th>
<th>b</th>
<th>*q</th>
</tr>
</thead>
<tbody>
<tr>
<td>*p</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>*q</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

As above, on the concrete memory description $m_1 \in \text{Mem}$.

<table>
<thead>
<tr>
<th>ALIAS$_{m_1}$</th>
<th>a</th>
<th>b</th>
<th>*q</th>
</tr>
</thead>
<tbody>
<tr>
<td>*p</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>*q</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Above, a graphical representation of the points-to abstraction $A$

\[ \alpha(\{C_0, C_1\}) = C_0 \cup C_1 = A \in A. \]

Below, an extract of the abstract alias query

\[ \alpha(\{\text{ALIAS}_{m_0}, \text{ALIAS}_{m_1}\}) = \text{ALIAS}_{m_0} \sqcup \text{ALIAS}_{m_1} = \text{ALIAS}^\sharp \in \text{AliasQ}^\sharp. \]

Note that $\text{ALIAS}^\sharp(*p, *q) = 1$, that is the abstract alias query is able to represent that ‘p’ and ‘q’ always point to the same location.

Figure 2.14: a representation of the points-to and alias information associated to the code in Listing 2.5.

2.5 Precision Limits
Listing 2.5: in this example two executions are possible. However, in both of them at line 7 the pointers ‘p’ and ‘q’ point to the same location.

Here both the table and the graph represent the points-to information $A$. Note that

$$\#(\text{eval}(A, *p) > 1;$$

that is (Definition 11) $A(*p, *q) = \top$, i.e., the points-to representation is unable to express that ‘p’ and ‘q’ will definitely point to the same location.

The concrete points-to information $C_2 \in \mathcal{C}$ is a spurious element of $\gamma(A)$. Note that

$$\text{alias}_{C_2} \notin \gamma(\text{alias}^s);$$

that is the concrete alias relation $\text{alias}_{C_2}$ induced by $C_2$ would not be generated using the alias representation.

The concrete points-to information $C_3 \in \mathcal{C}$ is another spurious element of $\gamma(A)$. Note that:

$$\gamma(A) = \{C_0, C_1, C_2, C_3\};$$
$$\gamma(\text{alias}^s) = \{\text{alias}_{C_0}, \text{alias}_{C_1}\}.$$

Figure 2.15: continuation of Figure 2.14

2.5 Precision Limits
Example 10. An abstract alias query is able to correctly represent when two pointers point to the same location, also when the pointed location is not known. The points-to representation is unable to do it, as illustrated in Listing 2.5: at line 7 the abstract alias query that approximates the program is able express that in all of the possible executions, the expressions ‘*p’ and ‘*q’ are aliases, that is the variables ‘p’ and ‘q’ point to the same location. On the other hand, the most precise points-to approximation cannot capture this fact. Let $A \in \mathcal{A}$ where

$$\mathcal{L} = \{p, q, a, b\},$$
$$A = \{(p, a), (p, b), (q, a), (q, b)\}.$$

We have $\gamma(A) = \{C_0, C_1, C_2, C_3\}$ where

- $C_0 = \{(p, a), (q, a)\}$,
- $C_1 = \{(p, b), (q, b)\}$,
- $C_2 = \{(p, b), (q, a)\}$,
- $C_3 = \{(p, a), (q, b)\}$.

Consider the condition $(\text{eq}, *p, *q) = c \in \text{Cond}$; we have

$$\phi(\gamma(A), c) = \gamma(A) \cap \text{modelset}(c) = \{C_0, C_1\};$$

but the abstraction yields

$$\alpha(\phi(\gamma(A), c)) = \alpha(\{C_0, C_1\}) = C_0 \cup C_1 = \{(p, a), (q, a), (p, b), (q, b)\} = A.$$

The $\alpha(\{C_0, C_1\})$ is the most precise points-to abstraction that approximates both $C_0$ and $C_1$; however, it also approximates $C_2$ and $C_3$, which are not models of the condition $c$. Again, this is due to the fact that the points-to representation is attribute independent: in the above example we are unable to record that when a concrete element $C$ is such that $C \models c$ and $(p, a) \in C$ then also $(q, a) \in C$. This situation is also is illustrated in Figures 2.14 and 2.15.

In other words, this example shows that it is not possible to define the filter operation such that it always filters away all the concrete points-to descriptions that are not model of the supplied condition $c$. In symbols:

$$\neg(\forall A \in \mathcal{A} : \forall c \in \text{Cond} : \gamma(\phi(A, c)) \subseteq \text{modelset}(c)).$$

2.5 Precision Limits
Listing 2.6: in this example two executions are possible. However, in both of them at line 10 the expression ‘**r’ is an alias of ‘b’.

Figure 2.16: a representation of the points-to information before and after the execution of line 9 in Listing 2.6.
Below, an extract of the concrete alias query $\text{ALIAS}_{m_0}$ induced by the concrete memory description $m_0 \in \text{Mem}$. Above, a graphical representation of the points-to information $C_0 \in \mathcal{C}$ associated to the same memory.

As above, on the concrete memory description $m_1$.

Figure 2.17: this example shows how, in the code of Listing 2.6, the points-to representation fails to describe that ‘**r’ is alias of ‘b’ on all of the possible executions.

2.5 Precision Limits
Above, a graphical representation of the most precise points-to abstraction $A$

$$\alpha(\{C_0, C_1\}) = C_0 \cup C_1 = A \in \mathcal{A}.$$ 

Below, an extract of the abstract alias query

$$\alpha(\{\text{alias} m_0, \text{alias} m_1\})$$ 
$$= \text{alias} m_0 \sqcup \text{alias} m_1$$ 
$$= \text{alias}^2 \in \text{AliasQ}^2.$$ 

Note that $\text{alias}^2(\ast\ast r, b) = 1$, that is the abstract alias query is able to represent that the expressions $\ast\ast r$ and $b$ are definitely aliases.

Here both the table and the graph represent the points-to information $A$. Note that

$$\text{EVAL}(A, \ast\ast r) \neq \text{EVAL}(A, b),$$

that is (Definition 11) $A(\ast\ast r, b) = \top$, i.e., the points-to representation is unable to express that $\ast\ast r$ and $a$ will be definitely aliases.
Example 11. The points-to representation keeps track only of the relations between pointers and pointed objects that span exactly one level of indirection. For example, in Listing 2.6, the points-to representation is unable to \textit{natively} express that ‘**r’ is an alias of ‘b’, this information —though present in the \textit{complete} alias relation— is \textit{inferred} from the points-to pairs explicitly memorized by applying the transitive property: it is known that ‘r’ points to ‘p’ and that ‘p’ points to ‘b’; then it can be deduced that ‘*r’ points to ‘b’. But this step causes a loss of accuracy when there are more intermediate variables (Figure 2.16). The alias query representation is able describe that after the execution of line 9, the expression ‘**r’ is \textit{definitely} an alias of ‘b’, whereas the points-to representation fails to do it. Let $A \in \mathcal{A}$ such that

\[
\mathcal{L} = \{p, q, r, a, b, c\},
\]
\[
A = \{(r, p), (r, q), (p, a), (q, c)\}.
\]

We have that $\{C_0, C_1\} = \gamma(A)$ where

\[
C_0 = \{(r, p), (p, a), (q, c)\},
\]
\[
C_1 = \{(r, q), (p, a), (q, c)\}.
\]

Let $(*r,b) = x \in \text{Assignments}$. Performing the assignment on the elements found in the concretization of $A$ we obtain

\[
\text{ASSIGN}(C_0, x) = \{(r, p), (p, b), (q, c)\},
\]
\[
\text{ASSIGN}(C_1, x) = \{(r, q), (p, a), (q, b)\}.
\]
Computing the abstraction of the result of the concrete operation we find
\[ \alpha \left( \text{assign} \left( \gamma(A), x \right) \right) = \alpha \left( \{ \text{assign}(C_0, x), \text{assign}(C_1, x) \} \right) \]
\[ = \text{assign}(C_0, x) \cup \text{assign}(C_1, x) \]
\[ = A \cup \{(p, b), (q, b)\}. \]

Let
\[ C_3 = \{(r, p), (p, a), (q, b)\} \subseteq \alpha \left( \text{assign} \left( \gamma(A), x \right) \right); \]

note that

\[ \begin{array}{ll}
H0 & \text{Theorem 8, correctness of the assignment.} \\
H1 & \text{Lemma 4, monotonicity of the concretization function.} \\
H2 & \text{Lemma 6, the abstraction effect.} \\
H3 & \text{Definition 22, the abstraction function.} \\
D0 & \text{assign} \left( \gamma(A), x \right) \subseteq \gamma \left( \text{assign} \left( A, x \right) \right) \\
D1 & \alpha \left( \text{assign} \left( \gamma(A), x \right) \right) \subseteq \alpha \left( \gamma \left( \text{assign} \left( A, x \right) \right) \right) \\
D2 & \alpha \left( \gamma \left( \text{assign} \left( A, x \right) \right) \right) \subseteq \text{assign} \left( A, x \right) \\
D3 & \alpha \left( \text{assign} \left( \gamma(A), x \right) \right) \subseteq \text{assign} \left( A, x \right) \\
\star & \gamma \left( \alpha \left( \text{assign} \left( \gamma(A), x \right) \right) \right) \subseteq \gamma \left( \text{assign} \left( A, x \right) \right)
\end{array} \]

then
\[ C_3 \in \gamma \left( \text{assign} \left( A, a \right) \right); \]

but
\[ C_3 \neq \text{assign}(C_0, a); \]
\[ C_3 \neq \text{assign}(C_1, a); \]

that is, there exist no concrete elements \( C \in \gamma(A) \) such that \( C_3 = \text{assign}(C, a) \). Again this inaccuracy is due to the lack of relational information in the points-to representation: in this example, given a concrete element \( C \in \text{assign}(A, a) \), we are unable to tell that if \((r, p) \in C\) then \((p, b) \in C\) and \((q, b) \notin C\). The situation just described is illustrated in Figures 2.17, 2.18 and 2.19.

In other words, this example shows that it is not possible to formulate the assignment operation in such a way that each concrete element approximated by \( \text{assign}(A, a) \) can be expressed as the result of the concrete assignment performed on one of the elements of \( \gamma(A) \). In symbols
\[ \neg \left( \forall A \in \mathcal{A} : \forall a \in \text{Assignments} : \forall C \in \gamma \left( \text{assign} \left( A, a \right) \right) : \exists D \in \gamma(A) . C = \text{assign}(D, a) \right). \]

2.5 Precision Limits
2.5.2 Precision of the Presented Method

The two examples introduced above present a limitation of the form — all points-to based methods are not enough precise to capture this fact. In terms of the partial order of the domain this can be seen as a lower limit to the precision attainable with points-to based methods. On the other hand it is also interesting to find out what are the precision upper limits of the proposed method, i.e., statements of the form — the given points-to based method is enough precise to capture that fact. In particular, we want to analyze the situation of the presented method with respect to the limitations of the points-to representation, that is whether or not the inclusions in Theorem 8 and 9 are also equalities, i.e., if it holds that, for all $A \in \mathcal{A}$, $e \in \text{Expr}$, $a \in \text{Assignments}$ and $c \in \text{Cond}$

$$\bigcup \{ \text{eval}(C, e) \mid C \in \gamma(A) \} \supseteq \text{eval}(A, e);$$

$$\text{assign}(\gamma(A), a) \supseteq \gamma(\text{assign}(A, a));$$

$$\phi(\gamma(A), c) \supseteq \gamma(\phi(A, c)).$$

From the characterization presented in Section 2.5.1 these can be rewritten to stress the attribute independent nature of the points-to representation, i.e., by focusing on the single arcs instead of the whole points-to relation. Let $A \in \mathcal{A}$ such that $\gamma(A) \neq \emptyset$ then we have

$$\forall l \in \text{eval}(A, e) : \exists C \in \gamma(A) \cdot \text{eval}(C, e) = \{l\},$$

$$\forall (l, m) \in \text{assign}(A, a) : \exists C \in \gamma(A), (l, m) \in \text{assign}(C, a),$$

$$\forall (l, m) \in \phi(A, c) : \exists C \in \gamma(A) \cdot C \in \text{modelset}(c) \land (l, m) \in C,$$

respectively. Unfortunately, for all these cases there exists a counterexample.

The Abstract Evaluation Is Not Optimal

The following example highlights that the abstract evaluation function (Definition 10) is not optimal with respect to the points-to representation, i.e., there exists $A \in \mathcal{A}$, $\gamma(A) \neq \emptyset$ and $e \in \text{Expr}$ such that

$$\text{eval}(A, e) \setminus \bigcup \{ \text{eval}(C, e) \mid C \in \gamma(A) \} \neq \emptyset.$$ 

Example 12. Let $A \in \mathcal{A}$ such that

$$\mathcal{L} = \{a, b, c\},$$

$$A = \{(a, a), (a, b), (b, c)\}.$$ 

We have that $\{C_1, C_2\} = \gamma(A)$ where

$$C_1 = \{(a, a), (b, c)\},$$

$$C_2 = \{(a, b), (b, c)\}.$$ 

\footnote{Note that the additional hypothesis, $\gamma(A) \neq \emptyset$, is required by Lemma 6 to prove the opposite of the inclusions used for the correctness results.}
Figure 2.20: the abstract evaluation function is not optimal.

Consider the expression $e = **a$. Performing the evaluation of $e$ as described in Definition 10 we obtain

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{EVAL}(C_1, e, i)$</th>
<th>$\text{EVAL}(C_2, e, i)$</th>
<th>$\text{EVAL}(A, e, i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>${a}$</td>
<td>${a}$</td>
<td>${a}$</td>
</tr>
<tr>
<td>1</td>
<td>${a}$</td>
<td>${b}$</td>
<td>${a, b}$</td>
</tr>
<tr>
<td>0</td>
<td>${a}$</td>
<td>${c}$</td>
<td>${a, b, c}$</td>
</tr>
</tbody>
</table>

Note that $b \in \text{EVAL}(A, e)$ but there exist no $C \in \gamma(A)$ such that $\{b\} = \text{EVAL}(C, e)$, indeed

$$\bigcup_{C \in \gamma(A)} \text{EVAL}(C, e) = \{a, c\}.$$  

The spurious location $b$ in the result of the evaluation of the expression $e$ in $A$ is due to the fact that the formulation of the abstract evaluation does not exploit that in a concrete points-to description a location can point to only one location; in this case there exists no $C \in \gamma(A)$ such that $\{(a, a), (a, b)\} \subseteq C$. A graphical representation of this example is reported in Figures 2.20 and 2.21.

**The Abstract Assignment Is Not Optimal**

We present another example that highlights how the abstract assignment operation formulated in Definition 14 is not optimal for the points-to representation, i.e., there exists $A \in \mathcal{A}$, $\gamma(A) \neq \emptyset$, $a \in \text{Assignments}$ such that

$$\text{assign}(A, a) \setminus \text{assign}(\gamma(A), a) \neq \emptyset.$$
Figure 2.21: the evaluation process of the expression \( \ast \ast a \) on the memory \( A \) of Example 12.

An optimal evaluation function would not follow the arc \((a, b)\) between \( i = 1 \) and \( i = 0 \) (the dashed arc in the figure).

Note that this limitation is still true also assuming to have an optimal abstract evaluation function.

**Example 13.** Let \( A \in \mathcal{A} \) such that

\[
\begin{align*}
\mathcal{L} &= \{a, b, c\}, \\
A &= \{(a, b), (a, c)\}.
\end{align*}
\]

We have that \( \{C_1, C_2\} = \gamma(A) \) where

\[
\begin{align*}
C_1 &= \{(a, b)\}, \\
C_2 &= \{(a, c)\}.
\end{align*}
\]

Let \((\ast a, \ast a) = x \in \text{Assignments}\). Performing the assignment \( x \) on the elements of \( \gamma(A) \) we obtain

\[
\begin{align*}
\text{assign}(C_1, x) &= \{(a, b), (b, b)\}, \\
\text{assign}(C_2, x) &= \{(a, c), (c, c)\}.
\end{align*}
\]

Computing the abstraction of the result of the concrete operation we find

\[
\alpha\left(\text{assign}(\gamma(A), x)\right) = \alpha\left(\{\text{assign}(C_1, x), \text{assign}(C_2, x)\}\right)
\]

\[
= \text{assign}(C_1, x) \cup \text{assign}(C_2, x)
\]

\[
= A \cup \{(b, b), (c, c)\}.
\]

Note that performing the abstract evaluation of the lhs and the rhs of the assignment as described in Definition 10 yields \( \text{eval}(A, \ast a) = \{b, c\} \), which is the most precise result possible for the abstract evaluation of the expression \( \ast a \), indeed

\[
\bigcup \left\{ \text{eval}(C, a) \mid C \in \gamma(A) \right\} = \text{eval}(C_1, \ast a) \cup \text{eval}(C_2, \ast a)
\]

\[
= \{b\} \cup \{c\} = \{b, c\}
\]

\[
= \text{eval}(A, \ast a).
\]

2.5 Precision Limits
In this case, the abstract assignment (Definition 14) yields
\[
\text{ASSIGN}(A, x) = A \cup \text{EVAL}(A, *a) \times \text{EVAL}(A, *a)
\]
\[
= A \cup \{b, c\} \times \{b, c\}
\]
\[
= A \cup \{(b, b), (b, c), (c, b), (c, c)\}.
\]
Note that
\[
\text{ASSIGN}(A, x) \setminus \text{ASSIGN}(\gamma(A), x) = \{(b, c), (c, b)\}.
\]
The arcs \{(b, c), (c, b)\} do not correspond to any concrete assignment: they are artifacts of this abstraction. But note that in this case the inaccuracy cannot be ascribed to the abstract evaluation of the expressions that, in this case, exposes an optimal behaviour. The problem is that the evaluation of the rhs and the lhs for the assignment are not related each other: this way it becomes possible that the lhs evaluates to ‘b’ and the rhs evaluates to ‘c’ — thus generating the spurious arc \((b, c)\) — also when the rhs and the lhs are the same expression. This example is illustrated in Figure 2.22.

