BDD-based Value Analysis for X86 Executables

Vom Promotionsausschuss der
Technische Universität Hamburg-Harburg
zur Erlangung des akademischen Grades
Dr. rer. nat.
genehmigte Dissertation

von
Sven Christoph Mattsen

aus
Schleswig

2017

1. Gutachter: Prof. Dr. Sibylle Schupp
2. Gutachter: Dr. Johannes Kinder

Datum der mündlichen Prüfung: 14.12.2017

Acknowledgements

First, I'd like to thank the STS institute, especially Volker Menrad and Sibylle Schupp, who have provided me with opportunities that at the time, I thought I was not worthy of. You had to suffer through me learning to write, together with the unnamed reviewers that read my articles over the years. Thanks for your selfless work!

Arne, you were and continue to be a dear friend. You made most of our time at the STS fun.

I also thank all students I was fortunate to advice with their theses and every student who ever asked a question in my exercise sessions. I wish I had done that when I was a student and try to follow your example now.

A big thanks goes to Johannes Kinder, Sibylle Schupp, and Dieter Gollmann, who reviewed my dissertation, and organized my defense.

Lastly, I thank my Family for the stability and calm they provide. Special thanks goes to my Grandmother, who's emotional support was invaluable. We miss you.

Abstract

We present an abstract domain for integer value analysis that is especially suited for the analysis of low-level code, where precision is of particular interest. Traditional value analysis domains trade precision for efficiency by approximating value sets to convex shapes. Our value sets are based on modified binary decision diagrams (BDDs), which enable size-efficient storage of integer sets. The associated transfer functions are defined on the structure of the BDDs, making them efficient even for very large sets. We provide the domain in the form of a library that we use in the implementation of a plug-in for the binary analysis framework Jakstab. The library and the plug-in are evaluated by comparison to set representations in traditional value analyses and application of the plug-in to CPU2006 benchmarks respectively.

Zusammenfassung

Wir präsentieren eine abstrakte Domäne zur Ganzzahlanalyse, die für maschinennahen Code, wo Präzision besonders wichtig ist, geeignet ist. Ganzzahlanalysen approximieren Ganzzahlmengen normalerwise durch konvexe Mengen, wobei Präzision verloren geht, aber Effizienz gewonnen wird. Unsere Ganzzahlanalyse basiert auf modifizierten binären Entscheidungsbäumen (BDD), die ein effizientes Speichern von Ganzzahlmengen ermöglichen. Die zugehörigen Transferfunktionen definieren wir auf der Struktur der BDDs, was sie selbst für sehr große Mengen effizient macht. Wir stellen die Domäne in Form einer Bibliothek bereit und nutzen diese, um ein Modul für Jakstab, eine Plattform zur Binäranalyse, zu implementieren. Die Bibliothek evaluieren wir durch Vergleich mit den Mengendarstellungen traditioneller Ganzzahlanalysen, das Modul durch Anwendung auf CPU2006-Testfälle.

Contents

Notation **1**

1. Introduction **3**

2. Analysis of Executables **7**
2.1. Example Program: C and Corresponding Assembly 8
2.2. Alternatives to Value Analysis of Binaries 10
2.3. Challenges during Value Analysis of Example Program 11

3. Value Analysis **13**
3.1. Abstract Domains for Value Analysis 13
3.2. Data Flow Analysis 16
3.3. Configurable Program Analysis 20
3.4. Lattices 24
3.5. Abstract Interpretation 25
3.6. Galois Connection 26
3.7. Heap Analysis 27
3.8. Value Analysis Tools for Binaries 28
3.9. Binary Decision Diagrams 29

4. BDDStab Library **35**
4.1. BDD Structure 35
4.1.1. Labelless Nodes 37
4.1.2. Structural Induction 38
4.1.3. Set Cardinality in $O(1)$ 39
4.2. Representing Integer Sets 43
4.3. Conversion Algorithms 44
4.3.1. Configurable BDD Construction from Singletons 44
4.3.2. BDD Creation from Standard Intervals 45
4.3.3. Approximated Set of Intervals from BDD 46
4.3.4. BDD Creation from Strided Intervals 47
4.3.5. BDD Transfer Functions from Interval Transfer Functions 50
4.4. Higher-Order Algorithm for Binary Bitwise Operations 50
4.5. Arithmetic Operations 53
4.6. Bitwise Operations on one BDD 68
4.7. BDDStab Library Implementation 74
4.7.1. The BDD Class 75
4.7.2. The CBDD Class 75
4.7.3. The IntSet Class 76
4.7.4. The IntLikeSet Class 76
4.8. Testing 76

4.9. Evaluation . . . 78
4.9.1. Measuring Procedure . . . 78
4.9.2. Representation Efficiency for Intervals . . . 79
4.9.3. Representation Efficiency for Congruence Classes . . . 80
4.9.4. Threats to Validity . . . 88

5. Jakstab **89**
5.1. Value Analysis with Jakstab . . . 89
5.2. Abstract Environment . . . 90
5.3. Intermediate Language . . . 92

6. BDDStab Jakstab Adapter **95**
6.1. Statements . . . 95
6.2. Abstract Evaluation Function . . . 96
6.3. Assume Statement . . . 99
6.4. Equivalence Class Analysis . . . 104
6.5. Widening . . . 108
6.6. CPA Configuration for BDDStab . . . 110
6.7. Implementation of BDDStab Jakstab Adapter . . . 112
6.8. Evaluation . . . 113
6.8.1. The CPU2006 Benchmark Suite . . . 113
6.8.2. Measuring Procedure . . . 113
6.8.3. Data Description . . . 114
6.8.4. Results . . . 116

7. Future Work **121**

8. Conclusion **125**

A. Auxiliary Algorithms **135**

B. BDDStab Library Example Project **137**

Notation

a	Scalar value
a_{x}	Scalar value with subscript x
$a_{\{n\}}$	Bit 2^n of bitvector a
$a_{\{n\}\,\mathrm{x}}$	Bit 2^n of bitvector a_{x}
A	Non-scalar value (possibly BDD representation)
A_{x}	Non-scalar value with subscript x (possibly BDD representation)
$A_{\{n\}}$	Non-scalar of n-bit integers ($\forall a \in A_n : 0 \leq a < 2^n$)
$A_{\{n\}\,\mathrm{x}}$	Non-scalar of n-bit integers ($\forall a \in A_n : 0 \leq a < 2^n$) named A_{x}
$(A \boxed{\circ} B)$	Decision node with true successor A and false successor B
$\boxed{1}$	Terminal true node
$\boxed{0}$	Terminal false node
$\cdot^{\#}$	Abstract version of $\cdot$
$\cdot^{\#[]}$	Interval version of $\cdot$
$\cdot^{\#s[]}$	Strided interval version of $\cdot$
$[l \,..\, h]$	Interval from l to h ($\forall a \in [l \,..\, h] : l \leq a \leq h$)
$s[l \,..\, h]$	Strided interval from l to h, with stride s ($\forall a \in [l \,..\, h] : (l \leq a \leq h \wedge \exists k : l + k * s = a)$)
$\sigma^{\#}$`.h`	The heap of the abstract state $\sigma^{\#}$.
$\sigma^{\#}$`.r`	The register table of the abstract state $\sigma^{\#}$.
$\sigma^{\#}$`.h`$[\alpha]$	Given address α, index the heap of $\sigma^{\#}$ at address α
$\sigma^{\#}$`.r`$[\alpha]$	Given register α, index the register table of $\sigma^{\#}$ at register α
$\sigma^{\#}[\alpha]$	Index the abstract state $\sigma^{\#}$. If α is an integer, then index heap, if α is a register name, then index the register table.
$\sigma^{\#}[\alpha] \coloneqq \beta$	Produce new state with storage location α set to β. All other elements of $\sigma^{\#}$ remain unchanged.
$\sqcap$	Logic $\wedge$ of two BDDs. Equivalent to intersection of two BDD-based integer sets.
$\sqcup$	Logic $\vee$ of two BDDs. Equivalent to union of two BDD-based integer sets.
$\bigsqcup$	Logic $\vee$ of a set of BDDs. Equivalent to union of set of BDD-based integer sets.
`%named`	A register with name "named".
`mov a b`	Move a value from `a` to `b`.
`cmp a b`	Compare the values `a` and `b`, and set the flags correspondingly.
`test a b`	Essentially the same as `cmp a b`.
`js t`	Jump to address `t` if the sign flag is set.
`jle t`	Jump to address `t` if `a` $\leq$ `b` in the preceding `cmp a b`.
`call t`	Call function at address `t`.
`$c`	The constant `c` in assembly.

Notation used in this Dissertation

1. Introduction

The advancement of general purpose computers and their integration into safety-critical devices such as braking systems of cars or autopilots in planes demands the identification and elimination of faulty behavior from their computers in the design phase. Since the behavior of general purpose computers is configured using programs, analyzing these programs is an important research topic in computer science.

The field of program analysis is divided in formal and non-formal methods. The advantage of formal methods is that they enable arguments about program properties with mathematical rigor. Analyses that are powerful enough to prove the absence of faults are called sound. These analyses must analyze the complete state space of a given program, which is not possible for infinite state transition systems such as the programs of general purpose computers.

The framework of abstract interpretation allows the conversion of an infinite state transition system of a program to a finite state transition system by summarizing states without losing any of the original system's behavior. The granularity of the summarized state, now called *abstract* state, determines which properties can be reasoned about using the resulting finite state transition system. The summarization itself is provided by abstract domains, which additionally provide a transition that operates on summarized states for each transition of the analyzed system. These transitions are called transfer functions.

In this dissertation, we define an abstract domain for the analysis of integer values in programs. As is common, this domain summarizes program states by using sets of values, called variation domain, where the original program used singletons. Summarization is performed by first replacing each value by a singleton set that contains the value, and then unioning all resulting sets variable per variable, a process called joining. As an example, a state that associates the variables x and y with 0 ($\{x \mapsto 0, y \mapsto 0\}$), and one that associates the variables with 1 ($\{x \mapsto 1, y \mapsto 1\}$) are first replaced by singleton set variants, and then joined to $\{x \mapsto \{0,1\}, y \mapsto \{0,1\}\}$. Generally, joining is approximative because relations between variables are not represented. In the example, the state $\{x \mapsto 0, y \mapsto 1\}$ is also included in the abstract state, but was not one of the summarized states.

A challenge in the implementation of abstract integer value domains is that they must be able to represent very large variation domains, including ones that contain all n-bit integers. Storing such large variation domains naively in a set data structure such as hash-based sets is not feasible. A common answer to this challenge is to approximate variation domains further using intervals. The interval representation of a set A is then $[\texttt{min}(A) \mathinner{..} \texttt{max}(A)]$, which clearly includes all elements from A, but may also add more elements, which deteriorates precision. Worse, the convexity constraint on intervals also causes the join operation to approximate stronger.

Another challenge is the definition of precise transfer functions that remain efficient even for very large variation domains. When variation domains are approximated using convex shapes such as intervals, defining such transfer functions is often impossible,

as convexity forces approximation. The exact transfer function of bit-level operations, such as bitwise `and` and bitwise `or`, produce non-convex sets even when applied to convex sets only. Hence, transfer functions for convex domains must restore convexity by approximation.

Altogether, we identify three main causes for approximation in integer value analyses:

1. Approximation caused by non-relational abstract states
2. Approximation due to approximative set representations
3. Approximation introduced by transfer functions

In our abstract domain, we address approximation Cause 2 by representing arbitrary integer sets using modified binary decision diagrams (BDDs), which are graph structures known to stay efficient even when representing large structures, thereby eliminating the need to approximate, e.g., by convex sets such as intervals. In difference to traditional BDDs, our BDDs are shared even if their decision node labels differ and allow $O(1)$ set cardinality. Operating on the BDD graph, and not on the integers represented by the BDD, allows our transfer function to remain efficient, even for very large sets. To address approximation Cause 3, many of our transfer functions, e.g., the ones for bitwise `and` and `or`, as well as addition and negation, are exact when operating on abstract values of unrelated variables. We provide an implementation of our abstract domain in terms of a library called BDDStab [1, 2] that is usable on the Java virtual machine. This library includes an equivalence class analysis that can be used to partially address approximation Cause 1 from outside our abstract domain itself.

We use our abstract domain library in the implementation of an integer value analysis plug-in for the binary analysis framework Jakstab [3]. The binary representation of programs is the form which general purpose computers use. It is normally not hand-written, but rather generated by programs from higher-level representations. This generation process, called compilation, generally does not preserve information that might be helpful in the understanding of programs, but is not needed for a general purpose computer to execute the program. Crucially, the structure of the compiled program, which constrains the number of feasible paths through the program, is lost and must be reconstructed by the analyzer, using integer variation domains for guidance. Furthermore, when there are several options to translate from a higher-level representation to the binary form, optimizing compilers try to choose the representation enabling the most efficient execution, which is often harder to analyze. Hence, program analysis on the binary representation presents a unique challenge for integer value analyzers, and therefore also for our domain.