The Abstract Filter Is Not Optimal

Finally, we report an example that shows the same inaccuracy in the filter operation, i.e., there exists \(A \in \mathcal{A}, \gamma(A) \neq \emptyset\), and \(c \in \text{Cond}\) such that
\[
\phi(A, c) \setminus \phi(\gamma(A), c) \neq \emptyset.
\]

**Example 14.** Let \(A \in \mathcal{A}\) such that
\[
\mathcal{L} = \{a, b\},
\]
\[
A = \{(a, a), (a, b), (b, b)\}.
\]
We have that \(\{C_1, C_2\} = \gamma(A)\) where
\[
C_1 = \{(a, a), (b, b)\},
\]
\[
C_2 = \{(a, b), (b, b)\}.
\]
Consider now the condition \((**a, b) = c \in \text{Cond}\). Since
\[
\text{EVAL}(C_1, **a) = \{a\},
\]
\[
\text{EVAL}(C_2, **a) = \{b\},
\]
only \(C_2\) satisfies \(c\), i.e.
\[
\phi(\gamma(A), c) = \gamma(A) \cap \text{MODELSET}(c) = \{C_2\}.
\]
Performing the filter operation as described in Definition 21 on \(A\) we do not improve the precision, that is \(\phi(A, c) = A\).

2.5 Precision Limits
The abstraction $A$ before the execution of the assignment $x = (\ast a, \ast a)$.

$\gamma(A) = \{C_1, C_2\}$.

The concrete memory description $C_1$.

The concrete memory description $\text{assign}(C_1, x)$.

The concrete memory description $C_2$.

The concrete memory description $\text{assign}(C_2, x)$.

The abstract memory

$$\alpha\left(\{\text{assign}(C_1, x), \text{assign}(C_2, x)\}\right) = \text{assign}(C_1, x) \cup \text{assign}(C_2, x).$$

The abstraction $\text{assign}(A, x)$ resulting from the execution of the assignment. The spurious arcs are $\{(b, c), (c, b)\}$.

Figure 2.22: the abstract assignment formulation is not optimal.
The abstraction $A$. 
\[ \gamma(A) = \{C_1, C_2\} \]

The concrete memory description $C_1$. This is not a model of 
\[ c = (**a, b) \].

The concrete memory description $C_2$. This is a model of $c$;

The abstraction $\phi(A, c) = A$. The spurious arc is \{(a, a)\}.

Figure 2.23: the abstract filter formulation is not optimal.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{EVAL}(A, **a, i)$</th>
<th>$\text{TARG}(A, **a, i)$</th>
<th>Removed arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{a}</td>
<td>{a}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>1</td>
<td>{a, b}</td>
<td>{a, b}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>0</td>
<td>{a, b}</td>
<td>{b}</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

Then note that  
\[ \phi(A, c) \setminus \alpha\left( \phi(\gamma(A), c) \right) = A \setminus C_2 = \{(a, a)\}. \]

This means that the filter is unable to remove the spurious arc $(a, a)$. A graphical representation of this situation is presented in Figure 2.23, while Figure 2.24 presents a graphical representation of the filter computation.

Though the current formulation of the filter operation is not optimal, in the next example we show that iterating the application of the filter on the same condition it is possible to refine the points-to approximation.

**Example 15.** Let $A \in \mathcal{A}$ such that 
\[ \mathcal{L} = \{a, b, c\}, \]
\[ A = \{(a, a), (a, b), (a, c), (b, c), (c, a)\} \].

2.5 Precision Limits
Consider the condition $x = (\text{eq}, **a, c) \in \text{Cond}$. From the definition of the evaluation function (Definition 10), we have

$$\text{eval}(A, **a) = \{a, b, c\},$$
$$\text{eval}(A, c) = \{c\}.$$

From the filter definition (Definition 21) we have

$$I = \text{eval}(A, **a) \cap \text{eval}(A, c) = \{c\},$$
$$\phi(A, x) = \phi(A, I, **a) \cap \phi(A, I, c) = \phi(A, \{c\}, **a) \cap \phi(A, \{c\}, c).$$

We consider only the lhs $\phi(A, \{c\}, **a)$ as, from the definition of the filter 2, it is clear that filtering on the rhs does not improve the precision of the approximation, that is, $\phi(A, \{c\}, c) = A$. We have

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{eval}(A, **a)$</th>
<th>$\text{TARG}(A, {c}, **a, i)$</th>
<th>$\phi(A, {c}, **a, i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>${a}$</td>
<td>${a}$</td>
<td>$A \setminus {(a,c)}$</td>
</tr>
<tr>
<td>1</td>
<td>${a, b, c}$</td>
<td>${a, b}$</td>
<td>$A$</td>
</tr>
<tr>
<td>0</td>
<td>${a, b, c}$</td>
<td>${c}$</td>
<td>$A$</td>
</tr>
</tbody>
</table>

That is, from the first application of the filter we can remove the spurious arc $(a, c)$. Now we proceed applying the filter again. Let $B = \phi(A, \{c\}, **a, i) = A \setminus \{(a,c)\}$. We have
Note that in the second application of the filter we are able to remove another arc, \((a, a)\), that it was not removed during the first iteration.

Another Consideration on the Precision of the Filter Operation

It is possible to show that the formulation of the abstract filter operation does not generate spurious memory descriptions not already present in the initial approximation, i.e., for all \(A \in A\) and \(c \in \text{Cond}\)

\[
\gamma(\phi(A, c)) \subseteq \gamma(A).
\]

Note that by composing this result with the result of correctness for the filter (Theorem 9) it is possible to write

\[
\gamma(A) \cap \text{modelset}(c) \subseteq \gamma(\phi(A, c)) \subseteq \gamma(A).
\]

Basically, the filter never adds new arcs then it is not possible to obtain a worse approximation of that given in input. Though the idea is quite simple, for completeness we report a formal proof.

Lemma 34. (Filter upper bound 1.) Let \(A \in A, M \subseteq \mathcal{L}, e \in \text{Expr and } n \in \mathbb{N}\); then

\[
\phi(A, M, e, n) \subseteq A.
\]

Proof. Let \(A \in A, M \subseteq \mathcal{L}, e \in \text{Expr and } n \in \mathbb{N}\).

\[
\begin{array}{c|c|c}
\text{TS} & \phi(A, M, e, n) \subseteq A \\
\text{H0} & \text{Definition 19} & \text{filter 1.}
\end{array}
\]

We proceed inductively on \(n\). For the first case we assume \(n = 0\).

2.5 Precision Limits
Now the inductive case.

\[
\begin{align*}
\text{H1} & \quad \phi(A, M, e, n) \subseteq A \quad \text{(ind. hyp.)} \\
\text{D0} & \quad \phi(A, M, e, n + 1) = \phi(A, M, e, n) \setminus \ldots \quad \text{(H0)} \\
\text{D1} & \quad \phi(A, M, e, n + 1) \subseteq \phi(A, M, e, n) \quad \text{(D0)} \\
\text{x} & \quad \phi(A, M, e, n + 1) \subseteq A \quad \text{(D1, H1)}
\end{align*}
\]

Lemma 35. (Filter upper bound 2.) Let \( A \in \mathcal{A} \), \( M \subseteq \mathcal{L} \), \( e \in \text{Expr} \); then

\[
\phi(A, M, e) \subseteq A.
\]

Proof. Let \( A \in \mathcal{A} \), \( M \subseteq \mathcal{L} \), \( e \in \text{Expr} \). Following the definition of the filter 2 (Definition 20) we consider separately two cases. For the first case let \( e = l \in \mathcal{L} \); if \( \text{eval}(A, l) \in M \) then we have \( \phi(A, M, l) = A \), otherwise \( \phi(A, M, l) = \bot \). In both the cases we have the thesis. For the second case let \( e \in \text{Expr} \setminus \mathcal{L} \).

\[
\begin{align*}
\text{TS} & \quad \phi(A, M, e) \subseteq A \\
\text{H0} & \quad \text{Definition 20, filter 2.} \\
\text{H1} & \quad \text{Lemma 34, filter upper bound 1.} \\
\text{D0} & \quad \phi(A, M, e) = \bigcap_{n \in \mathbb{N}} \phi(A, M, e, n) \quad \text{(H0)} \\
\text{D1} & \quad \forall n \in \mathbb{N} : \phi(A, M, e, n) \subseteq A \quad \text{(H1)} \\
\text{D2} & \quad \bigcap_{n \in \mathbb{N}} \phi(A, M, e, n) \subseteq A \quad \text{(D1)} \\
\text{x} & \quad \phi(A, M, e) \subseteq A \quad \text{(D2, D0)}
\end{align*}
\]

Lemma 36. (Filter upper bound 3.) Let \( A \in \mathcal{A} \) and \( c \in \text{Cond} \); then

\[
\gamma(\phi(A, c)) \subseteq \gamma(A).
\]

Proof. Let \( A \in \mathcal{A} \) and let \( c \in \text{Cond} \).

\[
\begin{align*}
\text{TS} & \quad \phi(A, c) \subseteq A \\
\text{H0} & \quad \text{Definition 21, filter 3.} \\
\text{H1} & \quad \text{Definition of concretization function.} \\
\text{H2} & \quad \text{Lemma 35, filter upper bound 2.}
\end{align*}
\]

As in the definition of the filter (Definition 21) we distinguish two cases

\[
\begin{align*}
c & = (\text{eq}, e, f); \quad \text{(C1)} \\
c & = (\text{neq}, e, f). \quad \text{(C2)}
\end{align*}
\]

For the first case (C1) we have
Now the second case (C2). If \( \#(\text{eval}(A, e) \cap \text{eval}(A, f)) \neq 1 \) from H0 we have that \( \phi(A, (\text{eq}, e, f)) = A \) then the thesis is trivially verified. Otherwise assume \( \#(\text{eval}(A, e) \cap \text{eval}(A, f)) = 1 \). Then we have

\[
\left\{ \begin{array}{l}
H3 \\
D0 \\
D1 \\
D2 \\
D3 \\
\blacklozenge
\end{array} \right.
\]

From the definition of the concretization function H1 we have that

\( \phi(A, c) \subseteq A \implies \gamma(\phi(A, c)) \subseteq \gamma(A) \),

Since we have just proved the antecedent of this implication, we have the truth of the consequent, which is the thesis.

### 2.5.3 A Final Consideration

As stated in the first few lines of this chapter, the presented model is intentionally simplified to ease the presentation and the proofs. However, these concepts can be generalized to treat more complex environments and languages. In Listing 2.7 we present an example\(^2\) that shows a more realistic implementation of the situation presented in Example 13. This example shows how using recursive data structures it is possible to generate the points-to relations presented in the previous examples: in particular loops and locations pointing to themselves, which are quite uncommon to see using only basic types.

\(^2\)This example comes from the test suite of our implementation of the algorithms.
Listing 2.7: An example of code that shows the incompleteness of the filter algorithm using a recursive data structure.
3 Extensions

With the aim of presenting a realistic points-to analysis, this chapter discusses some extensions to the simplified model previously introduced.

3.1 The Extended Abstract Memory Model

This section describes a more realistic memory model by augmenting the previously described domains with some details not directly related to the points-to problem, which are however necessary for the definition of a working memory.

3.1.1 Abstract and Concrete Locations

One of the main limitations of the formal model presented in Chapter 2 is due to the assumption that both the concrete and the abstract domains share the same set of locations $L$. Any abstract domain that aims to be practically applicable cannot rely on this assumption. From the definitions in Chapter 2 we have that for every variable created in a concrete execution there must be a distinct location in the abstract memory description. This is obviously a problem since, with the use of recursion and dynamic allocation, the number of variables created during a concrete execution can be unbounded. But also when the number of variables is known statically it is usually unfeasible to use a one-to-one approximation; consider for instance the case of arrays: under this assumption an abstract memory would be required to represent every element of an array with a distinct location. Typically, real implementations use one abstract location to approximate a set of concrete locations. For instance, a simple strategy is to approximate all the elements of an array, independently from their number, with the same abstract location. Previously we have used the symbol $L$ to denote the set of locations. From now on we denote with $L$ the set of the concrete locations and with $L^\#$ a set that we call the abstract location set. We still formalize the concrete domain as the complete lattice generated by the powerset of the total functions $L \rightarrow L$. However, we have to adapt the definition of the abstract domain as follows.

Definition 24. (Extended abstract domain.) Let $\mathcal{A}$ the support set of the abstract domain be defined as

$$\Lambda \overset{\text{def}}{=} \mathcal{C} \times \mathcal{L} \rightarrow \mathcal{L}^\#;$$
$$\mathcal{A} \overset{\text{def}}{=} \Lambda \times \mathcal{L}^\# \times \mathcal{L}^\#.$$

In words, an element $A \in \mathcal{A}$ is a pair $(f, P)$ where $f \in \Lambda$ represents the abstraction function from the concrete to the abstract locations and $P \subseteq \mathcal{L}^\# \times \mathcal{L}^\#$ is an abstract
Listing 3.1: the annotations resulting from the use of strong updates.

```
int a, b, c, d, *p;
p = &a; // EVAL(*p) = {a}
p = &b; // EVAL(*p) = {b}
p = &c; // EVAL(*p) = {c}
p = &d; // EVAL(*p) = {d}
```

points-to relation. We call abstract domain the complete lattice
\[ \langle A, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle, \]
where, for all \( \langle f, P \rangle, \langle g, Q \rangle \in A \), holds that
\[ \langle f, P \rangle \sqsubseteq \langle g, Q \rangle \overset{\text{def}}{\iff} f = g \land P \subseteq Q; \]
\[ \langle f, P \rangle \sqcap \langle g, Q \rangle \overset{\text{def}}{=} \begin{cases} \langle f, P \cap Q \rangle, & \text{if } f = g; \\ \bot, & \text{otherwise}; \end{cases} \]
\[ \langle f, P \rangle \sqcup \langle g, Q \rangle \overset{\text{def}}{=} \begin{cases} \langle f, P \cup Q \rangle, & \text{if } f = g; \\ \top, & \text{otherwise}. \end{cases} \]

and the bottom (\( \bot \)) and top (\( \top \)) elements are defined ad-hoc to satisfy the properties of the complete lattice.

Informally, given an abstract element \( \langle f, P \rangle = A \in A \), for every concrete element \( C \in C \) and every concrete location \( l \in L \), \( f(C, l) \) is the abstract location that in \( C \) abstracts \( l \).

The semantics of the abstract domain can thus be defined as follows.

**Definition 25. (Extended abstract domain semantics.)** Let \( C \in C \) and \( \langle f, P \rangle = A \in A \). We define
\[ C \in \gamma(A) \overset{\text{def}}{=} \left\{ \left( f(C, l), f(C, m) \right) \mid (l, m) \in C \right\} \subseteq P. \]

The initial definition of the concretization function (Definition 6) simply checks if all the pairs of \( C \) are also in \( A \); now, to handle the concept of abstract locations, every concrete points-to pair \( (l, m) \in C \) is abstracted, obtaining the pair \( \left( f(C, l), f(C, m) \right) \), and then we check in this “abstract pair” is in \( A \). But the distinction between concrete and abstract locations introduces a new problem in the formalization of the abstract analysis.

### 3.1.2 Weak Updates and Strong Updates

This section gives an insight of the distinction between weak and strong updates. In the literature, the term update usually means an operation that acts on a memory, concrete or abstract, modifying its state. An update can be triggered by any the of usual operations, e.g., as the assignment (Definition 14). However, the distinction between strong and weak
1 int a, b, c, d, *p;
2 p = &a; // EVAL(*p) = {a}
3 p = &b; // EVAL(*p) = {a, b}
4 p = &c; // EVAL(*p) = {a, b, c}
5 p = &d; // EVAL(*p) = {a, b, c, d}

Listing 3.2: the annotations resulting from the use of weak updates.

1 int **pp, *p1, *p2, a, b, c;
2 if (...) pp = &p1;
3 else pp = &p2;
4 p1 = &a;
5 p2 = &c;
6 *pp = &b;

Listing 3.3: an example where it is necessary to apply weak updates to obtain a safe approximation.

updates pertains only to the formalization of the abstract domain. A strong update has the effect of overwriting the previous information with new data; instead, a weak update acts by merging the original with the new data. Listings 3.1 and 3.2 present the different results of the analysis performed on the same program: in the first case using strong updates, whereas in the second case weak updates are applied. By using weak updates it is not possible to increase the precision of the approximation — each weak update yields a new abstraction that subsumes the original information. Note that in Listing 3.2, to illustrate the difference between the two options, we have forced the analysis to use weak updates. However, there are situations where the use of weak updates is necessary to obtain a safe approximation. Consider the example in Listing 3.3. The abstract execution reaches the last line with the approximation

$$\text{EVAL(*p1)} = \{a\}, \quad \text{EVAL(*p2)} = \{c\}, \quad \text{EVAL(*pp)} = \{p1, p2\}.$$ 

By applying the assignment as presented in Definition 14 we obtain the description

$$\text{EVAL(*p1)} = \{a, b\}, \quad \text{EVAL(*p2)} = \{b, c\}, \quad \text{EVAL(*pp)} = \{p1, p2\}.$$ 

In this case the abstract assignment algorithm has performed a weak update: the old values of the variables ‘p1’ and ‘p2’ are not overwritten. By forcing a strong update we
would obtain instead

\[ \text{eval}(\ast p_1) = \{b\}, \]
\[ \text{eval}(\ast p_2) = \{b\}, \]
\[ \text{eval}(\ast pp) = \{p_1, p_2\}, \]

which is clearly a wrong approximation because there exists at least a concrete execution such that, after the execution of the assignment ‘\(\ast p = \&b\)’, \(\text{eval}(\ast pp) = \{p_2\}\) holds and then \(\text{eval}(\ast p_1) = \{a\}\). Note that in the definition of the abstract assignment (Definition 14), given \((e, f) \in \text{Assignments}\), what triggers the use of a strong instead of a weak update is the fact that the lhs \(e\) evaluates to a single location:

\[
K \overset{\text{def}}{=} \begin{cases} 
\cdots, & \text{if } \# \text{eval}(A, e) = 1; \\
\emptyset, & \text{otherwise.}
\end{cases}
\]

where \(K\) denotes the set of the killed points-to pairs. The basic idea behind this approach is that when we have to update a set of more than one location it is possible that there exists a concrete memory description approximated by the current abstraction in which only one of the locations of this set will be modified while the others will retain their original value. In the above example when ‘\(pp\)’ points to ‘\(p_1\)’ then ‘\(p_2\)’ is left unchanged by the assignment ‘\(\ast pp = \&b\)’. Otherwise, when we are sure that the there is only one possible modified location we can afford that in none of the concrete memories \(C \in \gamma(\text{op}(A, \cdots))\) that location will still have the old value. However, by distinguishing between concrete and abstract locations, we are no more able to discern when a strong update can be used. Now, also when the lhs evaluates to a single location, \(\text{eval}(A, e) = \{l^2\}\), we cannot safely apply a strong update as it is possible that \(l^2\) approximates more than one concrete locations.

To overcome this problem we introduce the following definition.

**Definition 26. (Singular locations.)** Let

\[ \text{Singular} \subseteq A \times L^i \]

be defined as follows. Let \(\langle f, P \rangle = A \in A\) and \(l^2 \in L^i\). We say that the location \(l^2\) is singular in the memory abstraction \(A\) when

\[
(A, l^2) \in \text{Singular} \iff \forall C \in \gamma(A) : \# \{ l \in L \mid f(C, l) = l^2 \} \leq 1.
\]

The above definition can be read as follows. We say that an abstract location \(l^2\) is singular with respect to the abstract memory description \(A \in A\) if it does not exist any concrete memory description \(C \in \gamma(A)\) such that \(l^2\) approximates more than one of the locations of \(C\). For convenience of notation we write \(\text{Singular}(A)\) to denote the set of the singular locations of the memory \(A\), i.e,

\[
\text{Singular}(A) \overset{\text{def}}{=} \{ l^2 \mid (A, l^2) \in \text{Singular} \}
\]

3.1 The Extended Abstract Memory Model
The abstract assignment operation (Definition 14) must be adapted in order to provide a safe approximation. In particular, the definition of the kill set needs to be rewritten as

\[ E \overset{\text{def}}{=} \text{eval}(A, e); \]
\[ K \overset{\text{def}}{=} \begin{cases} E \times L & \text{if } \# E = 1 \land E \subseteq \text{Singular}(A); \\ \emptyset, & \text{otherwise}. \end{cases} \]

Also the definition of the filter operation (Definition 19) must be updated accordingly. Given \( x \in A \times \wp(L) \times \text{Expr} \) and \( i \in \mathbb{N} \); we have

\[ K \overset{\text{def}}{=} \mathcal{L} \setminus \text{Targ}(x, i); \]
\[ T \overset{\text{def}}{=} \text{Targ}(x, i + 1); \]
\[ \phi(x, i + 1) \overset{\text{def}}{=} \begin{cases} K, & \text{if } \# T = 1 \land T \subseteq \text{Singular}(A); \\ \emptyset, & \text{otherwise}. \end{cases} \]

Also the definition of the filter for the ‘neq’ operator (Definition 21) needs to be updated accordingly. Given \( A \in \mathcal{A} \) and \( e, f \in \text{Expr} \); let

\[ I \overset{\text{def}}{=} \text{eval}(A, e) \cap \text{eval}(A, f); \]
\[ E \overset{\text{def}}{=} \text{eval}(A, e) \setminus \text{eval}(A, f); \]
\[ F \overset{\text{def}}{=} \text{eval}(A, f) \setminus \text{eval}(A, e); \]

then

\[ \phi(A, (\text{neq}, e, f)) \overset{\text{def}}{=} \begin{cases} \phi(A, E, e) \cup \phi(A, F, f), & \text{if } \# I = 1 \land I \subseteq \text{Singular}(A); \\ A, & \text{otherwise}. \end{cases} \]

Finally, also the definition of the alias relation induced by a points-to abstraction must be adapted in the same way. From Definition 11, for all \( A \in \mathcal{A} \), we define \( \gamma(A) \overset{\text{def}}{=} \text{ALIAS}^{\sharp} \) as follows. For every \( e, f \in \text{expressions} \)

\[ E \overset{\text{def}}{=} \text{eval}(A, e); \]
\[ F \overset{\text{def}}{=} \text{eval}(A, f); \]
\[ \text{ALIAS}^{\sharp}(e, f) \overset{\text{def}}{=} \begin{cases} 0, & \text{if } E \cap F = \emptyset; \\ 1, & \text{if } \# E = 1 \land E = F \land E \subseteq \text{Singular}(A); \\ \top, & \text{otherwise}. \end{cases} \]

With these modified definitions, assuming for instance to approximate all the elements of an array with only one (non-singular) abstract location, the analysis applied to the code in Listing 3.4 produces the indicated annotations.

3.1 The Extended Abstract Memory Model
```c
int *p[10], *q;

int main() {
    int x, y, z;

    // EVAL(*p) = {null}
    p[...] = &x; // EVAL(*p) = {null, x}
    p[...] = &y; // EVAL(*p) = {null, x, y}
    p[...] = &x; // EVAL(*p) = {null, x, y}
    p[...] = &z; // EVAL(*p) = {null, x, y, z}

    // EVAL(*q) = {null}
    q = &x; // EVAL(*q) = {x}
    q = &y; // EVAL(*q) = {y}
    q = &x; // EVAL(*q) = {x}
    q = &z; // EVAL(*q) = {z}
}
```

Listing 3.4: this code shows the difference between singular and non-singular locations. Remember that in the C language global variables are zero-initialized. Assume that all the indices left unspecified are valid.

### 3.1.3 Notation

In the following description we use more than once the concept of *sequence*. With sequence we mean a set $S$ whose elements are enumerated, thus they can be identified and compared against their position inside the sequence. With position we mean an index ranging from 0 up to $n$ where $n + 1$ is the number of elements of $S$. For convenience of notation we write ‘$S$.size’ to denote the number of elements of the sequence $S$; we write $S_i$ or $S(i)$ to denote the element of $S$ with index $i$ and dom($S$) as an abbreviation of the set of the indices of $S$, i.e., dom($S$) = { $n \in \mathbb{N} | 0 \leq n < S$.size }. To explicitly represent the elements of the sequence we write $S = [S_0, \ldots, S_n]$. When we are not interested in the definition of any particular order among the elements of $S$, we use the concept of labelled set. A labelled set can be defined as the triple $⟨F, L, S⟩$, where $S$ is the set of the labelled elements, $L$ is a set of labels and $F: L \rightarrow S$ is a partial surjective labelling function that gives a unique name, or label, to all the elements of $S$. For convenience of notation, when $F$ and $L$ are clear from the context, we write only $S$ to refer to the labelled set $⟨F, L, S⟩$; we write $S_l$ or $S(l)$ as an abbreviation of $F(l)$ and dom($S$) as a shortcut for dom($F$). To explicitly represent the elements of $S$ we write $S = \{S_0, \ldots, S_n\}$ [1]. Note that this definition of labelled set is a generalization of the concept of sequence where $L = \mathbb{N}$ — hence the following definitions given for labelled sets can be applied also to sequences. We use also the concept of attribute. Given two labelled sets $S$ and $A$, we say that the pair $⟨S, A⟩$ is a labelled set with attributes set $A$. Again, when the attribute set $A$ is clear from the context, we write $S$ to mean the pair $⟨S, A⟩$; we say that $S$ has the attribute $X$

---

[1] The concept of position is not defined for the empty sequence.

[2] That is, at the only extent of denoting the elements of $S$, we enumerate it.

### 3.1 The Extended Abstract Memory Model
to mean that \( X \in \text{dom}(A) \) and we write ‘\( S.X \)’ as a shortcut for ‘\( A(X) \)’.

### 3.1.4 The Concept of Memory Shape

The abstract memory model that we want to describe is parametric with respect to the underlying abstract domain, e.g., the points-to domain or some numerical domain. In other words, the analysis can be seen as the coupling of a chosen abstract domain and some additional ‘structural’ information, concerning for instance the memory model of the target language/machine. With the concept of \( \text{shape} \) we want to formalize this ‘structural’ information. Recalling the definition of the extended abstract domain Definition 24, this information is needed to identify the function \( f \in \Lambda \), that is, how concrete locations are mapped to abstract locations.

**Definition 27. (Shape of a labelled set.)** Let \( \langle F, L, S \rangle \) be a given labelled set. We define the shape of \( \langle F, L, S \rangle \) as the other labelled set

\[
\text{shape}(\langle F, L, S \rangle) \triangleq \langle G, L, T \rangle,
\]

where

\[
T \triangleq \{ \text{shape}(e) \mid e \in S \},
\]

and \( G : L \rightarrow T \) is such that \( \text{dom}(G) = \text{dom}(F) \) and is defined, for all \( l \in L \), as

\[
G(l) \triangleq \text{shape}(F(l)).
\]

Now let \( \langle S, A \rangle \) be a labelled set with the attribute set \( A \). We define its shape as

\[
\text{shape}(\langle S, A \rangle) \triangleq \langle \text{shape}(S), A \rangle.
\]

Note that, as a consequence of this definition, the shape of a sequence \( S = [S_0, \cdots, S_n] \) is the sequence of the shapes

\[
\text{shape}([S_0, \cdots, S_n]) \triangleq [\text{shape}(S_0), \cdots, \text{shape}(S_n)].
\]

Note also that the \( \text{shape} \) function does not change the attributes of a labelled set.

### 3.1.5 Common Concepts

The following sections will describe the structure of both the concrete and abstract memory models. Before proceeding we need to introduce some common concepts. We refer to [BHPZ07] for a rigorous formalization of some of the ideas that we present only informally.

**Location.** The basic unit for describing the structure of the memory is the concept of *location*. Each location has a ‘type’ attribute.
Allocation. We use the concept of allocation block to describe the unit of allocation of the memory. An allocation block is a sequence of locations, it has a ‘type’ attribute and it is the base case of the inductive definition of the concept of shape (Definition 27). We define the shape of an allocation block $A$ as its ‘type’ attribute.

\[ \text{SHAPE}(A) \text{ def } A.\text{type}. \]

The type attribute of an allocation block uniquely determines the shape of the sequence of its locations; the details of this aspect will be clarified later. Informally, we say that each variable definition in the analyzed program has the effect of creating an allocation block in the memory, or, if speaking of an abstract memory, updating an already existing allocation block. In the next, when clear from the context, we call an allocation block simply allocation.

To describe the structure of the concrete stack and its abstraction we introduce these definitions.

Block. We use the term deallocation block to mean a sequence of allocations. The deallocation block is the unit for the deallocation of stack allocated memory. Ideally, the deallocation block is intended to represent the concept of block of declarations as it is defined by the C language. The order of the allocations inside a deallocation block reflects the order of creation of the variables. For conformance with the C Standard, when clear from the context we will refer to a deallocation block simply as a block. A block can also be described as the portion of the stack between two subsequent block marks [BHPZ07].

Frame. With frame we mean a sequence of blocks. In the concrete memory model, a frame can also be characterized as the portion of the stack segment between two subsequent link marks [BHPZ07]. Each link mark uniquely identifies the call statement that has generated the link mark. To identify the call statements of the program under analysis we use the concept of call site — each call statement in the program is uniquely identified by a call site. Each frame has a ‘callsite’ attribute. The value of this attribute is equal to the call site of the link mark that closes the frame — with this definition, from the program source code, the call site of a frame uniquely determines the shape of the whole frame.

Example 16. Consider the code in Listing 3.5. The frame identified by the call site 1, that corresponds to the call statement at line 8, can be described as

\[ [[\text{int } p],[\text{int } a,\text{int } b],[\text{int } c]]. \]

Instead, the frame identified by the call site 2, that corresponds to the call statement at line 12, can be described as

\[ [[\text{int } p],[\text{int } a,\text{int } b],[\text{int } d,\text{int } e]]. \]

3That is, the shape of an allocation is its ‘type’ attribute and not the sequence of the shapes of its locations.

4A reasonable choice to implement the call site concept is to use the program point associated to the call statement. However, for clarity we want to keep separate the concept of call site and program point.

3.1 The Extended Abstract Memory Model
void g();

void f(int p) {
    int a, b;
    if (...) {
        int c;
        ...
        g(); // Call site 1
    } else {
        int d, e;
        ...
        g(); // Call site 2
    }
}

Listing 3.5: the call site completely identifies the shape of the frame.

In the next we apply to the concepts just introduced the qualifiers concrete and abstract. If \( X \) is a labelled set of \( Y \) objects, then with concrete \( X \) we mean a labelled set of concrete \( Y \) objects; with abstract \( X \) we mean a labelled set of abstract \( Y \) objects. For example we call ‘abstract frame’ a sequence of abstract blocks; with ‘concrete allocation’ we mean a sequence of concrete locations. When the qualifier abstract/concrete is not specified, the context will clarify the intended one or if the statement is applicable to both cases.

3.1.6 The Concrete Memory Model

The concrete memory is organized as a labelled set of segments.

Text. The text segment is a labelled set of allocations used to represent the set of the possible targets of function pointers: basically there is one allocation for each function declared in the analyzed program. Each allocation is identified by the program point associated to the function declaration.

Heap. The heap segment is a labelled set of allocations used to represent the objects created using the functions of the ‘malloc’ family. In this segment each allocation is labelled by an address and has the attribute ‘ppoint’ (program point) that uniquely identifies the statement that has caused the allocation. Note that once fixed the analyzed program, the program point of the allocating statement identifies the type attribute of the allocation, that is, the shape of the allocation. As a consequence, given two heap allocations with the same program point attribute we know that these allocations have also the same shape.

\(^5\) In case the same function is defined multiple times, then obvious disambiguation methods are necessary; for example, as considering only the first occurrence of the declaration.

\(^6\) At this level we are not interested in the details of the addressing schema of the concrete execution model. We simply require that each heap allocation can be identified inside the segment by a tag or address.

3.1 The Extended Abstract Memory Model
Global. The *global segment* is a sequence of allocations that represent the global variables of the analyzed program. Note that the order of the allocations inside the global segment is not specified by the C Standard; thus, this detail is left to the particular execution model implemented; for instance, this order may be influenced by the particular combination of architecture/compiler chosen as target the for the analysis.

Stack frames. The *stack frames* segment is a sequence of frames. The sequence is organized such that the frame of index 0 represents the topmost frame on the stack and the frame of index $n$ —where $n + 1$ is the size of this segment— represents the oldest frame.

Stack top. The *stack top* segment represents the locations above the topmost link mark. It is a sequence of blocks (not contained in any frame), followed by a sequence of allocations (not contained in any block.).

Then we define a concrete memory $a \in \text{Mem}$ as a labelled set of the form

$$a = \{ \text{text}, \text{heap}, \text{global}, \text{stackframes}, \text{stacktop} \}.$$  

For convenience of notation we use the notation ‘$a.X$’ to refer to the $X$ segment of the memory $a$. For example, we write ‘$a.text$’ to denote the text segment of $a$. Before describing how the type attribute of a concrete allocation determines the shape of the sequence of its locations, we need to introduce some notation.

**Definition 28. (Concatenation of sequences.)** Let $A = [A_0, \cdots, A_n]$ and $B = [B_0, \cdots, B_m]$ be two sequences; then we define $A :: B$ as the concatenation of the two sequences

$$A :: B \overset{\text{def}}{=} [A_0, \cdots, A_n, B_0, \cdots, B_m].$$

**Definition 29. (Concrete allocations.)** We define the ‘ALLOC’ function by structural induction on the set of types ‘Types’. Let $t \in \text{Types}$. If $t$ is a scalar type or a function type then

\[
\text{ALLOC}(t) \overset{\text{def}}{=} [t].
\]

If $t$ is an array of $n \in \mathbb{N}$ elements of type $t_0$ then

\[
\text{ALLOC}(t) \overset{\text{def}}{=} \underbrace{\text{ALLOC}(t_0) :: \cdots :: \text{ALLOC}(t_0)}_{n+1 \text{ times}}.
\]

If $t$ is a structure type with fields: $t_0 \text{ field}_0; \cdots; t_n \text{ field}_n$; we define

\[
\text{ALLOC}(t) \overset{\text{def}}{=} \underbrace{\text{ALLOC}(t_0) :: \cdots :: \text{ALLOC}(t_n)}_{n+1 \text{ times}}.
\]
struct A {
  int a[4];
  float b;
};

struct B {
  double x;
  struct A a;
  char y;
};

Listing 3.6: the definition of an aggregate type.