In summary, we make the following contributions:

- Formulation and implementation of a modified BDD structure to represent integer sets
- Definition and implementation of algorithms for transfer functions on this BDD structure

- Definition and implementation of an integer value analysis on Jakstab's intermediate language
- An evaluation of our BDD structure and associated algorithms within its application in Jakstab and in isolation

We present our own contributions in Chapters 4 and 6. We review challenges unique to the analysis of executables in Chapter 2. We motivate the relation between integer value analysis and control flow reconstruction, as well as unique challenges for transfer functions. For these unique challenges, we provide an overview of the approaches taken in related work.

In Chapter 3, we describe traditional value analyses, based on abstract interpretation and data flow analysis. We review configurable program analysis (CPA), which is an analysis formulation used by Jakstab, allowing the implementation of data flow analysis as well as model checking, and is also suitable for the analysis of binaries. Penultimately, we give an overview of tools that use the presented techniques for the analysis of binaries. The chapter finishes with an introduction to BDDs.

Next follows Chapter 4 about our BDD-based integer sets and associated algorithms. We first present our modifications to traditional BDDs, and set up an induction scheme we use later to prove correctness of our main algorithm. A precise description of the representation of integer sets using these BDDs follows, which includes a discussion about variable ordering. Then, we introduce algorithms that convert well-known integer set approximations to BDD-based integer sets and algorithms that implement transfer functions for operators common to many program representations. The library's implementation details are presented next, and we end the chapter with an in vitro evaluation of our BDD-based integer set data structure as well as a selection of the presented algorithms.

We provide an overview of Jakstab's analysis process and its intermediate language that Jakstab uses internally in the definition of its analyses in Chapter 5. This language defines which transfer functions are needed in the implementation of our Jakstab plug-in.

We instantiate our Jakstab plug-in in Chapter 6, which requires the definition of transfer functions for statements, and an abstract evaluation function for expressions. The unique challenge of conditional statements in binary representations of programs deserves particular attention and leads us to define a specialized analysis of equivalence relations. We provide an overview of our widening operator, and instantiate a CPA instance formally. The chapter ends with an in vivo evaluation of our plug-in on CPU2006 benchmarks. We suggest future work (Chapter 7) and conclude the dissertation in Chapter 8.

2. Analysis of Executables

Executables, or binaries, are the representation of programs that is accepted for execution by general purpose computers and essentially consists of a sequence of bytes that encodes instructions of varying length. The core challenge in analyzing executables comes from the fact that they are unstructured. In structured programs, a control flow graph (CFG) can be recovered from the source code, because the language constructs of structured programs make explicit where execution may continue [4]. The key control flow construct missing or discouraged in structured programming is the `goto` statement, especially with computed target addresses. In contrast, the `goto` like jump instructions are the essential control flow constructs in executables. The absence of a CFG for executables means that without analysis, it is not known which paths may be taken through it, an information that is usually the starting point for an analysis.

Worse yet, the bytes of executables are not split into code and data. If control reaches a location in the executable, then the byte sequence starting at this location is interpreted as an instruction. The question of whether a sequence of bytes is an instruction or not is therefore equivalent to the question whether control reaches this byte sequence or not, a question that is undecidable [5], and can therefore only be answered in an approximative way.

If reachability is approximated, then it is likely that locations within the executable are falsely determined as reachable, which can lead to re-interpretation of byte sequences. Consider Figure 2.1, which uses a dynamic jump to an address stored in the register `%eax`. Therefore, approximating which program locations are reachable from the jump instruction requires an approximation of the contents of `%eax`. Assuming the register can only contain the address of the first byte in the sequence in any actual execution of the corresponding program, and the contents of `%eax` is approximated to also include the location of the second byte in the sequence, then we get two interpretations of the same byte sequence. The red interpretation is the desired interpretation, while the green one is spurious, and can lead to spurious parts in the reconstructed CFG, which in turn may deteriorate analysis precision. Control flow reconstruction (CFR) is therefore the core challenge in the analysis of executables [6].

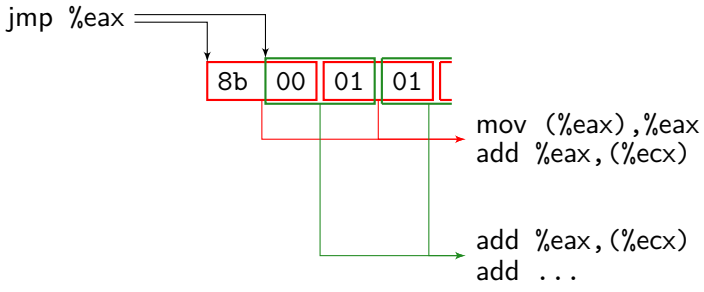


Figure 2.1.: Re-Interpretation of Bytes in the Executable

The most common and therefore relevant pattern generated by compilers that uses dynamic jumps is the jump table pattern, which consists of an indexable sequence of jump addresses, an instruction that loads an address from this table and a jump to the loaded address. C compilers frequently generate jump tables in the translation of switch statements and in the presence of function pointers. In higher-level programming languages, jump tables are used, e.g., to implement dynamic dispatch.

In the next section we first introduce an example C program and an excerpt of corresponding assembly, which we will use in Section 2.2 to review binary analysis approaches that are not based on value analysis, and then motivate some differences between value analysis of executables and structured programs, leading to additional challenges (Section 2.3).

2.1. Example Program: C and Corresponding Assembly

In the next two sections, we use the example C program in Figure 2.2, taken from the paper "ByteWeight: Learning to Recognize Functions in Binary Code" [7] that presents a technique to identify function entry points within an executable. Within the paper, the example is used to show that many analysis tools do not identify the `sum`, `sub`, and `assign` functions as reachable, and will therefore not mark them as functions in their results.

The program implements parts of a calculator that supports addition, subtraction, and assignment. It uses an array of function pointers, `funcs`, to store the addresses of the corresponding `sum`, `sub`, and `assign` functions. It then reads an integer and two strings from the user, stores them in `f`, `a`, and `b` respectively, and calls the function that is stored at position `f` in `funcs` with `a` and `b` as arguments. We changed the original program by adding lines 26 and 27, to avoid accessing the `funcs` array out of bounds, which is undefined in *C*, leaving the compiler free to generate code with unwanted behavior such as a jump to a non-deterministic value as in our example. For an analyzer, it is impossible to decide why it was unable to resolve this jump target, leading to a warning about an incomplete analysis.

Figure 2.3 shows an excerpt of an assembly representation of the C program from Figure 2.2 in GNU assembly syntax (GAS). We denote static control flow using green arrows, and dynamic control flow using blue arrows. It is important to keep in mind that our goal is not to analyze assembly, but rather to analyze binaries. However, we use assembly in this illustration, as it is unstructured as well and easier to read than a mere byte sequence. Analyzing on the machine code level is strictly harder than analyzing on assembly, as faulty re-interpretation of bytes is not possible on the assembly level.

At the addresses `02a` to `03a`, the code copies the addresses of the `sum`, `sub`, and `assign` functions to the stack, initializing the jump table. Afterwards, at address `042` it reads the integer and the two strings, and stores the integer in `%edx`, which now corresponds to `f` from the C representation. At the addresses `04b` to `057` it checks whether the read integer is a valid index into the `funcs` array. The code only reaches address `067` if the index is valid, and then calls the corresponding function using a computation involving

`%edx` at address `06e`.

```c
#include <stdio.h>
#include <string.h>
#define MAX 10

void sum(char *a, char *b) {
        printf("%s + %s = %d\n", a, b, atoi(a) + atoi(b));
}
void sub(char *a, char *b) {
        printf("%s - %s = %d\n", a, b, atoi(a) - atoi(b));
}
void assign(char *a, char *b) {
        char pre_b[MAX];
        strcpy(pre_b, b);
        strcpy(b, a);
        printf("b is changed from %s to %s\n", pre_b, b);
}

int main(int argc, char **argv) {
        void (*funcs[3])(char *x, char *y);
        int f;
        char a[MAX], b[MAX];
        funcs[0] = sum;
        funcs[1] = sub;
        funcs[2] = assign;
        scanf("%d %s %s", &f, a, b);
        if(f < 0) return 1;
        if(f > 2) return 2;
        (*funcs[f])(a, b);
        return 0;
}
```

Figure 2.2.: Indirect Jump Example in C

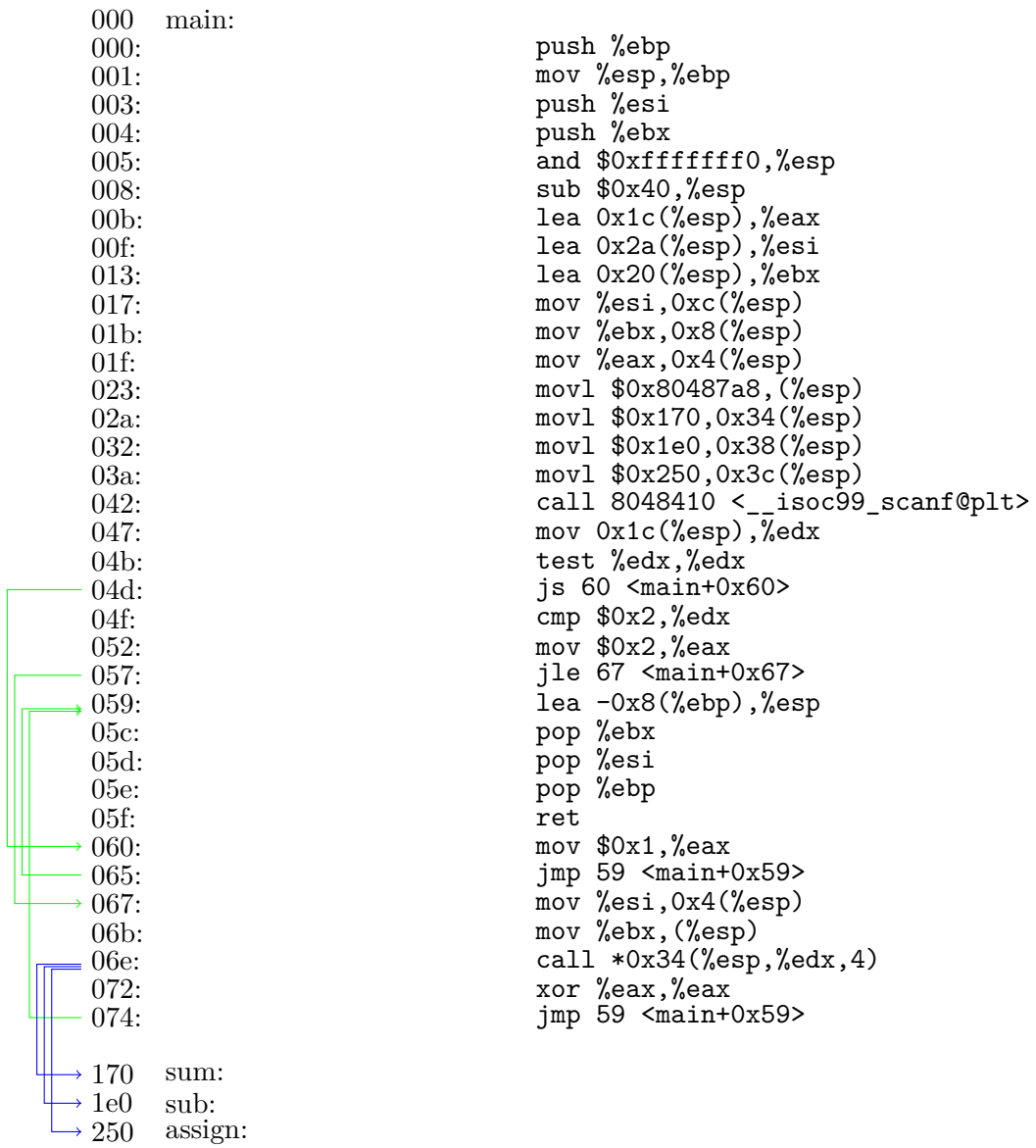


Figure 2.3.: Extract from Indirect Jump Example in GAS, with Address Offset Removed

2.2. Alternatives to Value Analysis of Binaries

Without value analysis, other means to determine the successors of each instruction must be found. Disassemblers [8, 9, 10] convert programs from their binary representation to a representation in assembly language that usually also contains some control flow information such as function entry points. Generally, disassemblers do not guarantee that their result will include all instructions that are in the binary, nor do they guarantee that

the instructions they found are used in the binary. The simplest class of disassemblers use the linear sweep method to determine the successor of each disassembled instruction. Whenever a linear sweep disassembler analyzed an n-byte instruction at address a, its heuristics is to assume that the successor instruction is found at position $a + n$, i.e., directly after the disassembled instruction. Because linear sweep disassemblers do not interpret any control flow instructions, they will interpret data, embedded in the executable, as instructions, which leads to false disassembly if the end of the data section is not aligned with its interpretation as code. Another, more capable class of disassemblers uses the recursive descent strategy [8]. Prominent examples of this class are IdaPro [10] and radare2 [9]. With recursive descent, the analysis of a program location yields not only a disassembled instruction, but also an approximated set of successor locations. This set of successor locations may be determined by sophisticated program heuristics that make use of the systematic way compilers generate code to resolve dynamic jumps. However, since our example implements jump tables in a non-standard way, i.e., via function pointers, it is unlikely that these heuristics will identify the entry points of `sum`, `sub`, and `assign`. Because the heuristics used by recursive descent disassemblers are known, it is possible to craft an executable that hides malicious behavior from disassemblers, making its disassembly appear harmless [11].