Example 17. Consider Listing 3.6 then we have

\[
\text{ALLOC}(\text{int}[4]) = \{\text{int, int, int, int}\};
\]

\[
\text{ALLOC}(\text{struct A}) = \{\text{int, int, int, int, float}\};
\]

\[
\text{ALLOC}(\text{struct B}) = \{\text{double, int, int, int, int, int, float, char}\}.
\]

3.1.7 The Abstract Memory Model

Now, having introduced these basic ingredients, we can describe the organization of the abstract memory that, as the concrete memory model, is composed by different segments.

**Text.** The *text* segment is a labelled set of abstract allocations that used as targets for function pointers. The definition of the abstract text segment is the same of the concrete case: there is one text location for each function declared in the analyzed program and each location is labelled by the program point associated to the function declaration.

**Heap.** The *heap* segment is a labelled set of allocations used to abstract all the possible heap-allocated objects. Each heap allocation has as attribute the program point of the statement that has caused the allocation which is also used as label to identify the allocation inside the segment. This means that the abstract heap segment contains only one allocation for each allocating statement of the analyzed program.

**Global.** The *global* segment is a sequence of allocations that represents the global variables of the analyzed program. The order of the allocations inside the *abstract* global segment is chosen to reflect the layout of the *concrete* global segment.

---

7With *topmost frame* we mean the most recent frame on the stack, that is the frame below the topmost link mark.

8For instance, in the analysis of a complete program, the oldest frame, if present, is generated by one of the call statements contained in the `main()` function.

9For the definition of the concept of *type* we refer to the C Standard [Int99, 6.2.5.21]: *arithmetic* types and *pointer* types are collectively called *scalar* types. Array and structure types are collectively called *aggregate* types.
To represent the abstraction of the concrete stack we use three distinct segments.

**Stack top.** The *stack top* segment represents the portion of the stack above the topmost link mark. As in the concrete case, the stack top is formalized as a sequence of blocks (not contained in any frame), followed by a sequence of allocations (not contained in any block.)

**Stack head.** The *stack head* segment is a sequence of frames.

**Stack tail.** The *stack tail* segment is a labelled set of frames where each frame is labelled by its ‘call site’ attribute. This means that the stack tail contains at most one frame for each of the possible call sites of the analyzed program.

Finally, we define an abstract memory $a \in \text{Mem}^\sharp$ as a labelled set

$$a = \{ \text{text}, \text{heap}, \text{global}, \text{stacktail}, \text{stackhead}, \text{stacktop} \}.$$

As for the concrete memory, for convenience of notation we write ‘$a.X$’ to refer to the $X$ segment of the abstract memory $a$; for example we write ‘$a.\text{text}$’ to denote the text segment of the abstract memory $a$. Now we present how the type of an abstract allocation determines the shape of the sequence of its locations.

**Definition 30. (Abstract allocations.)** Let $t \in \text{Types}$. If $t$ is a scalar type or a function type, then

$$\text{ALLOC}^\sharp(t) \overset{\text{def}}{=} [t],$$

If $t$ is an array of $n \in \mathbb{N}$ elements of type $t_0$ then

$$\text{ALLOC}^\sharp(t) \overset{\text{def}}{=} \text{ALLOC}^\sharp(t_0) :: \text{ALLOC}^\sharp(t_0) :: \text{ALLOC}^\sharp(t_0).$$

If $t$ is a structure type with fields: $t_0$ field$_0$; ··· ; $t_n$ field$_n$; we define

$$\text{ALLOC}^\sharp(t) \overset{\text{def}}{=} \text{ALLOC}^\sharp(t_0) :: \cdots :: \text{ALLOC}^\sharp(t_n).$$

**Example 18.** Consider again Listing 3.6; this time we have

$$\text{ALLOC}^\sharp(\text{int}[4]) = [\text{int, int, int}; 3 \text{ times}]$$

$$\text{ALLOC}^\sharp(\text{struct A}) = [\text{int, int, int, float}];$$

$$\text{ALLOC}^\sharp(\text{struct B}) = [\text{double, int, int, int, float, char}].$$

Note that we approximate arrays using three parts. In Section 3.2 we show how these parts can be used by the analysis.

---

3.1 The Extended Abstract Memory Model
3.1.8 The Lattice Structure

As in Chapter 2, we formalize the concrete domain as the complete lattice generated by the powerset of the concrete memories \( \text{Mem} \). Our next step is to introduce the missing elements required to complete the structure of complete lattice for the abstract domain. The bottom (\( \bot \)) and the top (\( \top \)) elements are defined ad-hoc. Now we introduce the two binary operations of meet (\( \cap \)) and join (\( \cup \)) and the partial order (\( \sqsubseteq \)). In our analysis the operations of join and meet, as well as the query on the partial order, always occur between abstractions having a similar structure; these are the cases that we consider “interesting” and on which we define the operations. However, since the formalization requires total operations, we will extend the definition to “non-interesting” cases in a trivial way, that is when asked to compute the join or the meet, we will simply answer \( \top \) and \( \bot \), respectively. Note that this is a specialization of the behaviour described in Definition 24.

In this sense, when we say that two elements of \( \text{Mem}^\sharp \), say \( \langle f, P \rangle, \langle g, Q \rangle \), share a similar structure we mean that \( f = g \). To formalize the concept of similar structure we introduce the relation ‘Compatible’.

**Definition 31. (Compatibility between abstract memories.)** Let

\[ \text{Compatible} \subseteq \text{Mem}^\sharp \times \text{Mem}^\sharp \]

be defined as follows. Let \( A, B \in \text{Mem}^\sharp \); then we say that \( (A, B) \in \text{Compatible} \) when the following conditions hold:

\[
\begin{align*}
\text{shape}(A.\text{text}) &= \text{shape}(B.\text{text}); \\
\text{shape}(A.\text{heap}) &= \text{shape}(B.\text{heap}); \\
\text{shape}(A.\text{global}) &= \text{shape}(B.\text{global}); \\
\text{shape}(A.\text{stacktop}) &= \text{shape}(B.\text{stacktop}); \\
\text{shape}(A.\text{stackhead}) &= \text{shape}(B.\text{stackhead}).
\end{align*}
\]

Note that in the definition of the ‘Compatible’ relation, no constraints are specified on the shape of the stack tail segment.

**Definition 32. (Abstract domain partial order.)** Let \( A \) and \( B \) be two labelled sets.\(^{10}\) Let

\[ A \leq B \quad \text{def} \quad \text{dom}(A) \subseteq \text{dom}(B) \land \forall l \in \text{dom}(A) : A(l) \leq B(l). \]

Let \( A, B \in \text{Mem}^\sharp \). We say that

\[ A \sqsubseteq B \quad \text{def} \quad (A, B) \in \text{Compatible} \land A \leq B. \]

Note that this definition proceeds inductively on the structure of the abstract memory. The base case of this induction are locations. On locations, the definition of the partial order ‘\( \sqsubseteq \)’, of the operations ‘\( \sqcap \)’ and ‘\( \sqcup \)’, depends on the particular abstract domain adopted.

\(^{10}\)As said above this definition is valid also if \( A \) and \( B \) are sequences, as the sequence is a particular case of labelled set.

3.1 The Extended Abstract Memory Model

108
Example 19. With *location address* we mean an information that allow to identify a location inside a memory. If the abstract memory is based on a points-to domain, locations are formalized as *sets of location addresses* — a set of location addresses is used to represent the set of the possibly pointed locations. In this case, the partial order on locations is simply the relation of containment ‘⊆’ between sets of location addresses.

**Definition 33. (Abstract domain join operation.)** Let $A$ and $B$ be labelled sets. We define $A \lor B$ such that \( \text{dom}(A) \cup \text{dom}(B) = \text{dom}(C) \) and, for all $l \in L$,

\[
(A \lor B)(l) \overset{\text{def}}{=} \begin{cases} 
A(l), & \text{if } l \in \text{dom}(A) \setminus \text{dom}(B); \\
B(l), & \text{if } l \in \text{dom}(B) \setminus \text{dom}(A); \\
A(l) \lor B(l), & \text{otherwise.}
\end{cases}
\]

Let $A, B \in \text{Mem}^\sharp$. We define

\[
A \sqcup B \overset{\text{def}}{=} \begin{cases} 
A \lor B, & \text{if } (A, B) \in \text{Compatible}; \\
\top, & \text{otherwise.}
\end{cases}
\]

**Definition 34. (Abstract memory meet operation.)** Let $A$ and $B$ be labelled sets. We define $A \land B$ such that \( \text{dom}(A) \cap \text{dom}(B) = \text{dom}(C) \) and, for all $l \in L$,

\[
(A \land B)(l) \overset{\text{def}}{=} A(l) \land B(l).
\]

Let $A, B \in \text{Mem}^\sharp$. We define

\[
A \sqcap B \overset{\text{def}}{=} \begin{cases} 
A \land B, & \text{if } (A, B) \in \text{Compatible}; \\
\bot, & \text{otherwise.}
\end{cases}
\]

In the computation of the meet operation it is possible to reach the bottom on some of the locations. Depending on the position of the locations inside the abstract memory, this bottom can be propagated. If the bottom is reached on a location contained in the stack tail, then the bottom can be propagated to the frame that contains the location: this is equivalent to removing the frame from the stack tail. If the bottom location is in any other segment then the bottom can be extended to the whole memory. The reason of this will be clarified by the definition of the semantics of the abstract memory.

### 3.1.9 Concretization Function of the Abstract Memory

This section presents the concretization function for the abstract memory model Mem$^\sharp$. The definition proceed by structural induction on the definition of abstract memory. The first step is to find a mapping between the shape of the concrete memory and the shape of the abstract memory. Note that at this point we are not interested in dealing with the *value* of the memory — which is defined by the value of the locations — but

---

11Locations represents elements of the underlying abstract domain. Computing the meet between two locations, it is possible to reach the bottom of the abstract domain.
only in describing a relation about the shape. In other words, given an abstract element \( \langle f, P \rangle \in \text{Mem}^2 \) and a \( m \in \text{Mem} \), we are now trying to identify the function \( f \in \Lambda \) is defined on \( m \) (Definition 24). As already done for the definition of the operations of meet and join, we first formulate a compatibility relation to express the requirements on the structure of the concrete and abstract memories.

**Definition 35. (Compatibility between concrete and abstract memories.)** Let

\[
\text{Compatible} \subseteq \text{Mem} \times \text{Mem}^2.
\]

Let \( A \in \text{Mem}^2 \) and \( C \in \text{Mem} \); then we say that \( (C, A) \in \text{Compatible} \) when hold the following conditions

\[
\begin{align*}
\text{SHAPE}(C.text) &= \text{SHAPE}(A.text); \\
\forall l \in \text{dom}(C.heap) : C.heap(l).ppoint &\in \text{dom}(A.heap); \\
\text{SHAPE}(C.global) &= \text{SHAPE}(A.global); \\
\text{SHAPE}(A.stacktop) &= \text{SHAPE}(C.stacktop); \\
A.stackhead.size &\leq C.stackframes.size; \\
\forall i \in \{0, \cdots, A.stackhead.size - 1\} : \\
\text{SHAPE}(A.stackhead(i)) &= \text{SHAPE}(C.stackframes(i)); \\
\forall i \in \{A.stackhead.size, \cdots, C.stackframes.size - 1\} : \\
C.stackframes(i).ppoint &\in \text{dom}(A.stacktail).
\end{align*}
\]

In words, a concrete memory \( C \in \text{Mem} \) and an abstract memory \( C \in \text{Mem}^2 \) are compatible when holds the following conditions.

1. The shapes of the text segments must be the equal. From the definition, both the concrete and the abstract segment contain an abstract allocation for every declared function. Hence, as long as \( A \) and \( C \) refer to the same program, this is always true.

2. Recall that, within the concrete heap segment, allocations are identified by addresses; whereas, in the abstract heap segment, allocations are identified by program points. For the heap segment we require that to each concrete heap allocation there corresponds an abstract heap allocation identified by the program point of the concrete allocation.

3. The shapes of the global segments must be equal. From the definition of shape, this implies that the global segments must contain the same number of allocations and that each concrete allocation corresponds to an abstract allocation with the same type. Again, as long as \( A \) and \( C \) refer to the same program this property is always true.

4. The stack top segments must have the same shape; that is, the parts of the stack above the topmost link mark must have the same shape.
5. The stack frames segment of $C$ does not contain less frames than the stack head segment of $A$.

6. The shape of stack head segment of $A$ must be a prefix of the shape of the stackframes segment of $C$.

7. The remaining part of the stack frames segment of $C$ must be compatible with the stack tail segment of $A$. Recall that in the stack tail the frames are identified by their call site; thus, this means that to every frame of $C$.stackframes corresponds in $A$.stacktail a frame with the same call site.

Given a concrete memory $m \in \text{Mem}$ and an abstract memory $m^\sharp = \langle f, P \rangle \in \text{Mem}^\sharp$, once we know that the $m$ is compatible with $m^\sharp$, we ask how the locations of $m$ map onto the locations of $m^\sharp$, that is, how the function $f(m, \cdot): L \to L^\sharp$ is defined, as this is required in order to complete the definition of the semantics of the abstract domain (Definition 25). Before going into the details we introduce the idea behind the approach. By looking at the definitions of the concrete and abstract memories, note that these objects can be seen as trees — every labelled set is a node with its elements as children. If memories are trees, then we can characterize locations as the leaves. In other words, a location can be uniquely identified within a memory by the path that connects the root of the tree to the corresponding leaf node. Under these assumptions we can identify concrete location addresses as the paths inside the concrete memories and the abstract location addresses as the paths inside the abstract memories. We can now restate our initial problem as the problem of determining a mapping from paths on a concrete tree to paths on an abstract tree. To do this we exploit the recursive structure of trees — for each subtree $S^\sharp$ of $m^\sharp$ we have to identify the set of subtrees $S_0, \ldots, S_n$ of $m$ that are mapped into $S^\sharp$; being the leaves the limit case of subtrees, we will end up having a map from the leaves of $m$ to the leaves of $m^\sharp$. To formalize this mapping we use triples of the form

$$\langle S^\sharp, \{S_0, \ldots, S_n\}, M \rangle,$$

where $S^\sharp$ is a subtree of $m^\sharp$, $S_0, \ldots, S_n$ are the subtrees of $m$ mapped into $S^\sharp$ and $M$ is a map that defines how the children of $S_0, \ldots, S_n$ are mapped to the children of $S^\sharp$.

**Definition 36. (Concretization of allocations.)** We define the map function by structural induction on the set Types. Let $t \in \text{Types}$; then

$$\text{MAP}(t) \stackrel{\text{def}}{=} \begin{cases} \{\langle 0, \{0\}, \emptyset \rangle\}, & \text{if } t \text{ is scalar or function type;} \\ \{\langle 0, \{0\}, \text{MAP}(t_0)\rangle, \langle 1, \{1, \ldots, n - 1\}, \text{MAP}(t_0)\rangle, \langle 2, \{n\}, \text{MAP}(t_0)\rangle\}, & \text{if } t \text{ is an array of size } n \text{ of type } t_0; \\ \{\langle i, \{i\}, \text{MAP}(t_i)\rangle \mid i \in \{0, \ldots, n\}\}, & \text{if } t = \text{struct: } t_0 \text{ field}_0, \ldots, t_n \text{ field}_n; \end{cases}$$

### 3.1 The Extended Abstract Memory Model
Let $a_0, \ldots, a_n, a^\#$ be allocations such that

$$\forall i \in \{0, \ldots, n\} : a_i\text{.type} = a^\#.\text{type}.$$  

Then we define

$$\text{MAP}(a^\#, \{a_0, \ldots, a_n\}) \overset{\text{def}}{=} \langle a^\#, \{a_0, \ldots, a_n\}, \text{MAP}(a_{0}\.\text{type}) \rangle.$$  

Recall that allocations are the base case of the definition of \textsc{shape}: the shape of an allocation is its type attribute. This means that the above condition on the types of the allocations is equivalent to say that $a_0, \ldots, a_n, a^\#$ must have the same shape.

**Definition 37.** (Concretization of labelled sets.) Let $S_0, \ldots S_n, S^\#$ be labelled sets such that

$$\forall i \in \{0, \ldots, n\} : \text{shape}(S^\#) = \text{shape}(S_i).$$  

Then we define $\text{MAP}(S^\#, \{S_0, \ldots, S_m\})$ as the set

$$\left\langle S^\#, \{S_0, \ldots, S_m\}, \left\{ \text{MAP}(S^\#(l), \{S_0(l), \ldots, S_n(l)\}) \mid l \in \text{dom}(S_0) \right\} \right\rangle.$$  

Note that from the definition of labelled set, if $S_0, \ldots, S_n, S^\#$ have the same shape, then they have also the same domain (Definition 27); that is, the definition is well formed.

**Definition 38.** (Concretization of memories.) Let $m \in \text{Mem}$ and $m^\# \in \text{Mem}^\#$ such that

$$m = \{\text{text}, \text{heap}, \text{global}, \text{stackframes}, \text{stacktop}\},$$

$$m^\# = \{\text{text}^\#, \text{heap}^\#, \text{global}^\#, \text{stacktail}, \text{stackhead}, \text{stacktop}^\#\}.$$  

If $(m, m^\#) \in \text{Compatible}$ we define $\text{MAP}(m, m^\#)$ as the set

$$\left\{ \text{MAP}(\text{text}^\#, \{\text{text}\}), \text{MAP}(\text{global}^\#, \{\text{global}\}), \text{MAP}(\text{stacktop}^\#, \{\text{stacktop}\}) \right\}$$

$$\cup \left\{ \text{MAP}(a^\#, \{ a \in \text{heap} \mid a\.\text{ppoint} = a^\#.\text{ppoint} \}) \mid a^\# \in \text{heap}^\# \right\}$$

$$\cup \left\{ \text{MAP}(f^\#, \{ \text{stackframes}(i) \mid \text{stackframes}(i).\text{callsite} = f^\#.\text{callsite}, \right.$$

$$\left. i \in \{\text{stackhead}.\text{size}, \ldots, \text{stackframes}.\text{size} - 1\} \right\}) \mid f^\# \in \text{stacktail} \right\}$$

$$\cup \left\{ \text{MAP}(\text{stacktail}(i), \{ \text{stackframes}(i) \}) \mid i \in \{0, \ldots, \text{stackhead}.\text{size} - 1\} \right\}.$$
Note that the requirement of compatibility between the concrete memory $m$ and the abstraction $m^\#$ ensures that the function $\text{MAP}$ is well defined. Once completed the definition of the function $f \in \Lambda$, the semantics of the abstraction can be completed following the idea described in Definition 25. Alternatively, using the approach informally presented in the introduction (Section 1.2.6), the concretization function can be expressed in terms of approximation between locations, thus relying on the definition of the concretization function for the elements of the underlying abstract domain. Let $m \in \text{Mem}$ and $m^\# = \langle f, P \rangle \in \text{Mem}^\#$ and let $f \in \Lambda$ the location abstraction function of $m^\#$, then we say that $m \in \gamma(m^\#)$ when

$$\forall l \in \mathcal{L} : f(m, l) \text{ is defined } \implies m[l] \in \gamma\left(m^\# \left[f(m, l)\right]\right);$$

that for a points-to domain can also be written as

$$\forall l \in \mathcal{L} : f(m, l) \text{ is defined } \implies f(m, \text{post}(m, l)) \in \text{POST}(m^\#, f(m, l)).$$

**Singular Locations**

The definition of singular location introduced in Definition 26 is not applicable in a practical implementation as it would require to explicitly check the existence of an $m$ in the concretization of $m^\#$ with certain properties. As a consequence we need a safe approximation of the set of singular locations of an abstract memory. From the above definitions it can be easily seen that every abstract location that represents the middle part of an array of size not less that three is certainly non-singular. The same holds also for stack tail segment: each frame in this segment can represent more concrete frames; then, during the analysis we assume that all the locations contained in the stack tail are non-singular. Analogously for heap allocations; it is impossible to tell for a given allocating statement if it can be executed at most one time; in other words, it is impossible to tell if there exist a $m \in \gamma(m^\#)$ such that a given abstract heap allocation abstracts more concrete heap allocations. As a consequence, we safely assume that all abstract heap allocations are non-singular.

**3.1.10 Abstract Operations**

Thus section presents some informal considerations about the remaining operations required in order to complete the description of the execution model. We have already described the problem of formalizing operations on the memory model in Section 1.2.2: some operations are necessary to formulate the concrete execution model $\text{Mem}$; these are then generalized to the concrete domain $\varphi(\mathcal{C})$ and an approximation on $\mathcal{A}$ is provided. Consider for instance the assignment operation. Other operations are not required by the concrete execution model, but are useful for the analysis; these operations are directly formulated on the concrete domain $\varphi(\mathcal{C})$ and, as usual, an abstract counterpart is formulated on $\mathcal{A}$. Consider for instance the filter, the merge and meet operations. A more rigorous description of some of these is presented in [BHPZ07].
Notation

Before proceeding we introduce some notation. Let $A$ be a non empty sequence. We write $A = [H | T]$ to mean with $H$ the first element of $A$, also called the head element of $A$; and with $T$ the remaining part of $A$, also called the tail of the sequence $A$. We denote with ‘[]’ the empty sequence.

The Mark Operation.

This operation has the effect of closing the current block. In our memory model we have modeled the stack top segment a sequence of blocks ‘Bs’ not contained in any frame, followed by a sequence of allocations ‘As’ not contained in any block. Let $m \in \text{Mem}$ be such that

$$m.\text{stacktop} = [As, Bs].$$

Then we have

$$\text{MARK}(m) = m_0 \in \text{Mem}$$

such that

$$m_0.\text{stacktop} = [\emptyset, [As | Bs]],$$

while the rest of the memory is left unchanged. In words, the allocations ‘As’ present in the stack top segment are moved in a block at the head of the sequence of blocks ‘Bs’. The abstract mark operation is defined in the same way.

The Link Operation

This operation has the effect of creating a new frame on the stack. Let $m \in \text{Mem}$ be such that

$$m.\text{stacktop} = [\emptyset, [B | Bs]],$$

$$m.\text{stackframes} = Fs.$$

Let

$$\text{LINK}(m) = m_0 \in \text{Mem},$$

then we have

$$m_0.\text{stacktop} = [\emptyset, [B]],$$

$$m_0.\text{stackframes} = [Bs | Fs],$$

and the rest of the memory is left unchanged. The block denoted above as $B$ is intended to represent the arguments and the return value of the function call that has triggered the
link operation. To emulate the arguments passing from the callee to the called context, the allocations of the block $B$ are left in the stack top segment. The abstract operation is formulated similarly, the only difference is that the ‘stackhead’ segment is used instead of the stack frames segment. Let $m^\sharp \in \text{Mem}^\sharp$ be such that

\[ m^\sharp\text{.stacktop} = [[], [B | Bs]], \]
\[ m^\sharp\text{.stackhead} = Fs, \]

Let

\[ \text{LINK}^\sharp(m^\sharp) = m_0^\sharp \in \text{Mem}, \]

then we have

\[ m_0^\sharp\text{.stacktop} = [[], [B]], \]
\[ m_0^\sharp\text{.stackhead} = [Bs | Fs]. \]

**The New Variable Operation**

This operation is required to populate the stack. Ideally this operation can be split in two parts: first, the creation of the new allocation; second the initialization of its locations. Since the initialization can be treated a sequence of assignments, here we consider only the creation of the new locations. Let $m \in \text{Mem}$ be such that

\[ m\text{.stacktop} = [As, Bs]. \]

Let $t \in \text{Types}$ be the type of the allocated object and let

\[ \text{NEW}_s(m, t) = m_0 \in \text{Mem}. \]

We have

\[ m_0\text{.stacktop} = [[A | As], Bs], \]

where (Definition 29)

\[ A = \text{ALLOC}(t). \]

Again, the abstract operation is defined in the same way, except that the new allocation $A$ is defined as (Definition 30) $A = \text{ALLOC}^\sharp(t)$.

**The Unlink Operation**

This operation can be thought as the inverse of the link operation — if the link emulates the effects of a call statement then the unlink emulates the effects of a return statement. Let $m \in \text{Mem}$ be such that

\[ m\text{.stackframes} = [F | Fs], \]
\[ m\text{.stacktop} = [[], [B]]. \]
The block $B$ contains the arguments and the return value of the called function that are
returned to the caller context. In particular we assume that the stack top contains only
one block and that the stack frames segment contains at least one frame — in words, this
requires that every return statement must be preceded by a call statement. Let

$$\text{UNLINK}(m) = m_0 \in \text{Mem}.$$  

We have

$$m_0.\text{stackframes} = F_s,$$
$$m_0.\text{stacktop} = [[]; [B | F]],$$

while the rest of the of the memory is left unchanged. Note that the topmost frame $F$ of
the stack frames segment of $m$ has been moved in $m_0$ to the stack top segment and the
block $B$ has been appended to it. Basically, the abstract operation is defined in the same
way; the only difference is that instead of using the ‘stackframes’ segment the ‘stackhead’
segment is used.

**The Unmark Operation**

This operation can be thought as the inverse of the mark operation — if the mark
operation creates a new block gathering all the ungrouped allocations of the stack top,
then the unmark operation deletes these allocations and replaces them with the allocations
contained in the topmost block. Let $m \in \text{Mem}$ be such that

$$m.\text{stacktop} = [A_s, [B | B_s]].$$

Let

$$\text{UNMARK}(m) = m_0 \in \text{Mem}.$$  

We have

$$m.\text{stacktop} = [B, B_s],$$

while the rest of the memory is left unchanged. Note that the sequence of allocations ‘$A_s$’
has been removed and in its place we now find the allocations of the block $B$. The abstract
operation is defined in the same way. It is worth stressing that the implementation of this
abstract operation probably requires an additional step to notify the remaining locations
that the locations in ‘$A_s$’ no more exist; for instance, this is required for a pointer that
was pointing to one of the deallocated locations ($A_s$). In this case, depending on the
concrete execution model adopted, this pointer can be marked as undefined.

In our model we use the following operations to set the degree of context-sensitivity of
the analysis and to approximate recursive function calls. Both these operations have no
effects on the concrete domain, that is, for all $m \in \text{Mem}$ we have $\text{OP}(m) = m$. In terms of
the approximation this means that for all $m^\sharp \in \text{Mem}$ we have that $\gamma(m^\sharp) \subseteq \gamma(\text{OP}(m^\sharp))$.

---

3.1 The Extended Abstract Memory Model

---
The Stack Tail Push Operation

This operation has the effect of moving the oldest frame of the stack head segment (from now the ‘pushed frame’) to the stack tail. Recall that the stack tail segment is a labelled set of frames where each frame is identified by a call site and that the call site uniquely identifies the shape of the frame. This means that for each call site the stack tail can contain only one frame. Thus, if it already contains a frame with the same call site of the pushed frame then the pushed frame will be merged into the corresponding stack tail frame. Otherwise, if no frames with the same call site are already present, the frame will be simply added to the stack tail. Let $m^\sharp \in \text{Mem}^\sharp$ be such that

$$m^\sharp.\text{stackhead} = \text{Fs} :: [F],$$

$$m^\sharp.\text{stacktail} = \{F_0, \cdots, F_n\},$$

where $F$ denotes the last element of the non-empty stack head segment; ‘Fs’ denotes the remaining part of the same sequence and $n = m^\sharp.\text{stacktail}.\text{size} \in \mathbb{N}$. Let

$$\text{TAILPUSH}^\sharp(m^\sharp) = m_0^\sharp \in \text{Mem}^\sharp,$$

then we have

$$m_0^\sharp.\text{stackhead} = \text{Fs},$$

$$m_0^\sharp.\text{stacktail} = \begin{cases} 
\{F_0 \cap F, \cdots, F_n\}, & \text{if } F_0.\text{callsite} = F.\text{callsite}; \\
\{F_0, \cdots, F_n, F\}, & \text{otherwise.}
\end{cases}$$

Note that the stack tail segment is a labelled set, thus the order indicated above, $F_0, \cdots, F_n$, among its frames is completely artificial and introduced for notational convenience — writing $F_0.\text{callsite} = F.\text{callsite}$ we mean that there exists a frame in the stack tail with the same call site of $F$.

The Stack Tail Pop Operation

This operation is the inverse of the stack tail push — it moves a frame from the stack tail back into the stack head segment. To do this we have to specify which frame to restore, that is the stack tail pop operation requires a call site. Let

$$\text{TAILPOP}^\sharp : \text{Mem}^\sharp \times \text{Callsites} \rightarrow \text{Mem}^\sharp$$

Given $c \in \text{Callsites}$ and $m^\sharp \in \text{Mem}^\sharp$, if the stack tail segment of $m^\sharp$ does not contain any frame labelled $c$ then the operation results in the $\bot$ element. Otherwise let

$$m^\sharp.\text{stackhead} = \text{Fs},$$

$$m^\sharp.\text{stacktail} = \{F_0, \cdots, F_n\}$$

be such that $F_0.\text{callsite} = c$. Then calling

$$\text{TAILPOP}^\sharp(m^\sharp) = m_0^\sharp \in \text{Mem}^\sharp,$$
we have
\[ m^s_{\text{stackhead}} = \text{Fs} :: [F], \]
while the rest of the memory, also the stack tail segment, remains unchanged.

### 3.1.11 Approximating the Stack

The concept of *stack tail* is introduced precisely to handle *recursion*. In presence of recursive function calls, the number of frames on the concrete stack cannot be limited by any finite bound. Beyond these theoretical considerations, just from a practical perspective it is unfeasible to keep an arbitrary number of distinct abstract frames. The idea of our abstraction to address this problem is to represent ‘precisely’ the variables of the local environment, approximated by the *stack top segment*, and global variables, represented by the *global segment*. Also the topmost \( k \) frames of the concrete stack are abstracted ‘precisely’ by the *stack head segment*. However, we approximate more roughly in the *stack tail segment*, the content of the concrete stack below the first \( k \) frames. Frames in the stack tail are identified by their *call site*; this means that the concrete frames labelled by the same call site \( c \) that are below the \( k \)-th topmost frame, are all approximated by the same abstract frame, which is contained in the stack tail and it is identified by \( c \).

### 3.2 Pointer Arithmetic

This section presents a prototype for handling pointer arithmetic. Complex approaches to this problem are already present in the literature; for example, *string cleanness* techniques associate an integer quantity to every possible target of a pointer, to represent the distance between the beginning of the pointed object and the pointed address. These integer quantities are then approximated by the analysis using a some numerical abstraction; with the availability of *relational* numerical domains, these methods can be precise but costly [Fra07]. The method that we present now is *attribute independent* and it is completely handled by the points-to domain; the presence of an external numeric domain is assumed only to query for the value of integer expressions during the evaluation of the pointer arithmetic. Let \( m^s \in \text{Mem}^s \) and consider the expression \( p + i \) where

- the expression \( p \) is of pointer type and its abstract evaluation results in a location that is part of an array. We assume to know the type of the elements of the array and the size of the array itself.
- The expression \( i \) is of integer type and it represents the added offset.

To represent the possible errors that can arise from the concrete evaluation of the expression \( p + i \), we use the set
\[ \text{RTSErrors} \overset{\text{def}}{=} \{ E^-, E^+ \}, \]
where with ‘$E^-$’ we denote the *array underflow error* and with ‘$E^+$’ we denote the *array overflow error*. To formalize the concrete evaluation of a pointer arithmetic expressions, let

$$\text{PTRARITH}: \text{Mem} \times \text{Expr} \times \text{Expr} \rightarrow \mathcal{L} \cup \text{RTSErrors}$$

be a partial function defined for every pair of expressions $p, i \in \text{Expr}$ where $p$ is of pointer type and $i$ is of integer type. Let

$$\text{PTRARITH}: \wp(\text{Mem}) \times \text{Expr} \times \text{Expr} \rightarrow \wp(\mathcal{L} \cup \text{RTSErrors})$$

be its extension to sets of concrete memories defined, for all $M \subseteq \text{Mem}$, as

$$\text{PTRARITH}(M, p, i) \overset{\text{def}}{=} \bigcup \{ \text{PTRARITH}(m, p, i) \mid m \in M \}.$$  

A rigorous definition of $\text{PTRARITH}(m, p, i)$ would require a rigorous definition concrete execution model [BHPZ07], an informal presentation of the concrete semantics used here is later discussed in Section 3.2.2. To denote the approximation for the concrete operation $\text{PTRARITH}$ we introduce the function

$$\text{PTRARITH}: \text{Mem}^\sharp \times \text{Expr} \times \text{Expr} \rightarrow \wp(\mathcal{L}^\sharp \cup \text{RTSErrors}).$$

Generally, in an abstract memory description $m^\sharp$, the evaluation of a pointer expression results in a set of abstract locations. It is however convenient to define the abstract semantics of the $\text{PTRARITH}$ function by working on one abstract location at a time. Thus, to ease the presentation we introduce the helper function

$$\text{PTRARITH}: \text{Mem}^\sharp \times \text{Loc}^\sharp \times \text{Expr} \rightarrow \wp(\mathcal{L}^\sharp \cup \text{RTSErrors})$$

where, given $m^\sharp, l \in \text{Loc}^\sharp$ and the integer expression $i \in \text{Expr}$, $\text{PTRARITH}(m^\sharp, l, i)$ represents the set of the possible abstract locations resulting from the addition of the value of $i$ to the location $l$ in the memory $m^\sharp$. Let again $p, i \in \text{Expr}$; then we define

$$\text{PTRARITH}(m^\sharp, p, i) \overset{\text{def}}{=} \bigcup \{ \text{PTRARITH}(m^\sharp, l, i) \mid l \in \text{eval}(m^\sharp, p) \}.$$  

To query the numerical domain about the value of the integer expression $i$ we assume the existence of a function

$$\text{EVALINT}: \text{Mem}^\sharp \times \text{Expr} \rightarrow \wp(\mathbb{Z})$$

with the following semantics

$$\text{EVALINT}(m^\sharp, i) \overset{\text{def}}{=} \{ z \in \mathbb{Z} \mid \exists m \in \gamma(m^\sharp) . m[\text{eval}(m, i)] = z \}.$$  

In words, the function $\text{EVALINT}$ returns the set of the possible values that the integer expression $i$ can assume in the concrete memories $m$ approximated by $m^\sharp$.