2.3. Challenges during Value Analysis of Example Program

In this section, we discuss aspects of the program from Figure 2.3 when using value analysis to determine control flow. We will focus on the following three challenging aspects:

1. Dynamically computed target of call instruction (Address `06e`)
2. Non-deterministic functions (Address `042`)
3. Conditional jumps with interleaved code (addresses `04b` to `057`)

The challenge in Aspect 1 is to compute a variation domain (VD) for the target expression, i.e., `*0x34(%esp,%edx,4)`, which corresponds to the value stored under the address given by `%edx` $*$ `4` + `%esp` + `0x34`. Computing a VD for this expression not only requires a VD of `%edx` and `%esp`, but also the ability to compute a VD for the multiplication of a VD by a constant and the addition of two VDs. The resulting VD must then be used to retrieve the call target from the jump table, which requires a model of the heap. Since the VD of the call target describes the target addresses, spurious elements in the VD may lead to spurious edges and nodes in the computed CFG, which reduces the analyses' precision. Should the spurious CFG edges and nodes lead to loops, then it is possible that the precision loss applies also to the original VD, which in turn may worsen the precision of the CFG. A value analyzer for binaries should therefore focus on precision even at reduced analysis efficiency.

Aspect 2 is a challenge, because the `scanf` function retrieves input from the user. During static analysis of the binary, the result of `scanf` is therefore non-deterministic,

and the analyzer must represent that with a special VD that contains all values of the corresponding type, e.g., all possible n-bit integers. Some analyzers only allow VDs up to a maximum size, and use a symbolic value to represent a VD that includes all values, but as we will see in the discussion of Aspect 3, this is not always sufficient as conditional branches may restrict VDs, resulting in very large VDs.

Aspect 3 is challenging because a value analyzer must be able to restrict an abstract state according to an analyzed conditional branch. In the example, the integer, retrieved from the user by the `scanf` function, is non-deterministic. Therefore, the VD of `%edx` must contain all integer values at address `04b`. The `test` instruction sets the sign flag when `%edx` is less than zero. If so, the `js` instruction jumps to address `60`, otherwise execution continues at `04f`. A precise value analyzer would compute two abstract values for `%edx`, one for address `60` containing all negative integers, and one for address `04f` containing all non-negative integers. This restriction is challenging because the `js` instruction does not directly refer to `%edx`. Therefore, `test` or `cmp` instructions must either be analyzed together with the corresponding jump, or the relation between each flag and the computation of its value must be kept. RREIL [12], an intermediate language for binary analysis, takes the first approach using instructions that are essentially the combination of `test` or `cmp` and the corresponding conditional jump. When the compiler interleaves instructions between the comparing instruction and the corresponding jump, as it has done in our example at addresses `04f` to `057`, then it must be ensured that the interleaved instructions do not change the value of the flags used by the jump. Jakstab [3] implements the second approach, by storing the value of flags symbolically.

3. Value Analysis

In this chapter, we provide an overview of value analyses, which statically approximate the set of values each variable may have in an analyzed program. We first categorize the approximation behavior of known value analyses (Section 3.1) and then review analysis methods and algorithms to implement value analyses (Sections 3.2 and 3.3). Afterwards, we give a short introduction to abstract interpretation (Sections 3.5 and 3.6). Section 3.7 briefly reviews approaches to model the heap of programs during value analysis and Section 3.8 gives an overview of existing binary analysis tools. We finish with an introduction to BDDs (Section 3.9), which we later extend for our value analysis domain (Chapter 4).

Value analyses approximate the infinite state transition system of Turing-complete programs by summarizing states to representative versions. These representative versions are constructed by systematically replacing the program's values with approximative abstract versions, which themselves represent collections of original values. This approximation is necessary as exact analysis of infinite state systems is generally not possible [5, 13]. Which abstract representations are admissible is defined by the analyses' abstract domain. Specialized versions are available for the analysis of specific kinds of values, e.g., Booleans [14], integers [2], and floating point numbers [15]. Their area of application depends on the type of approximation used in the abstraction. Value analyses that can prove that a variable at a specific program location cannot have certain values are called sound, otherwise they are called unsound. Unsound value analyses are used for bug finding, where proving the absence of bugs is not required, or for disassembly that does not have to identify all instructions [16]. Because these analyses are unsound, they do not have to cover the whole state space of the analyzed program, and instead use a higher precision for the covered state space. Sound value analyzers, on the other hand, are able to prove that programs do not reach states that are defined as erroneous or unwanted. They are therefore used to provide assurance in safety-critical areas of applications [17, 18]. In this dissertation, we focus on sound integer value analysis, and provide an overview of contemporary abstract domains next.

3.1. Abstract Domains for Value Analysis

An abstract domain determines the kind of abstraction performed in a value analysis. It consists of abstraction and concretization functions, which convert between the kind of values used by the original program and the abstract representation, a set of admissible abstract values. Furthermore, abstract domains must supply a transfer function, which performs computations corresponding to program instructions on a given abstract state. We discuss the requirements for soundness on the components of an abstract domain in Section 3.5.

One of the best-known abstract domains in value analysis is the interval abstract domain [19], which we use to illustrate how the individual parts of an abstract domain are used. Abstraction, i.e., summarizing a set of program states to one abstract state,

is done independently per variable, by assigning each variable in the abstract state to an interval that covers all of the variable's values from the original program states. The concretization, i.e., creating the set of represented states from an abstract state, must return the largest set of states of which the abstraction yields the original abstract state.

Consider the following set of states S, which assign x and y to values between 0 and 4:

$$S = \{(x \mapsto 0, y \mapsto 0), (x \mapsto 4, y \mapsto 4), (x \mapsto 4, y \mapsto 0), (x \mapsto 3, y \mapsto 0)\}$$

Figure 3.1 shows a visual representation of the four original states, using black colored dots, and their interval abstraction using the rectangle. The corresponding concretization includes all states within the rectangle, which, assuming integer value analysis, are visualized by the intersections of the dotted lines. In this example, the interval abstract domain therefore overapproximates S by 21 states. We can identify two causes for the approximation of the interval domain in our example:

1. Convexity requirement
2. Loss of relational information

Cause 1 comes from the fact that intervals can only represent convex sets precisely, i.e., all values between two included values must be included as well. Abstract domains supporting only convex abstractions are aptly called convex domains. The main advantage of convexity is that it simplifies the implementation of transfer functions, as they can usually be formulated exclusively on the borders.

Cause 2 comes from abstracting each variable independently, which in this case means that the abstract state does not include the fact that $y \leq x$ for all states in S. An abstract domain with this kind of per variable abstraction is called non-relational and all represented elements are included in hyper rectangles.

There also exist well-known relational convex abstract domains. This kind of domain allows the representation of S using the triangle in Figure 3.1, which causes an overapproximation of 11 states, instead of the 21 for intervals. One of the earliest of such domains is the polyhedra abstract domain [20, 21], which represents convex shapes in n-dimensional space, where each dimension corresponds to one variable. To improve efficiency with moderate precision loss, several abstract domains have been devised that restrict the n-dimensional shape further [15, 22].

Non-convex, non-relational abstract domains such as the domain of strided intervals [23, 24], which combines intervals with congruence information [25], retain the property that their representations are enclosed using hyper rectangles. However, they do not enforce that all states framed by the border must be included. In strided intervals, all included elements must be equi-spaced, meaning the holes between any two included elements must have the same size. This restriction applies with different hole sizes in each dimension. In our example, the y components of all states in S fulfill $y\%4 = 0$, and can therefore be represented without approximation. However, the x components of these states have irregular holes between them and their representation can therefore

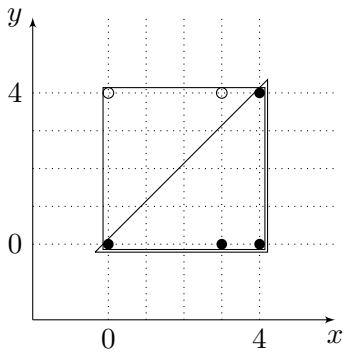


Figure 3.1.: Abstraction of States using the Interval Domain

not be improved using congruence information. Therefore, the strided interval domain can approximate to an abstract state that represents all states on the lower and upper horizontal of the rectangle, which is an overapproximation by only 6 elements.

A perfect abstract domain would be non-convex and relational, and would neither restrict the shape of the border of abstract states, nor the type of non-convexity. This type of domain corresponds to what is used in classic model checking [26], which is applicable to finite state systems. Nevertheless, symbolic model checking [27] uses BDDs to store the state space in a similar way to our domain, without abstracting each variable individually. Non-convex, relational analyses that are used in infinite systems are often implemented using limited disjunctive refinement, i.e., the representation of states using up to k abstract states [18, 28] of a base abstract domain. Should no optimal disjunction points be known, then disjunctive refinement can be used in an automatic iterative process [29]. Alternatively, relational, non-convex domains only support very limited types of non-convexity and relations [30, 31].

With BDDStab, we present a non-relational, non-convex abstract domain for integer value analysis, where each dimension is not restricted in its non-convexity. In essence, we track an arbitrary set of integer values for each variable, which means that in our example, we can represent S by an abstract state that approximates only by the two circled states in Figure 3.1. To stay efficient, even for large variation domains, we use BDD-based integer sets.

Until now, we have discussed the abstraction and concretization functions, as well as the set of admissible abstract values of abstract value analyses. Another crucial part of abstract domains are the transfer functions, which modify a given abstract state in accordance to an analyzed instruction. As an example, given the interval abstraction of S and the instruction $y = x + y$, the interval abstract domain sets y to $[0\,..\,8]$, which is computed by addition of the lower and upper bounds of the values x and y in the abstraction of S. However, to analyze a program, it is not sufficient to simply replace all transitions with the corresponding transfer function and execute the resulting program, because in the resulting approximative abstract version, it is in general not possible to decide the outcome of conditional statements. Therefore, these programs are

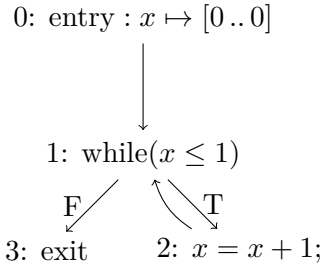
not deterministic and a different way of analysis must be used. We present the most common one, namely data flow analysis, in Section 3.2.

3.2. Data Flow Analysis

Data Flow Analysis (DFA) is a general analysis framework for structured programs, originally proposed by Gary Kildall [32] to perform global program optimization. It has since evolved into a general analysis tool that is used not only in optimization [33], but also in verification [17, 18].

As an example, we will execute the interval analysis on the CFG in Figure 3.2a. We use the following simplified transfer functions, where the subscript denotes to which statement they apply. Additionally, we assume the abstract state $\sigma^{\#[]}$ to contain only the abstract value of x and use $f^{\#[]}_{[\text{while}(x\leq 1)]^T}$ when the condition $x \leq 1$ is true and $f^{\#[]}_{[\text{while}(x\leq 1)]^F}$ otherwise. The $\emptyset$ symbol denotes an empty interval.

$$
\begin{aligned}
&f^{\#[]}_{[x=x+1]}(\sigma^{\#[]}) = [l+1\,..\,h+1] \text{ with } [l\,..\,h] = \sigma^{\#[]}\\
&f^{\#[]}_{[\text{while}(x\leq 1)]^T}(\sigma^{\#[]}) = \begin{cases} [l\,..\,\texttt{min}(h,1)] & \text{if } [l\,..\,h] = \sigma^{\#[]} \wedge \exists v \in \sigma^{\#[]} : v \leq 1\\ \emptyset & \text{otherwise}\end{cases}\\
&f^{\#[]}_{[\text{while}(x\leq 1)]^F}(\sigma^{\#[]}) = \begin{cases} [\texttt{max}(l,2)\,..\,h] & \text{if } [l\,..\,h] = \sigma^{\#[]} \wedge \exists v \in \sigma^{\#[]} : v > 1\\ \emptyset & \text{otherwise}\end{cases}\\
&f^{\#[]}_{[\text{entry}]}(\sigma^{\#[]}) = \sigma^{\#[]}
\end{aligned}
$$



(a) Example Program

$$
\begin{aligned}
&\text{IV}_\circ(0) = \text{IV}_\bullet(0) = [0\,..\,0]\\
&\text{IV}_\circ(1) = \text{IV}_\bullet(0) \cup \text{IV}_\bullet(2)\\
&\text{IV}_\bullet(1_T) = f^{\#[]}_{[\text{while}(x\leq 1)]^T}(\text{IV}_\circ(1))\\
&\text{IV}_\bullet(1_F) = f^{\#[]}_{[\text{while}(x\leq 1)]^F}(\text{IV}_\circ(1))\\
&\text{IV}_\circ(2) = \text{IV}_\bullet(1_T)\\
&\text{IV}_\bullet(2) = f^{\#[]}_{[x=x+1]}(\text{IV}_\circ(2))
\end{aligned}
$$

(b) Associated Constraint System

Figure 3.2.: Examples of Integer Interval Analysis

DFA establishes a constraint system between abstract states on all edges. Using subscript $\text{IV}_\circ(n)$ to denote the abstract state before the statement n, $\text{IV}_\bullet(n)$ for the abstract state directly after the statement n, and again denoting true and false information using subscript T and F, we get a constraint system as depicted in Figure 3.2b.

Because the program from Figure 3.2a contains a loop, the constraint system includes a cyclic dependency. $\text{IV}_\circ(1)$ is the summarization of all abstract states from all incoming

edges, namely the entry node and Node 2. However, $\text{IV}_\bullet(2)$ depends on $\text{IV}_\circ(2)$, which in turn depends on $\text{IV}_\circ(1)$.

In his foundational work [32], Kildall proposes a worklist algorithm (Algorithm 1) to compute a solution to constraint systems as the one above. As a generalization, the algorithm assumes that the property space, i.e., the set of possible data flow facts, forms a complete lattice. Complete lattices are non-empty partially ordered sets that support a join operation, denoted $\sqcup$, that the algorithm uses instead of the union we used in $\text{IV}_\circ(1)$, and provide a unique smallest element, denoted $\bot$, which the analysis uses for initialization. We provide a formal definition of complete lattices in Section 3.4. For the interval abstract domain, $\bot$ is the empty interval $\emptyset$, $\sqcup$ is the same as the union of two intervals, and the ordering $\sqsubseteq$ is given by interval inclusion, i.e., if an interval A covers B then $B \sqsubseteq A$. The worklist algorithm is defined in Figure 1. It takes as input the set of edges F of the analyzed program, a set of entry nodes E, an initial abstract value ι, the complete lattice L of the analyses' property space defining $\bot$, $\sqcup$, and $\sqsubseteq$, as well as a transfer function f_l. The algorithm computes the most precise fix point of the constraint system in $\texttt{MFP}_\circ$ and $\texttt{MFP}_\bullet$, where again the subscript $\circ$ denotes information before a given node and the corresponding subscript $\bullet$ information after this node.

In Step 1, the algorithm initializes the worklist W with all edges from the program's CFG, as well as the table $\texttt{A}$, which contains the data flow information at each node l. By default, this table is initialized with the least element of the complete property space lattice ($\bot$). The only exception is that the information at initial nodes is set to the initial abstract state ι.

Step 2 is the main part of the algorithm, which computes a solution to the given constraint system. The worklist contains all edges along which constraints may not yet have been propagated. Therefore, if the worklist is empty, the algorithm terminates, as the constraints along all edges are fulfilled. However, initially, the worklist is not empty and the algorithm pops the first edge from it. It retrieves the corresponding flow data from $\texttt{A}$ and applies the corresponding transfer function f_l. If the newly computed information is already aptly represented by the data flow information at the target ($\texttt{A}[l] \sqsubseteq \texttt{A}[l']$), then it restarts at Step 2. Otherwise, it updates the data flow information at the target, which means that constraints along paths starting at the target have to be reestablished. Therefore, the algorithm pushes the start edges of such paths onto the worklist, so that they will be operated on in a subsequent iteration.

In Step 3, the algorithm presents the solution, where the entry information at each node is given by $\texttt{A}$, and the exit information is recomputed using the entry information and the transfer function.

Algorithm `worklist` *(F, E, ι, L, f_l)* **is**
Input: Analysis flow F, analysis entries E, initial value ι, complete lattice L, transfer functions f_l
Result: Solution in $\texttt{MFP}_\circ$ and $\texttt{MFP}_\bullet$

Step 1: Initialization
1 **forall** $(l, l') \in \texttt{F}$ **do** $W \coloneqq (l, l') :: W$
2 **forall** $l \in \texttt{F} \cup \texttt{E}$ **do** **if** $l \in \texttt{E}$ **then** $\texttt{A}[l] \coloneqq \iota$ **else** $\texttt{A}[l] \coloneqq \bot_L$

Step 2: Iteration
3 **while** $W \neq \emptyset$ **do**
4 $(l, l') :: t = W$
5 $W \coloneqq t$
6 **if** $f_l(\texttt{A}[l]) \not\sqsubseteq \texttt{A}[l']$ **then**
7 $\texttt{A}[l'] \coloneqq \texttt{A}[l'] \sqcup f_l(\texttt{A}[l])$
8 **forall** $\{l'' \mid (l', l'') \in \texttt{F}\}$ **do** $W \coloneqq (l', l'') :: W$
end
end

Step 3: Presentation
9 **forall** $l \in \texttt{F} \cup \texttt{E}$ **do**
10 $\texttt{MFP}_\circ(l) \coloneqq \texttt{A}[l]$
11 $\texttt{MFP}_\bullet(l) \coloneqq f_l(\texttt{A}[l])$
end
end

Algorithm 1: DFA Worklist Algorithm

Table 3.1 shows the evolution of the worklist algorithm's values while solving the constraint system from Figure 3.2b. The underlined entry in the worklist is the element that is extracted from the worklist in Line 4 to produce the next iteration's state. After initialization, only the entry node has data flow information different from the least element of the correponding lattice. Since the transfer function for the entry node is the identity function, and since the information at A[1] is smaller than that at A[entry], A[1] is updated. Since information is updated at position 1, the algorithm would add all edges starting at 1 to the worklist. However, we omit adding entries that are already in the worklist, as this would only lead to additional steps that do not change the algorithm's result. In iteration 2, the transfer function $f^{\#[]}_{[\text{while}(x \leq 1)]^T}$ is used, since we compute information for the true successor of Node 1. Because all elements in the interval $[0\,..\,0]$ fulfill the condition, we update A[2] to $[0\,..\,0]$. The next iteration (#3) applies transfer function $f^{\#[]}_{[x=x+1]}$ producing $[1\,..\,1]$, which, unioned with the existing $[0\,..\,0]$ at A[1], gives $[0\,..\,1]$. Since information at position 1 is updated, the edge $(1, 2)$ is added to the worklist. Executing another round of iterations over the edges $(1, 2)$ and $(2, 1)$ leaves us with $\texttt{A}[1] = [0\,..\,2]$ in iteration 6. This time, the application of the

#	A[entry]	A[1]	A[2]	A[exit]	Worklist
1	$[0\,..\,0]$	$\emptyset$	$\emptyset$	$\emptyset$	$\underline{(\text{entry}, 1)}, (1, 2), (2, 1), (1, \text{exit})$
2	$[0\,..\,0]$	$[0\,..\,0]$	$\emptyset$	$\emptyset$	$\underline{(1, 2)}, (2, 1), (1, \text{exit})$
3	$[0\,..\,0]$	$[0\,..\,0]$	$[0\,..\,0]$	$\emptyset$	$\underline{(2, 1)}, (1, \text{exit})$
4	$[0\,..\,0]$	$[0\,..\,1]$	$[0\,..\,0]$	$\emptyset$	$\underline{(1, 2)}, (1, \text{exit})$
5	$[0\,..\,0]$	$[0\,..\,1]$	$[0\,..\,1]$	$\emptyset$	$\underline{(2, 1)}, (1, \text{exit})$
6	$[0\,..\,0]$	$[0\,..\,2]$	$[0\,..\,1]$	$\emptyset$	$\underline{(1, 2)}, (1, \text{exit})$
7	$[0\,..\,0]$	$[0\,..\,2]$	$[0\,..\,1]$	$\emptyset$	$\underline{(1, \text{exit})}$
8	$[0\,..\,0]$	$[0\,..\,2]$	$[0\,..\,1]$	$[2\,..\,2]$	

Table 3.1.: Worklist Evolution of Interval Analysis on Program in Figure 3.2a

transfer function $f^{\#[]}_{[\text{while}(x \leq 1)]^T}$ results again in $[0\,..\,1]$, meaning that no new information was computed and therefore no edge must be added to the worklist. After the application of the transfer function $f^{\#[]}_{[\text{while}(x \leq 1)]^F}$ in iteration 7, the worklist is empty and the loop terminates. The entries for $\mathtt{A}[n]$ correspond to $\text{IV}_\circ(n)$ from our constraint system in Figure 3.2b. We omit the computation of the corresponding set $\mathtt{IV}_\bullet(n)$ by application of the corresponding transfer function in Step 3.

Apart from forward analyses, where F is the set of edges in the control flow graph of the analyzed program, there also exist analyses that propagate flow data in the opposite direction of the control flow. If the data flow information at an edge depends on that of control flow successors, then the corresponding analysis must be specified as a backward analysis, and otherwise as a forward analysis. Because we analyze the possible values of a program, and the program's values at a specific location depend on the values in predecessor locations, value analyses are generally defined as forward DFA. However, there also exist formulations for bi-directional DFA [34]. Since DFA stores one flow value per edge, the memory demands of an analysis can get unwieldy if no sharing can be implemented and the abstract representation is not compact. In such cases, sparse DFA [35] reduces the number of stored values during analysis. Storing one abstract state per edge is especially demanding in the analysis of low-level code, because one instruction in a high-level language is usually compiled to many instructions in low-level code. Hence, this code contains more edges, requiring the storage of more abstract values. Our BDD-based integer sets are shared, meaning that we never store the same BDD twice, which alleviates the storage problem.

An alternative formulation of DFA is configurable program analysis (CPA) [28], which aims at providing a framework that can scale between classic DFA and model checking. CPA makes analyses composable via special operator configurations and a common interface that all analyses must support. In difference to DFA, the CPA formulation does not require a static control flow graph. Instead, the control flow automaton is computed during the analysis itself [28].

3.3. Configurable Program Analysis

When applying traditional data flow analysis to binaries, one encounters a problem: The algorithm assumes the existence of a correct static control flow graph (CFG). On structured programs, this CFG is provided by language constructs. As an example, the targets of a structured `if` instruction are limited to the true and the false successor. As discussed in Chapter 2, this is not the case for executables, where the successor address can be taken from a register, and hence can hold an arbitrary value. Therefore, binaries do not provide a precise CFG. Theoretically, it would be possible to supply a fully connected graph that contains as nodes all possible disassemblies of the binary. The analysis itself can determine which edges are not traversable. Even though this method would work, the fully connected CFG would obviously not yield any meaningful analysis and can therefore be ignored.

Jakstab instead builds on an alternative formulation of data flow analysis called configurable program analysis (CPA) [28]. CPA was initially created as a way to define and implement model checking and data flow analysis in the same framework. To that end, CPA defines two operators that control when to merge information and when to stop the analysis. Delayed merging shifts the analysis towards model checking, where, traditionally, no merging is performed at confluence points, while aggressive merging shifts the analysis towards data flow analysis, where, traditionally, merging is performed at each confluence point. Furthermore, CPA facilitates the implementation of composed analyses to improve the result. One of the composed analyses usually provides control flow information that is provided by the static CFG in DFA.

CPA is presented formally in Definition 1. D is the abstract domain, which itself consists of a set of concrete states C, a join semi-lattice $\mathcal{E}$ of abstract states, and a concretization function $[\![\cdot]\!]$ that maps abstract states to concrete states. Furthermore, $\rightsquigarrow$ is the transfer relation, which assigns to each abstract state e, a possible new abstract states e', and to each such relation a control flow label $g \in G$, written $e \overset{g}{\rightsquigarrow} e'$.

Definition 1. Configurable Program Analysis $\mathbb{D}$

$$\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$$

$$D = (C, \mathcal{E}, [\![\cdot]\!])$$

$$\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$$

$$[\![\cdot]\!] : E \to \mathcal{P}(C)$$

$$\rightsquigarrow \subseteq E \times G \times E$$

Additionally, the following conditions apply for correct CPAs:

1. $[\![\top]\!] = C$
2. $[\![\bot]\!] = \emptyset$

3. $\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$ (join operator overapproximates)
4. $\forall e \in E : \exists e' \in E, g \in G : e \overset{g}{\rightsquigarrow} e'$ (transfer relation is total)
5. $\forall e \in E, g \in G : \bigcup_{e \overset{g}{\rightsquigarrow} e'} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \{c' \mid c \overset{g}{\rightarrow} c'\}$ (transfer relation overapproximates)
6. $e' \sqsubseteq \text{merge}(e, e')$ (merge relaxes second argument with information from first)
7. $\text{stop}(e, R) \implies [\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$ (analysis only stops if e is already covered by reached abstract states)

For both the merge and stop operators, there exist variants that configure the analysis towards model checking, where information is kept separate at confluence points, or towards traditional data flow analysis where information is joined at confluence points. The two operator variants are as follows:

$$\text{merge}^{\text{sep}}(e, e') = e'$$
$$\text{merge}^{\text{join}}(e, e') = e \sqcup e'$$
$$\text{stop}^{\text{sep}}(e, R) = \exists e' \in R : e \sqsubseteq e'$$
$$\text{stop}^{\text{join}}(e, R) = e \sqsubseteq \bigsqcup_{e' \in R} e'$$

One of the most common CP analyses is the location analysis, as described in Definition 2, which introduces control flow graph information to CPA. Essentially, the location analysis provides a transfer relation from one location to another, exactly when there exists a corresponding edge in the analyzed program's CFG. In binary analysis, the fact that we can alter the way locations are treated is useful as no static CFG is available. By itself, the location analysis does not compute a useful result. Hence, it is usually combined with another analysis, which constitutes the main part of the analysis. CPAs can therefore be composed as described in the following section.