The function $\text{PTRARITH}$ is defined as follows. We first introduce some notation. Let $S \in \mathbb{N} \setminus \{0\}$ be the size of the array on which we are performing pointer arithmetic.

---

12 We assume that the two sets $\mathcal{L}$ and RTSErrors have disjoint representations.

13 Also in this case we assume that $\mathcal{L}^\sharp$ and RTSErrors have disjoint representations.
The abstract memory model described in Section 3.1.3 approximates array variables using three distinct abstract locations here denoted with ‘H’, ‘T’ and ‘O’; we use the symbols ‘\(E^+\)’ and ‘\(E^-\)’ to denote the possible exceptional outcome of the arithmetic operation due to the exceeding of the array bounds. Let \(S \in \mathbb{N} \setminus \{0\}\) be the size of the considered array; we distinguish four possible cases: \(S = 1, S = 2, S = 3, S \geq 4\). Each of these cases is described by one of the below tables. In each of this tables, the first column contains a set of intervals of \(\mathbb{Z}\) that forms a partition of \(\mathbb{Z}\) itself. The first row of these tables represents instead the three possibility for the abstract locations \(l\) supplied to the function \(\text{ptrarith}\). Let \(D = D(S)\) be the table corresponding to the location \(l\). We denote as ‘\(D.\text{rows}\)’ the number of rows of the table \(D\). For each \(n \in \{1, \cdots, D.\text{rows}\}\) we denote as ‘\(D.\text{row} (n)\)’ the \(n\)-th row of the table \(D\). Given a row \(R\) of \(D\) we denote as ‘\(R.\text{interval}\)’ the interval of \(\mathbb{Z}\) associated to \(R\), which is located in the first column. With ‘\(R.\text{loc}(l)\)’ we denote the cell at the intersection of the row \(R\) and the column associated to the location \(l\) — the second column if \(l\) represents the head location \(H\) of the array, the third column if \(l\) represents the tail location \(T\), or the fourth column if \(l\) represents the off-by-one location \(O\). With this notation, the function \(\text{ptrarith}\) can be defined as

\[
L(S, n) \overset{\text{def}}{=} \begin{cases} 
D(S).\text{row}(n).\text{loc}(l), & \text{if } D(S).\text{row}(n).\text{interval} \cap \text{EVALINT}(m^i, i) \neq \emptyset; \\
\emptyset, & \text{otherwise};
\end{cases}
\]

\[
\text{ptrarith}(m^i, l, i) \overset{\text{def}}{=} \bigcup \{ L(S, i) \mid i \in \{1, \cdots, D.\text{rows}\} \}.
\]

Since the C language provides various mechanism to create arrays whose size is computed at run-time, we ought to consider the case of handling pointer arithmetic on arrays of unknown size\(^\text{14}\). To handle the case of arrays of unknown size we compute a merge of the above cases, obtaining the following table.

<table>
<thead>
<tr>
<th>(S &gt; 0)</th>
<th>(H)</th>
<th>(T)</th>
<th>(O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((-\infty, -2])</td>
<td>(E^-)</td>
<td>(E^-, H, T)</td>
<td>(E^-, H, T)</td>
</tr>
<tr>
<td>(-1)</td>
<td>(E^-)</td>
<td>(H, T)</td>
<td>(H, T)</td>
</tr>
<tr>
<td>(0)</td>
<td>(H)</td>
<td>(T)</td>
<td>(O)</td>
</tr>
<tr>
<td>(1)</td>
<td>(T, O)</td>
<td>(T, O)</td>
<td>(E^+)</td>
</tr>
<tr>
<td>([2, \infty))</td>
<td>(T, O, E^+)</td>
<td>(T, O, E^+)</td>
<td>(E^+)</td>
</tr>
</tbody>
</table>

\(^{14}\text{Or of partially unknown size. For example, the analysis could be able to determine some approximation of the value used to specify the size the array during its allocation.}\)
3.2.1 Examples

The following examples illustrate the described method applied to Listing 3.7. For convenience of notation we represent the steps of the computation using a table with two columns: the first column shows the program point currently executed and the second column shows the abstract value of the variable ‘first’; note indeed that the value of the pointer variable ‘last’ is never changed by the execution of the function ‘foo’. Since the ‘foo’ function contains a loop, the abstract computation terminates when a fix-point is reached; to separate the different iterations of the loop analysis we use horizontal lines. In the last row of the table we will show the result of the merge of all the exit states of the loop. For simplicity of presentation we assume that the array ‘a’ declared at line 2 contains at least four elements.

Example 20. Consider the call ‘foo(a, a + N)’. In the concrete domain the expression ‘a + N’ evaluates to the address one-past-the-end of the array ‘a’, that in the abstract domain corresponds to the off-by-one abstract location O. During all the execution of the ‘foo’ function we have eval(\(m^\#, \text{last}\)) = \{O\}. Instead, the expression ‘a’ evaluates to the address of the begin of the array ‘a’, that in the abstract domain corresponds to the head abstract location H. Thus, at the entry point of ‘foo’, the expression ‘first’ evaluates to H. These are the steps of the execution.
const unsigned int N = ...;

int a[N];

void foo(const T* first, const T* last) {
    // PP0
    while (true) {
        // PP1
        if (first == last) break;
        // PP2
        ... = *first;
        // PP3
        ++first;
    }
    // PP5
}

Listing 3.7: an example of a simple loop that depends on the pointer arithmetic computation.

<table>
<thead>
<tr>
<th>PP</th>
<th>EVAL($m^Z$, first)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$H$</td>
</tr>
<tr>
<td>1</td>
<td>$H$ (1st)</td>
</tr>
<tr>
<td>2</td>
<td>$\bot$</td>
</tr>
<tr>
<td>3</td>
<td>$H$</td>
</tr>
<tr>
<td>1</td>
<td>$T$ (2nd)</td>
</tr>
<tr>
<td>2</td>
<td>$T$</td>
</tr>
<tr>
<td>3</td>
<td>$T$</td>
</tr>
<tr>
<td>1</td>
<td>$T,O$ (3rd) Fixpoint</td>
</tr>
<tr>
<td>5</td>
<td>$O$</td>
</tr>
<tr>
<td>2</td>
<td>$T$</td>
</tr>
<tr>
<td>3</td>
<td>$T$</td>
</tr>
</tbody>
</table>

Note that the filter on the guard condition of the loop ‘first == last’ at line 8, is able to split the points-to information

\{⟨first, a.T⟩\} for the else branch –that represents the continuation of the loop– and into \{⟨first, a.O⟩\} for the then branch, that represents the execution paths that exit from the loop. In this case the analysis finds the fixpoint of the loop without signalling any error due to the pointer arithmetic; that is, it is able to prove the absence of errors in the execution of the loop.

3.2 Pointer Arithmetic
Example 21. Consider the call ‘foo(a, a)’. During all the execution we have
\[ \text{eval}(m^2, \text{last}) = \{H\}. \]
These are the steps of the execution

<table>
<thead>
<tr>
<th>PP</th>
<th>eval(m^2, first)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>H (1st) Fixpoint</td>
</tr>
<tr>
<td>5</td>
<td>H</td>
</tr>
<tr>
<td>2</td>
<td>⊥ (unreachable)</td>
</tr>
<tr>
<td>5</td>
<td>H</td>
</tr>
</tbody>
</table>

In this case the analysis is able to prove that the execution exits immediately from the loop without modifying the value of ‘first’ and without any error.

Example 22. Consider the call ‘foo(a + N, a + N)’. This case is very similar to the previous one. During the execution we have \[ \text{eval}(m^2, \text{last}) = \{O\}. \] These are the steps of the execution

<table>
<thead>
<tr>
<th>PP</th>
<th>eval(m^2, first)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>O</td>
</tr>
<tr>
<td>1</td>
<td>O (1st) Fixpoint</td>
</tr>
<tr>
<td>5</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>⊥ (unreachable)</td>
</tr>
<tr>
<td>5</td>
<td>O</td>
</tr>
</tbody>
</table>

Note that at the first iteration of the loop the filter is able to prove that ‘first’ and ‘last’ are definitely aliases. Also in this case the analysis is able to prove that the execution exits immediately from the loop without modifying the value of ‘first’ and without any error.

Example 23. Consider the call ‘foo(a + N, a)’. During the execution we have
\[ \text{eval}(m^2, \text{last}) = \{H\}. \]
These are the steps of the execution

<table>
<thead>
<tr>
<th>PP</th>
<th>eval(m^2, first)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>O</td>
</tr>
<tr>
<td>1</td>
<td>O (1st)</td>
</tr>
<tr>
<td>5</td>
<td>⊥</td>
</tr>
<tr>
<td>3</td>
<td>O (+ Dereference Warning)</td>
</tr>
<tr>
<td>1</td>
<td>E^+, ⊥ (2nd) Fixpoint</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>⊥</td>
</tr>
</tbody>
</table>
In this case the analysis is able to detect that in the first iteration of the loop at program point 3 an off-by-one location is dereferenced. Depending on the concrete execution model adopted, the analyzer may assume that the concrete execution terminates or not. In the last case the analysis is able to prove that during the next iteration of the loop, the pointer ‘first’ is incremented beyond the legal bounds of the array.

Example 24. Consider the calls \( \text{foo}(a + 4, a + 6) \), \( \text{foo}(a + 5, a + 5) \) and \( \text{foo}(a + 6, a + 4) \), which have the same abstraction. Indeed the expressions \( a + 4 \), \( a + 5 \), \( a + 6 \) —and more generally the expressions \( a + i \) with \( i \in \{1, \cdots, N-1\} \)— all evaluate in the abstract memory to the tail location \( T \) of the array \( a \). These are the steps of the execution

\[
\begin{array}{c|c}
\text{PP} & \text{EVAL}(m^2, \text{first}) \\
\hline
0 & T \\
1 & T, O \star (1st) \\
5 & T \\
2 & T \\
3 & T \\
\hline
1 & T, O \star (2nd) Fixpoint \\
5 & T \\
2 & T, O \\
3 & T (+ Dereference warning) \\
\hline
5 & T \\
\end{array}
\]

Note that \( a.T \) is not singular; thus, the filter at the guard of the loop cannot remove the arc \( \langle \text{first}, a.T \rangle \) from the else branch. Then at program point 2 we still find \{T, O\}. The above table represents the case in which the execution model forbids to dereference pointers to the off-by-one location of an array. In this case, when the abstract execution reaches program point 3 in the last iteration of the loop the analyzer filters away the off-by-one locations from the possible targets of ‘first’ and raises a warning. In this case the analysis successfully detects the possibility of an error, indeed there exist at least one concrete execution in which the off-by-one location is dereferenced. Otherwise, if the analyzer accepts as valid the dereference of the off-by-one location at line 3 we would obtain

\[
\begin{array}{c|c}
\cdots \\
3 & T, O \\
1 & T, O \star (+E^+) \\
\end{array}
\]

That is the analysis detects that the increment of ‘first’ at line 12 can produce an error due to the exceeding of the array bounds.

Note that this model is symmetrical with respect to the direction of the increasing indices — the only difference is that the off-by-one location cannot be dereferenced, while the head location \( H \) can.
3.2.2 Derivation of the Rules

This section provides the reader with a justification of the presented rules for the handling of pointer arithmetic. However, in this case the concepts are intuitive and the additional burden required to introduce a rigorous model to describe the rules does not worth the effort. Therefore, we limit the presentation to an informal justification of some of the cases with the conviction that the remaining cases can be deduced similarly. Consider the case of an array whose elements are of scalar type \( t \in \text{Types} \) which contains at least four elements, that is, \( S \geq 4 \). Under these assumptions, the concrete allocation block generated by \( t \) is

\[
\text{alloc}(t[S]) = [l_0, l_1, \cdots, l_{S-1}, l_S];
\]

where the last location of the sequence \( l_S \) represents the off-by-one location of the array.

To this concrete allocation block corresponds the following abstract allocation block

\[
\text{alloc}^\#(t[S]) = [H, T, O].
\]

In this sense we can say that

\[
\begin{align*}
\gamma(H) &= [l_0], \\
\gamma(T) &= [l_1, \cdots, l_{S-1}], \\
\gamma(O) &= [l_S].
\end{align*}
\]

Let \( p \in \text{Expr} \), \( A \in \mathcal{A} \) and \( C \in \gamma(A) \).

- Consider the case \( \text{eval}(C,p) = \{l_0\} \). Since \( C \) is approximated by \( A \) and \( \gamma(H) = [l_0] \) we have that \( H \in \text{eval}(A, p) \). In the concrete model if we move below the location \( l_0 \) we cross the boundaries of the array triggering an undefined behaviour. In the abstract model we approximate this with \( E^- \) to mean the array underflow. If we move above the location \( l_0 \) of \( n \in \mathbb{N} \) positions, with \( n \leq S \), we reach the concrete location \( l_n \). In the abstract model, staring from the head abstract location \( H \) and adding \( n \), with \( n \in [1, S-1] \), we reach the tail location \( T \); otherwise, for \( n = S \) the off-by-one location \( O \) is reached. If we move above the location \( l_0 \) of \( n \in \mathbb{N} \) positions, with \( n > S \), we trespass the boundaries of the array producing an error, that we abstract with \( E^+ \). Summing up we have, for the concrete model

<table>
<thead>
<tr>
<th>Offset</th>
<th>( l_0 + \text{Offset} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (-\infty, 0) )</td>
<td>Error: array underflow.</td>
</tr>
<tr>
<td>0</td>
<td>( l_0 )</td>
</tr>
<tr>
<td>1</td>
<td>( l_1 )</td>
</tr>
<tr>
<td>( \cdots )</td>
<td></td>
</tr>
<tr>
<td>( S-1 )</td>
<td>( l_{S-1} )</td>
</tr>
<tr>
<td>( S )</td>
<td>( l_S )</td>
</tr>
<tr>
<td>( [S+1, +\infty) )</td>
<td>Error: array overflow.</td>
</tr>
</tbody>
</table>

and its abstraction is
In case we start from the off-by-one location $l_S$, that is \textsc{eval}(C, p) = \{l_S\}$, in the abstract model we have $O \in \textsc{eval}(A, p)$. This case is quite symmetrical to the case of starting on the head location.

<table>
<thead>
<tr>
<th>Offset</th>
<th>$H + \text{Offset}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(-\infty, 0)$</td>
<td>$E^-$</td>
</tr>
<tr>
<td>0</td>
<td>$H$</td>
</tr>
<tr>
<td>$[1, S)$</td>
<td>$T$</td>
</tr>
<tr>
<td>$S$</td>
<td>$O$</td>
</tr>
<tr>
<td>$[S + 1, +\infty)$</td>
<td>$E^+$</td>
</tr>
</tbody>
</table>

and its abstraction is

<table>
<thead>
<tr>
<th>Offset</th>
<th>$l_S + \text{Offset}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(-\infty, -S)$</td>
<td>Error: array underflow.</td>
</tr>
<tr>
<td>$-S$</td>
<td>$l_0$</td>
</tr>
<tr>
<td>$1 - S$</td>
<td>$l_1$</td>
</tr>
<tr>
<td>...</td>
<td>$l_{S-1}$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$l_S$</td>
</tr>
<tr>
<td>0</td>
<td>$l_S$</td>
</tr>
<tr>
<td>$[1, +\infty)$</td>
<td>Error: array overflow.</td>
</tr>
</tbody>
</table>

- We now consider all the cases $\text{eval}(C, p) = \{l_n\}$ with $n \in [1, S - 1]$ as these cases have the same abstraction. All the concrete locations $l_1, \ldots, l_{S-1}$ are indeed abstracted by the same abstract location $T$. The difference with respect to the two previous cases is that when we perform pointer arithmetic on the tail of an array we do not know on which concrete location we are working: there is indeed a set of possible locations. This means for instance that if we move from the $l_1$ by an offset of 1 we reach $l_2$, which is still in the tail; but starting from $l_{S-1}$ we obtain $l_S$, which is in the off-by-one location $O$. From this reasoning it can be easily derived the result presented in the following tables.

<table>
<thead>
<tr>
<th>Offset</th>
<th>$O + \text{Offset}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(-\infty, -S)$</td>
<td>$E^-$</td>
</tr>
<tr>
<td>$-S$</td>
<td>$H$</td>
</tr>
<tr>
<td>$[1 - S, 0)$</td>
<td>$T$</td>
</tr>
<tr>
<td>0</td>
<td>$O$</td>
</tr>
<tr>
<td>$[1, +\infty)$</td>
<td>$E^+$</td>
</tr>
</tbody>
</table>
and its abstraction is

<table>
<thead>
<tr>
<th>Offset</th>
<th>$O + \text{Offset}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(-\infty, -S]$</td>
<td>$E^-$</td>
</tr>
<tr>
<td>$1 - S$</td>
<td>$E^-, H$</td>
</tr>
<tr>
<td>$[2 - S, -2]$</td>
<td>$E^-, H, T$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$H, T$</td>
</tr>
<tr>
<td>$0$</td>
<td>$T$</td>
</tr>
<tr>
<td>$1$</td>
<td>$T, O$</td>
</tr>
<tr>
<td>$S - 1$</td>
<td>$O, E^+$</td>
</tr>
<tr>
<td>$[S, \infty)$</td>
<td>$E^+$</td>
</tr>
</tbody>
</table>

Composing these three cases we obtain the complete table for the case $S \geq 4$ for the abstract pointer arithmetic rules.