Definition 2. Location CPA $\mathbb{L}$ for CFG with set of nodes N and edges G

$$\mathbb{L} = (D_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}}, \text{merge}_{\mathbb{L}}, \text{stop}_{\mathbb{L}})$$
$$D_{\mathbb{L}} = (C_{\mathbb{L}}, \mathcal{E}_{\mathbb{L}}, [\![\cdot]\!]_{\mathbb{L}})$$
$$\mathcal{E}_{\mathbb{L}} = (N \cup \{\top_{\mathbb{L}}, \bot_{\mathbb{L}}\}, \top_{\mathbb{L}}, \bot_{\mathbb{L}}, \sqsubseteq_{\mathbb{L}}, \sqcup_{\mathbb{L}})$$
$$d_{\text{a}} \sqsubseteq_{\mathbb{L}} d_{\text{b}} \iff d_{\text{a}} = \bot_{\mathbb{L}} \vee d_{\text{b}} = \top_{\mathbb{L}}$$
$$l \rightsquigarrow_{\mathbb{L}} l' \iff \exists (l, l') \in G, l \rightsquigarrow_{\mathbb{L}} \bot_{\mathbb{L}} \text{ otherwise}$$
$$\text{merge}_{\mathbb{L}} = \text{merge}^{\text{sep}}$$
$$\text{stop}_{\mathbb{L}} = \text{stop}^{\text{sep}}$$

Composing Configurable Program Analyses

It is possible and common to combine two configurable program analyses into one. Assuming two input CPAs $\mathbb{D}_i = (D_i, \rightsquigarrow_i, \text{merge}_i, \text{stop}_i)$, the combined CFA will use the Cartesian product to combine the property spaces, i.e., $\mathbb{D}_\times = (D_\times, \text{merge}_\times, \text{stop}_\times)$ with $D_\times = D_1 \times D_2 = (C, \mathcal{E}_\times, [\![\cdot]\!]_\times)$ and $\mathcal{E}_\times = \mathcal{E}_1 \times \mathcal{E}_2 = (E_1 \times E_2, (\top_1, \top_2), (\bot_1, \bot_2), \sqsubseteq_\times, \sqcup_\times)$. The ordering and the join are defined pointwise, i.e., $(e_1, e_2) \sqsubseteq_\times (e_1', e_2') \iff e_1 \sqsubseteq_1 e_1' \wedge e_2 \sqsubseteq_2 e_2'$ and $(e_1, e_2) \sqcup_\times (e_1', e_2') = (e_1 \sqcup_1 e_1', e_2 \sqcup_2 e_2')$. In difference to the traditional reduced direct product, the transfer relation, merge and stop operators are defined specifically to each composition and can therefore improve precision more than reduction, e.g., because the transfer relation of one domain may be improved by the information from the other in a more direct way.

Algorithm `worklistCPA(`$\mathbb{D}$`, `ι`)` **is**
- **Input:** Configurable program analysis $\mathbb{D} = (D = (C, \mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup), [\![\cdot]\!]), \rightsquigarrow, \text{merge}, \text{stop})$, initial abstract state $\iota \in E$
- **Result:** Set of reachable abstract states
- Step 1: Initialization
- `waitlist` $\coloneqq \{\iota\}$
- `reached` $\coloneqq \{\iota\}$
- Step 2: Iteration
- **while** `waitlist` $\neq \emptyset$ **do**
  - $e :: t =$ `waitlist`;
  - `waitlist` $\coloneqq t$;
  - **forall** e' `with` $e \rightsquigarrow e'$ **do**
    - **forall** $e'' \in$ `reached` **do**
      - $e_{\text{new}} \coloneqq \text{merge}(e', e'')$
      - **if** $e_{\texttt{new}} \neq e''$ **then**
        - `waitlist` $\coloneqq$ (`waitlist` $\cup \{e_{\text{new}}\}) \setminus \{e''\}$
        - `reached` $\coloneqq$ (`reached` $\cup \{e_{\text{new}}\}) \setminus \{e''\}$
      - **end**
    - **end**
    - **if** $\neg$`stop`$(e', \texttt{reached})$ **then**
      - `waitlist` $\coloneqq$ `waitlist` $\cup \{e'\}$
      - `reached` $\coloneqq$ `reached` $\cup \{e'\}$
    - **end**
  - **end**
- **end**
- **return** `reached`

end

Algorithm 2: CPA Worklist Algorithm

Similarly to the worklist algorithm for DFA (see Algorithm 1), the CPA algorithm

$$\mathcal{E}_{\text{IV}} = (\mathcal{E}_{\mathbb{L}} \times \mathcal{E}'_{\text{IV}})$$

$$(l, i) \rightsquigarrow_{\text{IV}} (l', i') \iff l \rightsquigarrow_{\mathbb{L}} l' \wedge f_l^{\text{IV}}(i) = i'$$

$$\text{merge}_{\text{IV}}((l, i), (l', i')) = \begin{cases} (l, \text{merge}^{\text{join}}(i, i')) & \text{if } l = l' \\ (l', i') & \text{otherwise} \end{cases}$$

$$\text{stop}_{\text{IV}}((l, i), I) = \text{stop}^{\text{join}}(i, \{i' \mid (l', i') \in I, l' = l\})$$

$$\mathbb{D}_{\text{IV}} = ((C, \mathcal{E}_{\text{IV}}, [\![\cdot]\!]), \rightsquigarrow_r, \text{merge}_{\text{IV}}, \text{stop}_{\text{IV}})$$

$\iota_{\text{IV}} = (l, \iota)$, where l is an entry node of the analyzed program

Figure 3.3.: Integer Interval CPA

#	`reached`	`waitlist`
1	{ $(0, [0\,..\,0])$ }	$\underline{(0, [0\,..\,0])}$
2	{ $(0, [0\,..\,0])$, $(1, [0\,..\,0])$ }	$\underline{(1, [0\,..\,0])}$
3	{ $(0, [0\,..\,0])$, $(1, [0\,..\,0])$, $(2, [0\,..\,0])$, $(3, \emptyset)$ }	$\underline{(2, [0\,..\,0])}, (3, \emptyset)$
4	{ $(0, [0\,..\,0])$, $(1, [0\,..\,1])$, $(2, [0\,..\,0])$, $(3, \emptyset)$ }	$\underline{(1, [0\,..\,1])}, (3, \emptyset)$
5	{ $(0, [0\,..\,0])$, $(1, [0\,..\,1])$, $(2, [0\,..\,1])$, $(3, \emptyset)$ }	$\underline{(2, [0\,..\,1])}, (3, \emptyset)$
6	{ $(0, [0\,..\,0])$, $(1, [0\,..\,2])$, $(2, [0\,..\,1])$, $(3, \emptyset)$ }	$\underline{(1, [0\,..\,2])}, (3, \emptyset)$
7	{ $(0, [0\,..\,0])$, $(1, [0\,..\,2])$, $(2, [0\,..\,1])$, $(3, [2\,..\,2])$ }	$\underline{(3, [2\,..\,2])}$
8	{ $(0, [0\,..\,0])$, $(1, [0\,..\,2])$, $(2, [0\,..\,1])$, $(3, [2\,..\,2])$ }	

Table 3.2.: CPA Evolution when Analyzing Figure 3.3

(Algorithm 2) uses a worklist to remember from which abstract states the transfer relation may compute new information. Corresponding to the DFA algorithm's `A` array, the CPA algorithm uses the `reached` set to hold all reached abstract states. The two main differences between the algorithms are that the CPA algorithm treats control flow as part of the analysis, and that it allows keeping data flow information separate using the merge and stop operators.

To repeat the integer interval analysis from Figure 3.2a, we define the interval analysis as a CPA in Figure 3.3. In CPA, the interval analysis is paired with the location analysis. Note that we neither supply the lattice of intervals $\mathcal{E}'_{\text{IV}}$ formally, nor do we provide a concretization function $[\![\cdot]\!]_{\mathbb{L}}$ or a set of concrete states $C_{\mathbb{L}}$. As in DFA, the CPA algorithm only needs the bottom, join, and relation operators of $\mathcal{E}'_{\text{IV}}$, which we discussed in the DFA section (Section 3.2). Concretization is only needed to show correctness of the analysis, and hence not required for its execution.

Table 3.2 shows the evolution of the `reached` and `waitlist` sets when analyzing the example in Figure 3.2a. With the given interval analysis configuration, a CP analysis computes the same result as the standard worklist algorithm from DFA, with the only difference that the information that was stored in `A` is now stored in the reached set.

3.4. Lattices

As described in Sections 3.2 and 3.3, DFA and CPA require the space of data flow information to form a lattice. Lattices are partially ordered sets (see Definition 4) that include a unique least upper bound (join, $\sqcup$) and greatest lower bound (meet, $\sqcap$) for any two elements of the set (see Definition 5). Complete lattices additionally support least upper bound and greatest lower bound operations for subsets (see Definition 6).

Figure 3.4 shows a Hasse Diagram of the powerset lattice $(\mathcal{P}(\{a, b\}), \subseteq)$. Hasse Diagrams are the most common way of visualizing small lattices. Essentially, Hasse Diagrams use the transitive reduction with an embedding such that if an element is depicted above another, and there exists a path between the two elements, then the lower element is smaller than the upper element. As an example, $\emptyset$ is depicted lower than $\{a, b\}$ in Figure 3.4 and there exists a path between the two, therefore $\emptyset \sqsubseteq \{a, b\}$.

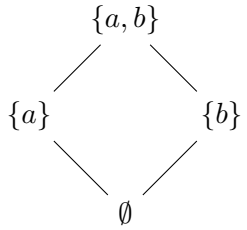


Figure 3.4.: Hasse Diagram of $(\mathcal{P}(\{a, b\}), \subseteq)$

Definition 3. Upper and Lower Bounds of Set S
$\mathrm{UB}(S) = \{e \mid \forall s \in S : s \sqsubseteq e\}$
$\mathrm{LB}(S) = \{e \mid \forall s \in S : e \sqsubseteq s\}$

Definition 4. Partially Ordered Set
A partially ordered set is a combination of a set P and an ordering relation $\sqsubseteq$: $(P \times P) -> \mathbb{B}$ that fulfills the following criteria:

1. $\forall a : a \sqsubseteq a$ (reflexivity)
2. $\forall a, b : (a \sqsubseteq b) \wedge (b \sqsubseteq a) \implies (a = b)$ (antisymmetry)
3. $\forall a, b, c : (a \sqsubseteq b) \wedge (b \sqsubseteq c) \implies (a \sqsubseteq c)$ (transitivity)

Definition 5. Lattice
A lattice is a partially ordered set P that supports the binary least upper bound ($\sqcup$) and greatest lower bound ($\sqcap$) operations (see Definition 3) as follows:
$\forall a, b \in P : (a \sqcup b) \in P \wedge (a \sqcup b) \in \mathrm{UB}(\{a, b\}) \wedge (\forall c \in \mathrm{UB}(\{a, b\}) : (a \sqcup b) \sqsubseteq c)$
$\forall a, b \in P : (a \sqcap b) \in P \wedge (a \sqcap b) \in \mathrm{LB}(\{a, b\}) \wedge (\forall c \in \mathrm{LB}(\{a, b\}) : c \sqsubseteq (a \sqcap b))$

Definition 6. Complete Lattice

A lattice L is complete when it supports the least upper bound and greatest lower bound operations on subsets $(\bigsqcup, \bigsqcap)$ as follows:

$\forall S \subseteq L : \bigsqcup S \in L \wedge \bigsqcup S \in \mathrm{UB}(S) \wedge (\forall c \in \mathrm{UB}(S) : \bigsqcup S \sqsubseteq c)$

$\forall S \subseteq L : \bigsqcap S \in L \wedge \bigsqcap S \in \mathrm{LB}(S) \wedge (\forall c \in \mathrm{LB}(S) : c \sqsubseteq \bigsqcap S)$

3.5. Abstract Interpretation

Abstract interpretation (AI) is a proof framework for proving correctness of program analyses with respect to formal semantics of the language of analyzed programs. AI was first described by Patrick and Radhia Cousot in 1977 [36], and has been a very active research field since. We will first present the AI framework, and then derive the requirements for the correctness of abstract value analysis domains.

With AI, one establishes a correctness relation R between an execution of program P and its analysis. Specifically, the relation R is established between the states of the analyzed program and their corresponding summarizations in the analysis. If a program maps a state c_1 to a state c_2, then the analysis must map an abstract state a_1, which is in a correctness relation with c_1, to an abstract state a_2, which is in a correctness relation with c_2. Figure 3.5a visualizes the correctness relation R. Here, execution of the program P maps the input state c_1 to c_2, denoted as $P \vdash c_1 \rightsquigarrow c_2$. Further, the analysis of the program P maps a value a_1 to a_2, denoted as $P \vdash a_1 \triangleright a_2$. If the correctness relation R holds between c_1 and a_1 ($c_1\ R\ a_1$), then it must also relate c_2 with a_2 ($c_2\ R\ a_2$) for the analysis to be correct. Generally, $\rightsquigarrow$ corresponds to a statement in the analyzed program, and $\triangleright$ to the corresponding transfer function.

Recall that each abstract domain requires its state space L to form a complete lattice. AI requires that if $s\ R\ a$ then $\forall a' \sqsupseteq a : s\ R\ a'$, meaning that if an abstract value is a correct description of a concrete value, then greater values must also be a correct description of that value. Intuitively speaking, this requirement enforces the lattice of abstract values to be ordered with respect to precision. Greater values are generally less precise than smaller values. AI further requires that there exists a most precise description for each concrete value. With these two requirements, we can alternatively formulate correctness via the representation function as depicted in Figure 3.5. This formulation of correctness makes explicit that we are allowed to approximate, using values greater than the least, i.e., most precise value.

For value analysis on integer programs, it is natural to accept β as $\beta(s) = \{s\}$, i.e., given a state s we return the singleton set containing s. The generated correctness relation R would then also accept all supersets of $\{s\}$ as a correct representation. Unfortunately, an analysis on sets of states is not feasible on infinite state systems, as no summarization of states has taken place. This summarization can be defined by establishing a Galois Connection between a precise, but possibly infeasible analysis, and an approximative variation. The Galois Connection ensures that if the precise analysis was correct, then so is the approximative one.

(a) Correctness Relation (b) Representation Function

Figure 3.5.: Correctness Requirements in AI

3.6. Galois Connection

A Galois Connection allows the conversion of values from one state space to another without losing correctness. It uses α to move from the precise space (concrete space) to the less precise one (abstract space), and γ to move back. Note that α corresponds to the abstraction function of an abstract domain, while γ corresponds to its concretization function. The beauty of this approach is that Galois Connections can be chained, meaning we can move to more and more abstract spaces without losing correctness. Hence, AI is a modular framework, because it is not necessary to start abstractions at the most basic level, i.e., at the level of program states, but instead, one can abstract from any proven point. It is, e.g., common for non-relational value analysis domains to abstract from integer sets, and avoid the work necessary to get to this abstraction.

The formal definition of Galois Connections is given in Definition 7.

Definition 7. Galois Connection between L (concrete space) and M (abstract space), Classic Formulation

α and γ are monotone (order-preserving) functions
$\alpha : L \to M$
$\gamma : M \to L$
$l \sqsubseteq \gamma(\alpha(l))$
$\alpha(\gamma(m)) \sqsubseteq m$

Definition 7 confirms that M (abstract space) is a less precise variant of L (concrete space) in that it requires that going from the more precise L to the less precise M and back will not increase precision ($\gamma \circ \alpha$ does not map to smaller values). Further, it also requires that going from the less precise M to the more precise L and back does not lose precision ($\alpha \circ \gamma$ does not map to greater values).

Assuming a concrete state using a set of associations between program variables $V^\star$ and integer values, and denoting an association between v and e as $v \mapsto e$, an abstraction from the analysis over sets of states can be constructed as follows, where we use a superscript # to denote an abstract state:

$$\alpha(S) = \{v \mapsto \bigcup_{s \in S} \{s(v)\} \mid v \in V^\star\}$$

$$\gamma(S^\#) = \prod_{v \in V^\star} \{v \mapsto e \mid e \in S^\#(v)\}$$

The abstraction function α goes through all program variables, and associates each with all values found under this variable in all states in S. Correspondingly, γ creates a set of associations for all values of each variable, and then uses the Cartesian Product over all resulting sets. As an example, let us abstract the set of states $S = \{\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 1\}\}$:

$$\alpha(S) = \{x \mapsto \{0, 1\}, y \mapsto \{0, 1\}\}$$

The concretization is then the following:

$$\gamma(\alpha(S)) = \{\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 0, y \mapsto 1\}, \{x \mapsto 1, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 1\}\}$$

We use this abstraction in our integer value analysis domain.

3.7. Heap Analysis

Value analyses are often combined with specialized analyses of the program's heap, which can be considered as an association of each address, given by an integer, with a value. Since value analyzers overapproximate integers, the target of read and write operations can, in general, not be determined. Sound integer value analyzers must overappoximate the heap, which requires read and write operations on sets of possible addresses, which may overlap. Implementing a sound heap model is often challenging without concretizing the abstract value that describes the address to read and write operations. Instead of viewing the heap as a simple association, addressable using integers, it is common to introduce abstract locations when objects are created. The contents of these abstract locations is then restricted to structures of specific shapes [37, 38]. When loops that created data structures cannot be shown to terminate, it must be possible to efficiently represent infinite structures. Finding an efficient representation is especially challenging if no information about the shape of the represented structure is known before the analysis. As an example, an abstract location may at first contain a hashmap, and later on a list; hence shape analysis must be adaptive. Many shape analyses are based on shape graphs [39], which represent which object may contain which other object. Parametric shape analysis [40] uses three valued logic to model the shape of structures in memory. BDDStab currently does not use shape analysis as such, but uses Jakstab's heap model instead. In Jakstab, each abstract heap location is indexable, i.e., each abstract location consists of a region and an offset address into that region [3]. The abstract regions are assumed to not overlap, which is unsound in general, but simplifies the analysis. In the default configuration, Jakstab only uses two regions to distinguish addresses pointing to the stack from those that do not.

3.8. Value Analysis Tools for Binaries

Different approaches with different soundness requirements are proposed for the analysis of machine code. Most approaches opt for improved applicability to larger programs and do not aim for soundness. However, this does not mean that increasing precision of a sound domain is not a valuable effort. It is, e.g., common to identify possible function start sequences in a binary and start value analysis at each of these sequences [7, 41]. A more precise abstract domain will improve the result of analyzers that use this approach.

Possibly the best-known commercial tool is GrammaTech's CodeSurfer/x86 [41], which assumes that the code it analyzes has a clear separation of data and code, does not modify itself, and adheres to stack discipline. Hence it may not be suited to analyze executables that are designed to hide their behavior. However, CodeSurfer/x86 does use value analysis (k-set analysis) in combination with abstract locations, determined in part using IDAPro's heuristics, to compute possible targets for dynamic jumps.

The Jakstab framework [3] uses the SSL intermediate language [42] to support the formulation of analyses. For the analyses, it makes use of the CPA formulation, which means that approaches known from software model checking as well as classic DFA and combinations thereof are implementable. Jakstab provides several value analyses to resolve jump targets, such as k-set, interval, and sign-agnostic interval domains [43]. We extend Jakstab with an unrestricted set domain with abstract values represented by BDDs.

Another analysis framework for the analysis of executables is the binary analysis platform (BAP) [44]. BAP's focus is on providing a well-specified intermediate language as a common grounds for implementing further analyses. BAP uses special instructions for indirect jumps, which may be resolved using value set analysis [45]. However, it seems that the main focus of BAP is on working with code in its intermediate language BIL, and not on soundly lifting machine code to BIL.

Bitblaze [46], which is in part a predecessor to BAP, combines static and dynamic analyses to increase precision. Its dynamic analysis part is based on the Qemu [47] emulator.

Another comprehensive effort is the Bincoa framework [48], which also formulates a common model for the analysis of executables. Its tool set consists of Osmose [49], a test data generation tool for binaries, Insight [50], a tool that supports lifting and analysis including value analysis, and TraceAnalyzer [51], a tool that uses value analysis and refinement of VDs that are used in dynamic jumps and have been overapproximated to $\top$.

All of the above-mentioned tools that use value analysis, approximate VDs using either a form of intervals, or k-sets, or a combination thereof. In difference, with BDDStab, we provide a value analysis that does neither enforce convexity nor is restricted to a specific type of non-convexity.

3.9. Binary Decision Diagrams

Binary Decision Diagrams (BDD) [52, 53, 54] are an efficient graph representation for Boolean functions, i.e, functions of the type $\mathbb{B}^n \to \mathbb{B}$, that consist of decision nodes and terminal nodes. Decision nodes have two outgoing edges to sub-BDDs and carry as label a variable that determines which edge must be followed during evaluation of the BDD. The terminal node's label determines the result of the function. Usually, BDDs are based on the Shannon Decomposition, shown in Definition 8, where x_k is used as label in a decision node, and the BDD representation of f_{x_k} and $f_{\neg x_k}$ are used as targets of the decision node's edges. When repeated application of Shannon Decomposition and construction of decision nodes yields a Boolean constant, then this constant is used as label in a terminal node. One early form of BDD is the free BDD (FBDD) [55], which enforces that no path within the BDD may contain two decision nodes with the same label, in order to avoid reading the same input more than once.

Definition 8. Shannon Decomposition

Assuming an n-ary Boolean function $f(x_1, \ldots, x_n) = f(x)$, the positive and negative cofactors of f are defined as

$f_{x_k}(x) = f(x_1, \ldots, x_{k-1}, 1, x_{k+1}, \ldots, x_{n-1})$

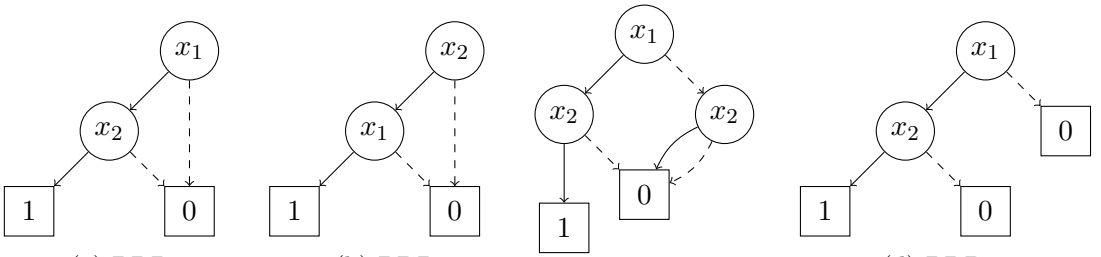
and

$f_{\neg x_k}(x) = f(x_1, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_{n-1})$.

Any n-ary Boolean function can be decomposed into two functions of arity $n-1$ by the Shannon Decomposition:

$f(x) = (x_k \wedge f_{x_k}(x)) \vee (\neg x_k \wedge f_{\neg x_k}(x))$

In the following, we will visualize decision nodes using a circular node, labeled with the corresponding variable. If this variable is true, then evaluation must continue at the sub-BDD reachable by traversing the solid edge, and otherwise it must continue at the sub-BDD reachable using the dashed edge. Terminals are represented using rectangular nodes, labeled with the Boolean constant they represent. In text form, we use $\boxed{1}$ for a true terminal, and $\boxed{0}$ for a false terminal. The graphs in Figure 3.6 represent Boolean `and`.



(a) BDD 1 (b) BDD 2 (c) BDD 3 (d) BDD 4

Figure 3.6.: Equivalent BDDs Representing `and`

The core advantage of BDDs is that they support canonicity, meaning that any two equivalent Boolean functions have the same BDD representation, which allows equivalence checking in $O(1)$, and in turn enables a systematic optimization of functions on BDDs using caches [56]. Since the BDDs in Figure 3.6 represent the same Boolean function, but use different graphs, it is clear that some restrictions are missing to enforce canonicity. The first restriction is to define a total order on all variables, and enforce that the encountered labels on all decision nodes are strictly ascending along all paths of the BDD. BDDs that fulfill this restriction are called ordered (OBDD) [54]. If we choose the order to include $x_1 < x_2$, then BDD 2 is disallowed. To disallow BDDs 3 and 4, only BDDs that have been fully reduced with the following two rules are allowed.

1. When both successors of a decision node are the same BDD, delete the node and redirect its incoming edges to the successor.

2. When a BDD contains two equal sub-BDDs, delete one and redirect its incoming edge to the other.

Since the right x_2 node in BDD 3 can be removed using Rule 1, and one of the 0 sub-BDDs in BDD 4 is redundant and can therefore removed by Rule 2, both BDDs are forbidden, leaving only BDD 1. OBDDs that are unchanged by both rules such as BDD 1 are called reduced OBDD (ROBDD) [54, 56] and support canonicity. Since ROBDDs are the most common BDD structure, we will use the terms BDD and ROBDD synonymously in the remainder of this thesis.

There has been extensive research in optimizing the size of BDDs by optimizing the variable order [57, 58, 59]. Unfortunately, ordering optimizations depend on heuristics, and reordering BDDs is NP-complete [60]. Further, most operations on BDDs require all operand BDDs to respect the same, or a compatible, ordering, which means that it is unlikely that a chosen ordering is near optimal for all BDDs.