### 3.3 Relational Operators

Just not cited above for simplicity of notation, we describe here one of the possible extensions to the filter operation that in some sense is bound to the handling of pointer arithmetic. In particular now we want to consider the use of relational operators—the ‘$\leq$', ‘$<$’ and their symmetric—and their interaction with the points-to problem. We report here the statement of the C standard about the use of relational operators between pointers [Int99 6.5.8.5]:

If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array.
with lower subscript values. […] If the expression P points to an element of an array object and the expression Q points to the last element of the same array object, the pointer expression Q+1 compares greater than P. In all other cases, the behavior is undefined.

Recalling the simplified model introduced in Chapter 2, we need to extend the set of the possible operators \{eq, neq\} to comprehend the additional operators of interest. Once augmented the set Cond with the new conditions we have to define a proper concrete semantics for the new elements. Formally, this requires the definition of a partial order on the set of location addresses. This partial order should satisfy the requirements of the C Standard reported above. Using the terminology of the extended memory model presented in Section 3.1.3 we can say that this order is required to be defined only between the locations that belong to the same allocation block. In this model we have defined the concept of allocation block as a sequence of locations and the order of the locations within the allocation in such a way to reflect the actual memory layout. Under these assumptions it is possible to define the required partial order as the order specified by the allocations; this way we are able to correctly describe the semantics of the C Standard not only for pointers to arrays but also for pointers to structure members.

Now, using the notation introduced in Chapter 2 assume to have already defined the needed strict partial order, denoted as ‘<’, on the set of locations \(< \subseteq \mathcal{L} \times \mathcal{L}\). Consider the following extension of the concrete execution model. From Definition 12 we extend the set of conditions ‘Cond’ by adding to the set of the possible operators the element ‘lt’, as to represent the ‘less-than’ operator of the C language.

\[ \text{Cond} \overset{\text{def}}{=} \{ \text{eq, neq, lt} \} \times \text{Expr} \times \text{Expr}. \]

We also need to extend Definition 13 to comprehend the newly added elements of ‘Cond’. Let \( \text{TrueCond} \subseteq \mathcal{C} \times \text{Cond} \) be extended, for all \( C \in \mathcal{C} \) and \( e, f \in \text{Expr} \), as

\[ (C, (\text{lt}, e, f)) \in \text{TrueCond} \overset{\text{def}}{=} \text{eval}(C, e) < \text{eval}(C, f). \]

Now we present a possible extension of the abstract filter operation (Definition 21) for handling the relational operator ‘lt’.

**Definition 39. (Filter on the less-than operator.)** Let

\[ \phi : \mathcal{A} \times \text{Cond} \rightarrow \mathcal{A} \]

be defined as follows. Let \( A \in \mathcal{A} \) and \( e, f \in \text{Expr} \). Let

\[ E \overset{\text{def}}{=} \{ l \in \text{eval}(A, e) \mid \exists m \in \text{eval}(A, f). l < m \}; \]
\[ F \overset{\text{def}}{=} \{ m \in \text{eval}(A, f) \mid \exists l \in \text{eval}(A, e). l < m \}; \]

For simplicity of exposition we treat explicitly only the operator ‘lt’ and we omit other relational operators whose formalization can be deduced from the formalization of ‘lt’ by symmetry and by composition with the equality operator.

3.3 Relational Operators 128
then
\[ \phi(A, (lt, e, f)) \stackrel{\text{def}}{=} \phi(A, E, e) \cap \phi(A, F, f). \]

But note that we have to consider separately the possible exceptional outcomes due to the comparison between incompatible locations — as reported above, the C Standard states that the order ‘<’ is defined only between addresses of the same object, or using our nomenclature, between locations of the same allocation block; in all other cases the behaviour is undefined. Listings \[3.8\] and \[3.9\] are two examples of the application of the filter on the relational operator ‘less-than’.

**Justification of the Definition**

Now we want to provide an intuitive description of the motivations behind the presented definition of the filter for the ‘less than’ operator. Let again \((lt, e, f) \in \text{Cond}, A \in \mathcal{A}\) and let
\[ C \in \gamma(A) \cap \text{MODELSET}((lt, e, f)) = \phi(\gamma(A), (lt, e, f)). \]
We know, from our definition of the concrete semantics of the operator ‘lt’ that
\[ C \in \text{MODELSET}((lt, e, f)) \implies \text{eval}(C, e) < \text{eval}(C, f). \]
Basically, since \(A\) is an abstraction of \(C\) then we have that the value of \(f\) in \(C\) is approximated by the value of \(f\) in \(A\). The same holds for the expression \(e\). This means that the sets \(E\) and \(F\) contain the value of \(e\) and \(f\) in \(C\), respectively, then \(C\) is also approximated by \(\phi(A, (lt, e, f))\). For completeness we also report a formal proof of the correctness of the above definition. First we prove an analogue of Lemma \[32\] for the ‘less than’ operator; then we extend the proof of Theorem \[9\] to the ‘lt’ operator.

**Lemma 37. (Less-than target.)** Let \(A \in \mathcal{A}\) and \(e, f \in \text{Expr}; \) let
\[
E \stackrel{\text{def}}{=} \{ l \in \text{eval}(A, e) \mid \exists m \in \text{eval}(A, f). l < m \}; \\
F \stackrel{\text{def}}{=} \{ m \in \text{eval}(A, f) \mid \exists l \in \text{eval}(A, e). l < m \};
\]
then, for all \(C \in \phi(\gamma(A), c)\), holds that
\[ \text{eval}(C, e) \subseteq E \land \text{eval}(C, f) \subseteq F. \]

**Proof.** Let \(c = (lt, e, f) \in \text{Cond}, A \in \mathcal{A}\) and let \(C \in \gamma(A) \cap \text{MODELSET}(c)\). Let \(E\) and \(F\) be defined as in the statement of this lemma. Recall that from the concrete semantics of the operator ‘lt’ described above we have that \(C \in \text{MODELSET}(c)\) implies that
\[ \text{eval}(C, e) < \text{eval}(C, f). \]
Then we have
Proof. (Correctness of the filter on the less-than.) Let \( A \in A \), let \( c = (lt, e, f) \in \text{Cond} \) and let \( C \in C \). Let \( E \) and \( F \) be defined as in Definition 39.

\[
\begin{array}{l}
\text{TS} \quad \text{EVAL}(C, e) \subseteq E \land \text{EVAL}(C, f) \subseteq F \\
\hline
H0 \quad C \in \gamma(A) \\
H1 \quad \text{EVAL}(C, e) = \{l_0\} \\
H2 \quad \text{EVAL}(C, f) = \{m_0\} \\
H3 \quad l_0 < m_0 \\
H4 \quad \text{Lemma 13, monotonicity of eval.} \\
H5 \quad \text{Definition 6, the concretization function.} \\
\text{D0} \quad C \subseteq A \quad \text{(H0, H5) } \\
\text{D1} \quad \text{EVAL}(C, e) \subseteq \text{EVAL}(A, e) \quad \text{(D0, H4) } \\
\text{D2} \quad \text{EVAL}(C, f) \subseteq \text{EVAL}(A, f) \quad \text{(D0, H4) } \\
\text{D3} \quad \exists l \in \text{EVAL}(A, e) \cdot l < m_0 \quad \text{(H3, H1, D1) } \\
\text{D4} \quad \exists m \in \text{EVAL}(A, f) \cdot l_0 < m \quad \text{(H3, H2, D2) } \\
\text{D5} \quad \text{EVAL}(C, e) \subseteq E \quad \text{(D4, H1) } \\
\text{D6} \quad \text{EVAL}(C, f) \subseteq F \quad \text{(D3, H2) } \\
\text{✠} \quad \text{EVAL}(C, e) \subseteq E \land \text{EVAL}(C, f) \subseteq F \quad \text{(D5, D6) } \\
\end{array}
\]

\[\square\]

Note that the structure of the proof for the correctness of the filter on the less-than operator is very similar to the structure of the proof for the equality case: actually the only difference is the definition of the target sets \( E \) and \( F \).
```c
int a[10], b[20], c[30], d[40], *p, *q;

if (...) {
    if (...) {
        if(...) { p = a; q = a + 5; }
        else { p = c; q = d + 20; }
    } else { p = b; q = b + 10; }
}

// EVAL(*q) = {a.T, b.T, d.T}

if (p < q) {
    // EVAL(*p) = {a.H, b.H}
    // EVAL(*q) = {a.T, b.T}
} else { /* Unreachable */ }

else {
    if (...) {
        if(...) { p = a + 5; q = a + 10; }
        else { p = c + 20; q = d; }
    } else { p = b + 10; q = b + 20; }
}

// EVAL(*p) = {a.T, b.T, c.T}
// EVAL(*q) = {a.O, b.O, c.H}

if (p < q) {
    // EVAL(*p) = {a.T, b.T}
    // EVAL(*q) = {a.O, b.O}
} else { /* Unreachable */ }
}

// EVAL(*q) = {a.T, a.O, b.T, b.O}

if (p < q) { /* The same */}
else { /* The same */}
```

Listing 3.8: an example of analysis of a program involving pointer arithmetic and filtering on relational operators.

3.3 Relational Operators
```c
struct T { int a, b, c; } t0, t1, t2, t3;
int *p, *q;

if (...) {
    if (...) {
        if(...) { p = &t0.a; q = &t0.b; }
        else { p = &t2.a; q = &t3.b; }
    } else { p = &t1.b; q = &t1.b; }
    /* EVAL(*p) = \{t0.a, t1.b, t2.a\} */
    /* EVAL(*q) = \{t0.b, t1.b, t3.b\} */

    if (p < q) {
        /* EVAL(*p) = \{t0.a\} */
        /* EVAL(*q) = \{t0.b\} */
    } else { /* EVAL(*p) = EVAL(*q) = \{t1.b\} */
    }
} else {
    if (...) {
        if(...) { p = &t0.b; q = &t0.c; }
        else { p = &t2.b; q = &t3.c; }
    } else { p = &t1.a; q = &t1.c; }
    /* EVAL(*p) = \{t0.b, t1.a, t2.b\} */
    /* EVAL(*q) = \{t0.c, t1.b, t3.c\} */

    if (p < q) {
        /* EVAL(*p) = \{t0.b, t1.a\} */
        /* EVAL(*q) = \{t0.c, t1.b\} */
    } else { /* Unreachable */ }
} else {
    if (...) {
        if(...) { p = &t0.a; q = &t0.b; }
        else { p = &t2.a; q = &t3.b; }
    } else { p = &t1.b; q = &t1.b; }
    /* EVAL(*p) = \{t0.a, t0.b, t1.a, t1.b\} */
    /* EVAL(*q) = \{t0.b, t0.c, t1.b\} */

    if (p < q) {
        /* EVAL(*p) = \{t0.a, t0.b, t1.a\} */
        /* EVAL(*q) = \{t0.b, t0.c, t1.b\} */
    } else { /* EVAL(*p) = EVAL(*q) = \{t0.b, t1.b\} */
    }
}
```

Listing 3.9: an example of analysis of a program involving pointer arithmetic and filtering on relational operators.

3.3 Relational Operators
3.4 Special Locations

One of the simplifications introduced in the model of Chapter 2 is that all locations are treated in the same way. In particular, in the definition of the abstract evaluation function (Definition 10) and of the assignment operation (Definition 14) there are no limitations on the locations that can be dereferenced or modified. However, a realistic memory model should provide a way to limit, on some locations, the possible operations. For instance, consider a null pointer. The C Standard specifies that dereferencing a null pointer produces an undefined behaviour. From [Int99, 6.5.3.2.4]

The unary * operator denotes indirection. […] If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined. […] Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime.

In other languages, like Java, dereferencing a null reference throws an exception. Besides of the different responses that each language exposes, it is quite common that a language has its own set of configurations that are considered exceptional and treated in an ad-hoc way. Consider for instance the case of uninitialized variables; it would be possible to formalize a concrete semantics where uninitialized variables, or pointers pointing to a deallocated memory area, cannot be evaluated and then not copied. Though this kind of conformance is uncommon in “every-day” programs, there exist application areas that require these restrictions [Mot04, Rule 9.1] [Loc05]. Note that the general idea is to capture some classes of exceptional behaviours; though the specific definition of what is exceptional can vary, also inside the same language. This section presents a possible extension of the model presented in Chapter 2 that can be used to represent the described concrete semantics. We introduce two sets of locations.

- Let NonEval ⊆ \mathcal{L} be the set of non-evaluable locations. Informally, we say that trying to evaluate a non-evaluable location results in an error.
- Let NonDeref ⊆ \mathcal{L} be the set of non-dereference-able locations. Informally, trying to apply the dereference operator to a location of this set will result in an error.

To represent the possible run-time errors we use the set

\[ \text{RTSErrors} \overset{\text{def}}{=} \{ \text{DerefError}, \text{EvalError} \}. \]

The concrete behaviour can be described by defining an extended version of the evaluation function (Definition 10). Let\(^\text{16}\)

\[ \text{eval}_{\text{c}} : \mathcal{C} \times \text{Expr} \rightarrow \mathcal{L} \cup \text{RTSErrors} \]

\(^{16}\)Here we assume that the two sets \mathcal{L} and RTSErrors have disjoint representations.
be the total function defined, for all $C \in \mathcal{C}$, $l \in \mathcal{L}$ and $e \in \text{Expr}$, as

$$\text{eval}(C, l) \overset{\text{def}}{=} \begin{cases} \text{EvalError}, & \text{if } l \in \text{NonEval;} \\ l, & \text{otherwise.} \end{cases}$$

$$\text{eval}(C, \ast e) \overset{\text{def}}{=} \begin{cases} \text{Eval}(C, e), & \text{if } \text{eval}(C, e) \in \text{RTSErrors}; \\ \text{DerefError}, & \text{if } \text{eval}(C, e) \in \text{NonDeref}; \\ \text{EvalError}, & \text{if } \text{post}(C, \text{eval}(C, e)) \in \text{NonEval}; \\ \text{post}(C, \text{eval}(C, e)), & \text{otherwise}. \end{cases}$$

Note that we have formalized the new evaluation function by tagging the exceptional paths with the elements of the set ‘RTSErrors’. An implementation of the execution model here proposed will handle these exceptional cases by signalling an error and terminating the execution, by raising an exception and modifying the execution mode or whatever else is considered appropriate. This operation can be generalized to sets as follows. For every element in the result of the concrete evaluation, we want to track the corresponding concrete memory description. Also, we want to explicitly separate the exceptional and the normal component. Let

$$\text{eval}_e : \wp(\mathcal{C}) \times \text{Expr} \rightarrow \wp(\mathcal{L} \times \mathcal{C}) \times \wp(\text{RTSErrors} \times \mathcal{C})$$

be a total function defined, for all $C \in \mathcal{C}$ and $e \in \text{Expr}$, as

$$\text{eval}_e(C, e) \overset{\text{def}}{=} \left\{ (l, D) \mid D \in C, \text{eval}(D, e) = l \in \mathcal{L} \right\},$$

$$\left\{ (x, D) \mid D \in C, \text{eval}(D, e) = x \in \text{RTSErrors} \right\}.$$

The abstract counterpart of the operation can thus be defined as

$$\text{eval}^\#_e : \wp(\mathcal{A}) \times \text{Expr} \rightarrow \wp(\mathcal{L}) \times \wp(\mathcal{A}) \times \wp(\text{RTSErrors}) \times \wp(\mathcal{A}).$$

Note that we are simplifying a little — indeed we assume to approximate elements of $\wp(\mathcal{L} \times \mathcal{C})$ with elements of the product $\wp(\mathcal{L}) \times \wp(\mathcal{A})$ and $\wp(\text{RTSErrors} \times \mathcal{C})$ with elements of $\wp(\text{RTSErrors}) \times \wp(\mathcal{A})$; this is not completely general, however is sufficient for our goals. Given $A \in \mathcal{A}$ and $e \in \text{Expr}$ we write

$$\text{eval}^\#_e(A, e) = (L, B, E, C);$$

where $L \subseteq \mathcal{L}$, $B, C \in \mathcal{A}$ and $E \in \text{RTSErrors}$ to mean that the abstract evaluation of the expression $e$ results in the set of abstract locations $L$ and the set of errors $E$; $B$ is an approximation of the abstract memory that generates $L$ and $C$ is an approximation of the abstract memory that generates $E$. The requirements for the soundness of the of the abstract operation are the following. Let, for all $A \in \mathcal{A}$ and $e \in \text{Expr}$,

$$\text{eval}^\#_e(A, e) = (L, B, E, C);$$

$$\text{eval}_e(\gamma(A), e) = (R_0, R_1);$$

3.4 Special Locations
then, to be sound, the abstract operation must satisfy the following requirements

\[
\begin{align*}
\{ l \mid (l, b) \in R_0 \} & \subseteq L; \\
\{ b \mid (l, b) \in R_0 \} & \subseteq \gamma(B); \\
\{ e \mid (e, c) \in R_1 \} & \subseteq E; \\
\{ c \mid (e, c) \in R_1 \} & \subseteq \gamma(C).
\end{align*}
\]

Let \( A \in \mathcal{A} \) and \( l \in \mathcal{L} \); then, for the base case, let

\[
\text{eval} \, ^\sharp (A, l) \overset{\text{def}}{=} \begin{cases} 
\langle \emptyset, \bot, \{\text{EvalError}\}, A \rangle, & \text{if } l \in \text{NonEval}; \\
\langle \{l\}, A, \emptyset, \bot \rangle, & \text{otherwise}.
\end{cases}
\]

For the inductive case, let \( e \in \text{Expr} \) and

\[
\text{eval} \, ^\sharp (A, e) = (L_0, A_0, E_0, B_0),
\]

\[
L_{0,X} = L_0 \cap \text{NonDeref};
\]

\[
L_{0,N} = L_0 \setminus \text{NonDeref};
\]

\[
A_{0,N} = \phi(A_0, e, L_{0,N});
\]

\[
A_{0,X} = \phi(A_0, e, L_{0,X});
\]

\[
L_1 = \text{post}(A_{0,N}, L_{0,N});
\]

\[
L_{1,X} = L_1 \cap \text{NonEval};
\]

\[
L_{1,N} = L_1 \setminus \text{NonEval};
\]

\[
A_{1,X} = \phi(A_{0,N}, *, e, L_{1,X});
\]

\[
A_{1,N} = \phi(A_{0,N}, *, e, L_{1,N});
\]

and

\[
E \overset{\text{def}}{=} E_0 \bigcup \begin{cases} 
\{ \text{DerefError}\}, & \text{if } L_{0,X} \neq \emptyset; \\
\emptyset, & \text{otherwise};
\end{cases}
\]

\[
\bigcup \begin{cases} 
\{ \text{EvalError}\}, & \text{if } L_{1,X} \neq \emptyset; \\
\emptyset, & \text{otherwise}.
\end{cases}
\]

Finally,

\[
\text{eval} \, ^\sharp (A, * e) \overset{\text{def}}{=} (L_{1,N}, A_{1,N}, E, B_0 \cup A_{0,X} \cup A_{1,X}).
\]

In words, to evaluate \( * e \) we

1. evaluate \( e \),

2. filter away the non-dereference-able locations,

\[3.4 \text{ Special Locations}\]
3. perform the actual dereference,
4. filter away the non-evaluable locations.