Another optimization is the complementable edges extension [61], which makes the interpretation of the terminal's label path-dependent, whereby it allows up to twice as many BDDs to be shared in memory. With this extension, a BDD and its complement share the same structure, and differ only in one Boolean that is stored outside of the shared BDD part. Consider the BDD in Figure 3.7, which represents $a \wedge \neg b$, where a and b are ordered as follows: $a > b$.

BDDs with complementable edges only require one terminal node, i.e., 1. With complemented edges, the incoming edges to a node can signal that the result of the evaluation must be complemented, hence the interpretation of the 1 node is path-dependent. We represent edges that signal complementation of their targets using a circle in the middle of them, while edges that do not signal complementation are visualized without the circle. To allow complementing of entire BDDs, a dangling incoming edge to the BDD is added. This edge is not part of the BDD structure itself, and is therefore not shared. Hence, a BDD and its complement are shared and the number of unique BDDs is halved.

The interpretation of a BDD with complement edges is as follows: Paths that contain an even number of complement edges lead to 1, paths that contain an odd number of

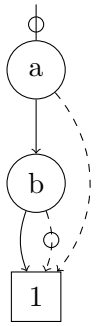


Figure 3.7.: BDD Representing $a \wedge \neg b$

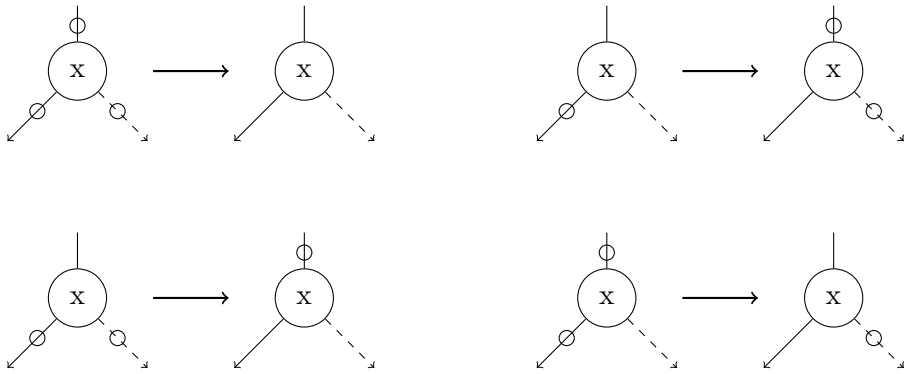


Figure 3.8.: Complemented Edges Conflicts

complement edges lead to $\boxed{0}$. The only path with an even number of complementations in the BDD from Figure 3.7 is $a = 1, b = 0$. All other paths have only the complementation from the dangling edge and will therefore lead to $\boxed{0}$.

There is, however, one restriction missing to make complemented edges canonical. Consider Figure 3.8, which contains pairs of semantically equivalent BDDs with differing structure. To restore canonicity to BDDs with complemented edges, it is forbidden to have a solid complement edge of the internal nodes. Consequently, whenever a node is to be constructed with a solid complement edge, the complement information on the dangling edge as well as the one on the unset edge are inverted instead.

An alternative formulation of BDDs can be derived from the Davio decomposition, provided in Definition 9, instead of the classic Shannon decomposition. Using either the positive or negative Davio Decomposition exclusively results in the functional decision diagrams, named pFDD for positive decomposition and nFDD for negative decomposition [62]. It is also possible to use positive or negative decomposition per variable, which results in general FDDs [63].

Similarly to FDDs, one can also allow using Shannon, and positive and negative Davio Decomposition per variable. Decision diagrams constructed with all three decomposition

types are called Kronecker functional decision diagrams (KFDD) [64].

Definition 9. Davio Decomposition
We define the Boolean derivation of a function $f(x_1, \ldots, x_n)$ as

$\frac{\partial f}{\partial x_k} = f_{x_k}(x) \oplus f_{\neg x_k}(x)$.

Any Boolean function with $n > 0$ inputs $f(x_0, \ldots, x_{n-1})$ can be decomposed in two ways into sub-functions with $n - 1$ inputs:

1. Positive decomposition: $f = f_{\neg x_k} \oplus x_k \wedge \frac{\partial f}{\partial x_k}$
2. Negative decomposition: $f = f_{x_k} \oplus \neg x_k \wedge \frac{\partial f}{\partial x_k}$

Apart from introducing alternative decompositions, other techniques have been researched to reduce the size of the decision diagrams for specific configurations. When solving combinatorial problems, it is often desirable to manipulate subsets of the powerset of an n-element set. Those subsets, commonly called combinations, can be represented by BDDs by assigning one input variable to each combination, and storing the indicator function of a set of combinations in a BDD. When n is large, there exist extremely many input variables to the indicator function, hence the reduction rule of BDDs that removes decision nodes must be efficient. It is common for combination sets to be sparse, i.e., combination sets usually are small compared to their maximum size. Therefore, if the indicator function of common combination sets is true, then it is likely that most inputs are false. Representing such function in a classic BDD would lead to many decision nodes where the true successor edge points to $\boxed{0}$, meaning the decision node cannot be deleted. To have a more efficient representation in this application, Minato introduced zero suppressed BDDs (ZDD) [65], which are classic BDDs based on Shannon's Decomposition with a modified reduction rule. Instead of eliminating a decision node exactly when both its successors are the same node, in ZDDs, decision nodes are removed if their true edge points to $\boxed{0}$, leading to an efficient representation for sparse combination sets.

A common generalization of classic BDDs is to relax the requirement that leaves contain a Boolean, and instead allow values from another finite domain. In verification, this domain of values is often a subset of integral numbers. Such decision diagrams are called multi-terminal BDDs (MTBDDs) [66]. A special form of MTBDDs are algebraic BDDs (ADDs), which are MTBDDs that are specialized for algebraic applications [67].

Yet another way in which decision diagrams differ from classic BDDs are attribute edges, where edges carry information other than whether they correspond to a true or false decision. An optimization to MTBDDs with integer leaves are edge-valued BDDs (EVBDDs) [68], which attribute the BDD edges with integer offsets to improve sharing. Such edge weights can also be generalized to factors, where the edge attributes along a path to a leaf are not added to the leaf's integer, but multiplied instead.

Another decision diagram, designed for the verification of timed automata, are difference decision diagrams (DDD) [69], which represent a Boolean logic over inequalities of the form $x - y \leq c$. Modified algorithms from classic BDDs support unification and intersection of the non-convex and relational space that DDDs represent.

Numeric decision diagrams (NDDs) [70] are designed with a similar goal as DDDs, namely the verification of timed automata. Specifically, NDDs target the representation of discrete and dense clock variables, the latter by discretization. Essentially, NDDs represent bitvectors that may be built from the concatenation of bitvectors that represent variables in the analyzed automaton.

For our BDD-based integer set implementation, we introduce a variant of ROBDDs with complement edges that is optimized for representing Boolean functions with few input variables. Our BDDs do not label the decision nodes with their corresponding variable, which enables sharing of BDDs that are structurally equivalent but represent different functions.

4. BDDStab Library

The BDDStab library enables the definition of integer value analysis domains for data-flow-based analysis tools using abstract interpretation. It takes its name from its use in the binary analysis platform Jakstab via the BDDStab Jakstab adapter. In this application, the BDDStab library provides the heavy lifting, i.e., the representation of abstract integer values and the transfer functions. By using maximally shared BDDs to represent abstract values, the BDDStab library minimizes the memory consumption of the many abstract states that must be stored during analysis. The right side of Figure 4.5 shows the BDD representation of the integer set $\{1, 2\}$. Each path of the BDD that terminates in $\boxed{1}$ represents included integers, in this case 1 for the grey path, which encodes $0 * 2^1 + 1 * 2^0$ since the decision node corresponding to 2^1 is false and the node corresponding to 2^0 is true, and 2 for the orange path, which encodes $1 * 2^1 + 0 * 2^0$. Complement edges, as introduced in Section 3.9, allow the same BDD to be represented by just two nodes. Further savings are achieved if sub-ranges covering powers of two, i.e., $[2^n .. 2^{n+m} - 1]$ are either included or excluded from the represented set, because then, entire sub-BDDs are set to $\boxed{1}$ or $\boxed{0}$ respectively. In difference to most other value analysis domains such as the interval abstract domain, the BDDStab library does not require convexity and provides precise transfer functions on the bit level. Table 4.1 gives an overview of the provided functions and in case of transfer functions, categorizes their approximation behavior, where we use `P` for precise, `A` for approximating, and an empty cell if the method is not a transfer function. We consider an n-ary transfer function of a concrete operator as precise exactly when it produces exactly the set of all combinations of the n input sets with the operator applied. A transfer function $\circ^\#$ is precise for the binary operator $\circ$ exactly when $A \circ^\# B = \{a \circ b \mid a \in A, b \in B\}$.

In this chapter, we describe our BDD representation in detail and provide implementations for all common transfer functions.

4.1. BDD Structure

To optimize for the operations that we use in BDDStab, we use a modified BDD structure. We base our structure on reduced ordered binary decision diagrams with complementable edges, but modify it to improve sharing. Specifically, we do not store variable names in each decision node's label as would be done using traditional BDDs, and make the correspondence between a decision node and its variable dependent on the depth of the decision node in the BDD. Additionally, we add set cardinality, i.e., determining the number of represented concrete elements, in $O(1)$ by precomputing values during each construction of a decision node. This precomputation needs to be adapted to work so that it is dependent on the BDD depth, because when each decision node's label is determined by its depth, the same BDD, rooted at different depths, may represent sets of different sizes. As an example, a $\boxed{1}$ node can represent varyingly many elements, depending on its depth in a BDD.

Method Name	Prec.	Description
`equals`		True exactly when two sets contain the same elements
`add`		Creates new set with a given element added
`remove`		Creates new set with a given element removed
`contains`		Check is a given element is included
`invert`		Creates a new set containing those elements not included in the original set
`intersect`		Creates a set by intersecting with another set
`union`		Creates a set by unioning with another set
`max`		Selects the largest element (signed)
`min`		Selects the smallest element (signed)
`size`		The set cardinality
`subsetOf`		True exactly when given set is a superset
`isEmpty`		True exactly when set is empty
`plus`	`P`	Transfer function for addition
`negate`	`P`	Transfer function for arithmetic negation
`bAnd`	`P`	Transfer function for bitwise $\wedge$
`bOr`	`P`	Transfer function for bitwise $\vee$
`bXor`	`P`	Transfer function for bitwise $\veebar$
`bNot`	`P`	Transfer function for bitwise $\neg$
`bShr`	`P`	Transfer function for shift right by singleton
`bSar`	`P`	Transfer function for shift right arithmetic by singleton
`bShl`	`P`	Transfer function for shift left by singleton
`bRor`	`A`	Transfer function for right rotation by singleton
`bRol`	`A`	Transfer function for left rotation by singleton
`mulSingleton`	`P`	Transfer function for multiplication by singleton
`mulPredicate`	`A`	Transfer function for general multiplication
`bitExtract`		Slicing on the bit level
`signExtend`		Sign extend to new bitwidth
`zeroFill`		Zero fill to new bitwidth
`restrictGreaterOrEqual`		Create subset of elements greater or equal given element
`restrictLessOrEqual`		Create subset of elements less or equal given element
`restrictGreater`		Create subset of elements greater given element
`restrictLess`		Create subset of elements less given element
`widenPrecisionTree`		Create widened superset
`apply`		Create a set from elements
`range`		Create a set from range
`strided`		Create a set from congruence class

Table 4.1.: Functions of BDDStab Library

4.1.1. Labelless Nodes

During analysis, a value analyzer, such as the BDDStab adapter to Jakstab (Section 5), produces at least one abstract state per reachable program location. Lifted versions of low-level code, such as executables, tend to increase the number of program locations, as each effect of an instruction is made explicit. Each abstract state in turn is composed of one abstract value per storage location, of which there may exist as many as there are heap entries. Therefore, a value analyzer tends to store extremely many abstract values, the collection of which must therefore be size-efficient. We therefore decide to optimize the well-known BDD structure to improve sharing. Additionally, improved sharing also allows for improved memoizing of functions. Our labelless nodes modification of BDDs does not store variable names in the decision nodes, but makes the decision node's variable dependent on the depth of the decision node in the BDD itself. The advantage of this modification is that two structurally equivalent BDDs can be shared, even if they represent different boolean functions. Consider the BDDs in Figure 4.1. The rectangular node that points to the BDDs contains the BDD ordering and is headed by the named Boolean function. The top-most marked (grey) decision node for f_{a} decides on variable x_2, as it is the second node on all BDD paths. Because this decision node points to $\boxed{1}$ for the true successor and $\boxed{0}$ for the false successor, it represents the function x_2. In f_{b}, the same marked structure represents the function x_1, as the top-most marked decision node is the first on all paths in this BDD. Since all three marked sub-BDDs in Figure 4.1 are structurally equivalent, it is possible to share them as done in the right-hand side forest. To make the depth of a node determine its label, we modify reduction Rule 1 as follows:

1. When both successors of a decision node are the same terminal, delete the node and redirect its incoming edges to the terminal.