We can have an error if the evaluation of $e$ produces an error ($B_0$), or if we obtain a non-dereference-able location ($A_{0,X}$) or if in the last step we obtain a non-evaluable location ($A_{X,1}$). We have a location if all this steps are error free ($A_{1,N} \subseteq A_{0,N} \subseteq A_0 \subseteq A$). Note that in the computation of $A_{0,N} = \phi(A_0, e, L_{0,N})$ the filter cannot always remove all the non-dereference-able locations from the result of $\text{eval}(A_0, e)$. Note however that we compute the result of the dereference operator, $L_1 = \text{POST}(A_{0,N}, L_{0,N})$, on the set $L_{0,N}$ that by definition does not contain non-dereference-able locations.

Also the formulation of the assignment operator can have its own class of special locations. For instance it is possible to define a set of non-modifiable (read-only) locations. Finding a read-only location in the result of the evaluation of the rhs, the analysis reacts by removing that location and signalling an error. For example, in our analyzer we have introduced two special locations.

- The null location that represents the concrete ‘NULL’ address described by the C Standard. This location can be evaluated but it cannot be dereferenced nor modified, i.e, null $\in$ NonDeref, null $\notin$ NonEval and it is read-only.  

- The undefined location to be used as a target for all undefined pointers and for pointers pointing to deallocated memory. We have modelled the undefined location as a non-evaluable location. 

**Example 25.** Consider the code in Listing 3.10. From the analysis point of view, the function ‘f’ possibly returns null pointers, i.e., at line 12 we have $\text{eval}(\ast p) = \{q, \text{null}\}$. The evaluation of the expression $\ast p$ at line 21 and the evaluation of the expression $\ast\ast p$ at line 17 produces the following sequence of steps:

<table>
<thead>
<tr>
<th>i</th>
<th>$\text{EVAL}(\ast p, i)$</th>
<th>$\text{EVAL}(\ast\ast p, i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>${p}$</td>
</tr>
<tr>
<td>1</td>
<td>${p}$</td>
<td>${\text{null}, q}$</td>
</tr>
<tr>
<td>0</td>
<td>${\text{null}, q}$</td>
<td>${a, \text{DerefError}}$</td>
</tr>
</tbody>
</table>

In this case, at line 17, the analysis warns about the possibility of a dereference of a null pointer and it continues the abstract execution assuming that ‘$p$’ is not null. Instead, at line 21, the evaluation of the expression ‘$\ast p$’ as the lhs of an assignment does not raise

---

17Limiting our view to the points-to analysis, non-dereference-able locations may be seen as locations that cannot be read. On the other side, the read-only locations proposed for the assignment operation cannot be written. In this sense the value of the null location can not be read or written: the null location can only be used as target for pointers.

18Recall that the syntax of the simplified language formalized in Chapter 2 is slightly different from the syntax of the C language. Indeed we do not distinguish between expressions and lvalues, then for example, the C-expression ‘$p$’ occurring as the rhs of an assignment corresponds to ‘$\ast p$’ in our language, the C-expression ‘$\ast p$’ as the rhs of an assignment corresponds to ‘$\ast\ast p$’, while ‘$\ast p$’ as the lhs of an assignment remains the same.
int *q, a;

int** f() {
if (...)  return &q;
else      return 0;
}

q = &a;

// EVAL(*q) = {a}
int **p = f();
// EVAL(*p) = {q, null}
int **p2 = p;
// Null can be evaluated, thus copied.
// EVAL(*p2) = {q, null}
if (...) {
  ... = *p;
  // Null cannot be dereferenced.
  // EVAL(**p) = {a}
} else {
  *p = ...;
  // Null cannot be written.
  // EVAL(*p) = {q}
}

Listing 3.10: an example of dereferentiation of a null pointer.
The points-to abstraction before the evaluation of the expression ‘*p’ at line 17;

The exceptional component resulting from filtering the points-to abstraction shown above.

The normal component resulting from filtering the points-to abstraction shown above. The normal execution will continue on this refined information.

Figure 3.1: an example of analysis involving the special location null.

any error and returns the set \{null, q\}. However at this point the assignment operation detects that the program is trying to modify the null location and it triggers an error since we have modeled the null location as read-only. See Figure 3.1 for a graphical representation of this example.

**Example 26.** Consider Listing 3.11 At line 5 the points-to information is \(\text{eval}(\ast pp) = \{p, \text{undef}\}\) and the abstract evaluation of the expression ‘*pp’ produces the following sequence of steps.

\[
\begin{array}{c|c}
  i & \text{eval}(\ast pp, i) \\
  \hline
  2 & \{pp\} \\
  1 & \{p, \text{EvalError}\} \\
  0 & \{a\} \\
\end{array}
\]

In the step \(i = 1\) of the evaluation, the algorithm detects the presence of the non-evaluable location ‘undef’ and it proceeds by removing it from the result of the evaluation and by filtering the memory state against the condition (\(\text{neq}, \ast pp, \text{undef}\)). As result, the analysis is able to infer that after the execution of line 7 holds that \(\text{eval}(\ast pp) = \{p\}\). Figure 3.2 shows a graphical representation of this situation. Instead at line 11, the variable ‘pp’ is reassigned without evaluating the undefined location, then without producing any error.

---

\[19\] Again, using the formalization of the assignment presented in Section 2 the C-expression ‘pp’ occurring as the rhs of an assignment corresponds to ‘*pp’ in our formalization.
The points-to abstraction before the evaluation of the expression `pp` at line 7.

The exceptional component resulting from the filtering of the points-to abstraction shown above.

The normal component resulting from the filtering of the points-to abstraction shown above. The execution will continue on this refined information.

Figure 3.2: an example of analysis involving the special location `undefined`.

3.4 Special Locations
int **pp, *p, *q, a;

q = &a;
// EVAL(*q) = \{a\}

if (...) pp = &p;
else    pp = &q;
// EVAL(*pp) = \{q,p\}

if (...) {
    int x;
    p = &x;
    ...
} else {
    p = &a;
}
// EVAL(*p) = \{a, undef\}

... = *pp;
// Undef cannot be evaluated.
// However, the filter cannot improve the precision.

Listing 3.12: an example of the evaluation of an undefined pointer, this time due to a memory deallocation.
Example 27. Consider the example in Listing 3.12. At line 18 the points-to information $m^2$ is

\[
\begin{align*}
\text{EVAL}(\ast pp) &= \{p, q\}, \\
\text{EVAL}(\ast p) &= \{a, \text{undef}\}, \\
\text{EVAL}(\ast q) &= \{a\}.
\end{align*}
\]

At this point the abstract evaluation of the expression ‘$\ast pp$’ produces the following sequence of steps \(^{20}\)

\[
\begin{array}{l}
\text{1 EVAL}_e(\ast p, i) \\
\text{2 } \{pp\} \\
\text{1 } \{p, q\} \\
\text{0 } \{a, \text{EvalError}\}
\end{array}
\]

In the last step of the evaluation the algorithm detects the presence of the non-evaluable location ‘undef’ and it proceeds by removing this location from the result of the evaluation. However, in this case the filter is unable to divide the exceptional from the normal component, as illustrated in Figure 3.3.

The idea of filtering away the exceptional component is formalized in [CDNB08]. Removing from the abstract execution state those exceptional configurations already signalled prevents that the same error is propagated by the analysis from the first point to all the subsequent program points with the result of soiling the results of the analysis. Note that also other semantics are possible. For instance, it would be possible to model the undefined location as a non-dereference-able location instead as of a non-evaluable location. Under this assumptions uninitialized pointers and pointers pointing to deallocated memory can be evaluated and thus copied, however it is still treated as an error their dereference. In the above formalization we explicitly keep track of an approximation of the exceptional execution paths; however, in many situations this is too expensive and useless. In these cases the implementation can simply skip the collection of the exceptional states and gather only the signalled memory errors.

### 3.5 Logical Operators

The model described in Chapter 2 presents a very simplified definition of boolean condition; for example, it does not consider logical operators: and (\&\&), or (||) and the not (!). The first step necessary in order to handle these operators, is to extend the set of conditions.

**Definition 40. (Extended conditions.)** Let ‘ExtCond’ be the set defined as the language generated by the grammar

\[
e := c \mid (\text{not } e_0) \mid (e_0 \text{ or } e_1) \mid (e_0 \text{ and } e_1)
\]

where \(c \in \text{Cond}\) is an atomic condition and \(e_0, e_1 \in \text{ExtCond}\) are two extended conditions.

\(^{20}\)Again, using our formalization of the assignment operation the C-expression ‘$\ast pp$’ corresponds to $\ast \ast p$.  

3.5 Logical Operators  141
The points-to abstraction $m^\dagger$ before the evaluation of the expression ‘*pp’ at line 19:

A concrete memory description model $m_0$ of the condition
\[ c = (\text{neq, } * \text{pp, undef}) \]
approximated by $m^\dagger$.

Another concrete memory description model $m_1$ of the condition $c$ approximated by $m^\dagger$. Note however that
\[ \alpha(\{m_0, m_1\}) = m^\dagger, \]
that is the filter cannot remove any arc.

Figure 3.3: an representation of the situation of Listing 3.12.
The next step is to define the value of the new conditions.

**Definition 41. (Concrete semantics of the extended conditions.)** Let $C \in C$ and $c_0, c_1 \in \text{ExtCond}$; then

\[
C \models (\text{not } c_0) \iff C \not\models c_0;
\]

\[
C \models (c_0 \text{ and } c_1) \iff C \models c_0 \land C \models c_1;
\]

\[
C \models (c_0 \text{ or } c_1) \iff C \models c_0 \lor C \models c_1.
\]

The definition of concrete filter do not need to be updated as it is expressed in terms of the value of the conditions. Finally, we update the definition of the abstract filter as to handle the new conditions.

**Definition 42. (Extended filter.)** Let

\[
\phi : A \times \text{ExtCond} \rightarrow A \times A;
\]

\[
\phi : A \times A \times \text{ExtCond} \rightarrow A \times A;
\]

be defined, for all $A, B \in A$ and $e, f \in \text{Expr}$, as

\[
\phi(A, B, (\text{eq}, e, f)) \equiv \langle \phi(A, (\text{eq}, e, f)), \phi(B, (\text{neq}, e, f)) \rangle;
\]

\[
\phi(A, B, (\text{neq}, e, f)) \equiv \langle \phi(A, (\text{neq}, e, f)), \phi(B, (\text{eq}, e, f)) \rangle;
\]

for all $c_0, c_1 \in \text{ExtCond}$, as

\[
\phi(A, B, \text{not } c_0) \equiv \phi(B, A, c_0);
\]

\[
\phi(A, B, c_0 \text{ or } c_1) \equiv \phi\left( A, B, \text{not } ((\text{not } c_0) \text{ and } (\text{not } c_1)) \right);
\]

\[
\phi(A, B, c_0 \text{ and } c_1) \equiv \langle A_0 \cap A_1, B_0 \cup (A_0 \cap B_1) \rangle;
\]

where

\[
\phi(A, B, c_0) = (A_0, B_0);
\]

\[
\phi(A, B, c_1) = (A_1, B_1).
\]

Finally, for all $A \in A$ and $c \in \text{ExtCond}$, we define

\[
\phi(A, c) \equiv \phi(A, A, c).
\]

In this formalization of the filter operation, $\phi(A, c)$ returns a pair of abstract memories: the first component is an approximation of the states of $A$ in which the condition $c$ is true; the second is an approximation of the states of $A$ in which the condition $c$ is false. In this definition we have mentioned only the equality and inequality operator; however, it can be easily extended to comprehend relational operators (Section 3.3).

### 3.5 Logical Operators

143
Note that as shown by Section 2.5, the formulation of the filter (Definition 21) is not optimal and Example 15 shows that iterating the application of the filter it is possible to improve the precision. In Figure 3.4 we show that also having a filter that is optimal on the atomic conditions, iterating the application of the filter can improve the precision. Note indeed that on the atomic conditions ‘***p4 == &a’ and ‘***q3 == &b’, the filter operates optimally (Definition 21).
A representation of the initial points-to information.

Filtering the points-to information against the expression ‘**q3’, and the target set \{b\}. The arc (q1, a) is removed.

Filtering the points-to information against the expression ‘***q4’, and the target set \{a\}. The arc (p2, q1) is removed.

Filtering the points-to information against the expression ‘**q3’, and the target set \{b\}. The arc (q3, p2) is removed.

Filtering the points-to information against the expression ‘***q4’, and the target set \{a\}. The arc (p4, q3) is removed.

Figure 3.4: an example of that shows that iterating the filter on more conditions can improve the precision of the approximation.

3.5 Logical Operators
4 Conclusions and Future Developments

Alias analysis is an important step in the process of static analysis of programs. Compiler oriented applications are the most common clients of alias information. However, compilers stress the focus on fast analyses, whereas verifier oriented applications require precise but slower techniques. The present work, trying to address verifier needs, discusses one of the most common method used to model the aliasing problem: the points-to representation. Known results are presented within a formal model; a novel operation of filter is described and finally a formal proof of correctness of the presented method is reported.

A working prototype of the method has been implemented as part of the ‘ECLAIR’ system, which targets the analysis of mainstream languages by building upon ‘CLAIR’, the ‘Combined Language and Abstract Interpretation Resource’, which was initially developed and used in a teaching context (see http://www.cs.unipr.it/clair/).

However, many tasks have to be completed. Some of the features of the C language are still missing. One of the questions not answered is how it is possible to exploit the knowledge of the architecture/compiler target of the analysis process. For instance, the precise handling of unions and casts requires know the relative size of basic types, the alignment issues and all the details that relates to the memory layout.

The memory model described in Section 3.1.3 and implemented makes strong hypotheses about the correctness of the type information. For example, the described abstract memory does not allow to precisely track pointers of type char* resulting from casts of pointer to objects of other types. Though in the literature contains some proposal of how to avoid the necessity of relying on type informations [WL95], and how to analyze union and casts [Min06], it is unclear whether these can be applied to our situation. On the other hand, the memory model does not require any special information about the type of variables. For instance, our analysis is able ‘out of the box’ to track pointer casted and assigned to integer. Architecture-specific information is also required in order to resolve the many implementation-defined issues present in the C Standard. When the behaviour of the analyzed programs depends on these rules of the language, the analyzer, if not provided with additional information, can only warn and proceed with conservative approximation of the execution that very often in few steps degenerates to the top approximation.

Consider for instance off-by-one locations. Currently, the memory model reserves an explicit abstract location address to represent off-by-one locations only at the end of arrays; this means that scalar variables do not have a corresponding off-by-one location. Hence, the current implementation forbids pointer arithmetics on the address of a scalar object, also when the increment is equal to 1, though the C Standard allows it [Int99, 6.5.6.7]. Moreover, in the presented formulation, the handling of pointer arithmetic on arrays assumes that the off-by-one address never overlaps with another valid location, though this is allowed by the standard [Int99, 5.6.9.6].
To increase the precision of the provided alias analysis it would be possible to couple the points-to analysis with a *shape analysis* that would produce a more precise approximation recursive data structures [Deu94].

For the implementation it will be necessary to realize a complete experimental evaluation of the proposed technique in order to produce quantitative data for the comparison with other approaches.
Bibliography


[LLV] University of Illinois at Urbana-Champaign, Urbana, IL, USA. The LLVM Compiler Infrastructure. Available at http://llvm.org/.