The disadvantage of this approach is that the reduction rules of traditional BDDs must be changed to allow a clear correspondence between decision node and its place in the BDD ordering. There may not be any gaps, meaning variables in the ordering that have no corresponding decision node, between any two decision nodes in a labelless BDD. Only nodes with false and true successor pointing to the same terminal node may be deleted. Therefore, a labelless BDD such as the one representing f_{c} may have more nodes than one with explicit labels within its decision nodes. In this BDD, the root node would traditionally have been removed, as both successors point to the same BDD. However, with labelless nodes, this is not possible as removal would change the interpretation of the top-most marked decision node. On the upside, none of the marked sub-BDDs could have been shared in a traditional BDD structure, but all of them are shared with labelless nodes.

Since equivalence check on BDDs is in $O(1)$, it is possible to skip redundant decision nodes in $O(1)$ during BDD operations. The penalty in the efficiency of the algorithms that has to be expected therefore depends on the maximal number of variables in the ordering. Since this number of variables in the ordering is bounded by the widest integer type to be analyzed, no huge negative impact must be expected. As an example,

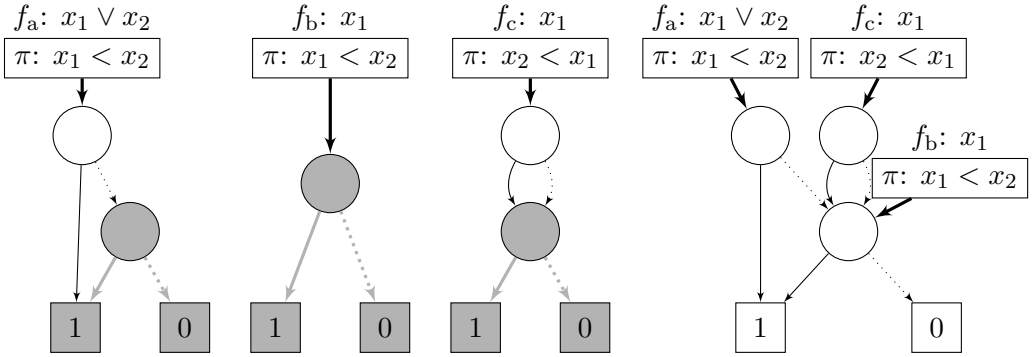


Figure 4.1.: Improved Sharing with Labelless Nodes

the BDDStab Jakstab adapter uses at most 64 variables in the ordering to adequately represent 32 bit integers after multiplication.

4.1.2. Structural Induction

To prove correctness of our algorithms on BDDs, we will use proof by induction. More specifically, we use structural induction, as we operate on trees rather than natural numbers. Assuming a property P that is to be shown for arbitrary trees, structural induction operates by showing P for the basic elements of the tree, i.e., the leaves, and then showing that if P holds for the sub-trees, it also holds for a tree constructed from those sub-trees. Formally, structural induction on BDDs for an arbitrary property P is therefore:

Base Cases: $P(\boxed{0})$, $P(\boxed{1})$

Inductive Case: $\forall X, Y : P(X) \wedge P(Y) \implies P((X \boxed{\circ} Y))$

It is important for an induction scheme to be complete, meaning that if base case and inductive case are proven, then it must be possible to construct P for arbitrary BDDs.

Induction Scheme for Cartesian Product BDD Algorithms

Interestingly, our algorithms for binary transfer functions, such as addition, operate on two BDDs and their correctness must therefore be encoded in a binary property, i.e., of the form $P(X, Y)$ instead of $P(X)$. Possibly the first idea that one might have is to use the Cartesian product of the base cases and extend the inductive step as follows:

Scheme 1. Incomplete Induction Scheme for Cartesian Product Fashion Algorithms

Base Cases: $P(\boxed{0}, \boxed{0})$
$P(\boxed{0}, \boxed{1})$,
$P(\boxed{1}, \boxed{0})$,
$P(\boxed{1}, \boxed{1})$

Inductive Case: $\forall A, B, C, D : P(A, B) \wedge P(C, D) \implies P((A \boxed{\circ} B), (C \boxed{\circ} D))$

This scheme, however, is not complete. Consider the BDD in Figure 4.2.

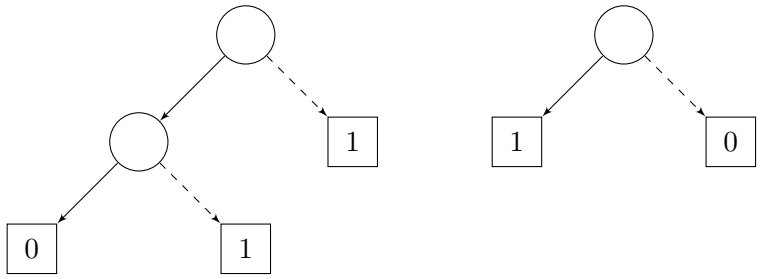


Figure 4.2.: BDDs not Covered by Induction Scheme 1

This tuple of BDDs is constructed using $(((\boxed{0}\odot\boxed{1})\odot\boxed{1}),(\boxed{1}\odot\boxed{0}))$. To conclude $P(((\boxed{0}\odot\boxed{1})\odot\boxed{1}),(\boxed{1}\odot\boxed{0}))$, it is necessary to have $P((\boxed{0}\odot\boxed{1}),\boxed{1})$ and $P(\boxed{1},\boxed{0})$. However, $P((\boxed{0}\odot\boxed{1}),\boxed{1})$ cannot be constructed since there exists no way to construct a non-terminal BDD on one side and a terminal on the other. Hence, P cannot be established using only the base and inductive cases for the BDD in Figure 4.2, because the inductive case forces the node constructor in both arguments to P, which is why only pairs of BDDs with the same depth are covered by this induction scheme. Furthermore, our algorithms proceed in a Cartesian product fashion, and require correct sub-results not only for $P(A,B)$ and $P(C,D)$ but also for $P(A,D)$ and $P(B,C)$.

We therefore propose the following induction scheme:

Scheme 2. Induction Scheme for Cartesian Product Fashion Algorithms

Base Cases: $\forall B : P(\boxed{0},B)$,
$\forall B : P(\boxed{1},B)$,
$\forall A : P(A,\boxed{0})$,
$\forall A : P(A,\boxed{1})$

Inductive Case: $\forall A,B,C,D : P(A,C) \wedge P(A,D) \wedge P(B,C) \wedge P(B,D)$
$\implies P((A\boxed{\odot}B),(C\boxed{\odot}D))$

With this induction scheme, it is possible to derive P for the example BDD in Figure 4.2 as follows:

$P((\boxed{0}\odot\boxed{1}),\boxed{1}) \wedge P((\boxed{0}\odot\boxed{1}),\boxed{0}) \wedge P(\boxed{1},\boxed{1}) \wedge P(\boxed{1},\boxed{0})$
$\implies P(((\boxed{0}\odot\boxed{1})\odot\boxed{1}),(\boxed{1}\odot\boxed{0}))$

All Ps in the antecedent (induction hypothesis) of this derivation are covered by base cases, hence P can be derived for the whole tuple of BDDs.

4.1.3. Set Cardinality in $O(1)$

When operating on sets, one often wants to know the cardinality, i.e., how many elements are included. With the BDD-based set encoding, the corresponding operation is satisfiability count, which essentially searches for paths to true terminals in the BDD. Each discovered path to $\boxed{1}$ represents 2^n satisfiable variable valuations, where n is the

number of variables without a corresponding decision node. Unfortunately, this simple algorithm requires full traversal of the BDD, which should be avoided as the operation is often executed, e.g., to judge precision loss in widening, determine if a precise heap operation would be too expensive as there are too many possible addresses, considering whether printing the set to the screen should list all elements, or to produce a good approximation in operations such as multiplication. For traditional BDDs, it is possible to compute and store the satisfiability count whenever a BDD is constructed from two other BDDs by taking the precomputed satisfiability count from each input BDD, multiplying them by 2^n, where n is the size of the gap between the new node and the root node of the corresponding input BDDs, and adding the result. However, since labelless nodes do not carry variables, this simple algorithm cannot be used as the number of satisfiable valuations of a labelless BDD depends on the layer it is rooted in. The solution is to cache not the total number of satisfiable variable valuations, but the minimal number and height instead. Consider the two left-most BDDs in Figure 4.1. The marked sub-BDD contains 2^0 satisfiable variable valuations in f_a, because there is no variable left in the ordering at the decision node's successors, and 2^1 such paths in f_b, because there is one variable left in the ordering at the decision node's successors. The difference between the two interpretations is that one is rooted at a higher layer in the ordering than the other.

A key insight is that if a BDD has height h, then each terminal at depth d represents at least $2^{h-d} = c$ satisfiable variable valuations. As we have learned from the example, the exact number of satisfiable variable valuations represented by each $\boxed{1}$ depends additionally on where the corresponding BDD is rooted in the ordering. Assuming the number of elements in the ordering to be n, the number of variables that are not referenced within a given BDD of height h is $n - h$. Since these variables multiply the number of satisfiable variable valuations by 2 each, the exact number of satisfiable variable valuations can be computed by multiplying the minimal number of satisfiable variable valuations with 2^{n-h}. The parameters c and h depend only on the structure of a given BDD, and not on its interpretation. Hence, it is safe to cache the two parameters for each decision node, even in the presence of sharing. If n is known, which is the case in the BDDStab library, then the parameters c and h are sufficient in computing the exact satisfiability count using $c * 2^{n-h}$.

To not negatively affect BDD performance, the computation of the parameters c and h for a new decision node must be in $O(1)$. The parameters c and h for $\boxed{0}$ and $\boxed{1}$ are $c = 0$, $h = 0$, and $c = 1$, $h = 0$ respectively. Given two BDDs s and u, with $s.h$, $u.h$, $s.c$, and $u.c$ denoting their height and minimum satisfiability count respectively, then the new height of a node $(s \boxed{\circ} u)$ is $\mathtt{max}(s.h, u.h) + 1 = (s \boxed{\circ} u).h$, the computation of which is indeed in $O(1)$. Computing $(s \boxed{\circ} u).c$ from $s.h$, $s.c$, $u.h$, and $u.c$ cannot be done by simply adding $s.c$ and $u.c$ because the minimum number of input variables, referenced by $(s \boxed{\circ} u)$, is determined by its height, which in turn is computed using the maximum height of s and u. Within $(s \boxed{\circ} u)$, the minimum number of referenced input variables of s and u is $\mathtt{max}(s.h, u.h)$. If $s.h \leq u.h$, we therefore compute $(s \boxed{\circ} u).c = s.c * 2^{|s.h-u.h|} + u.c$, which adjusts the minimum satisfiability count of s to the depth of u. Similarly if $u.h \leq s.h$,

$$\texttt{satnaive}(n, \boxed{1}) = 2^n$$
$$\texttt{satnaive}(n, \boxed{0}) = 0$$
$$\texttt{satnaive}(n, (s \boxed{\circ} u)) = \texttt{satnaive}(n-1, s) + \texttt{satnaive}(n-1, u)$$

Figure 4.3.: Satcount Naive

we compute $(s \boxed{\circ} u).c = u.c * 2^{|s.h - u.h|} + s.c$. In both cases, the computation is in $O(1)$.

As an example, consider the creation of the unmarked node of f_a. The height of the newly computed BDD is 2, the minimum satisfiability count is 3, while the height of the marked sub-BDD A is 1 and its minimum satisfiability count is 1. With $A = (\boxed{1} \boxed{\circ} \boxed{0})$, i.e., the marked sub-BDD, the corresponding computation of the minimum satisfiability count for f_a is as follows:

$$A.c = \boxed{1}.c * 2^{\boxed{1}.h - \boxed{0}.h} + \boxed{0}.c = \boxed{1}.c + \boxed{0}.c = 1$$

$$(\boxed{1} \boxed{\circ} A).c = \boxed{1}.c * 2^{\boxed{1}.h - A.h} + A.c = 1 * 2^1 + 1 = 3$$

Since the number of input variables n is known for each BDD r, and the height and minimum satisfiability count are cached, we define our set cardinality as:

$$\texttt{sat}(n, r) = 2^{n - r.h} * r.c$$

The $O(1)$ BDD-set cardinality has been implemented and evaluated as part of a bachelor thesis [71].

In the following, we prove equivalence between the non-naive method of computing the number of satisfiable variable valuations as outlined above, and a naive implementation as given in Figure 4.3. This naive algorithm takes the number of input variables to the represented function and a corresponding BDD, and recurses the BDD until it reaches a terminal node. During this recursion, it keeps track of the number of traversed input variables, and is thereby able to compute the number of unique paths to $\boxed{1}$, represented by a $\boxed{1}$.

Figure 4.4 proves equivalence between the naive and the non-naive satisfiability count algorithms. The proofs of the two base cases (4.1, 4.2) are trivial. The value of $r.c$ is 0 for a terminal false node and 1 for a terminal true node. The proof for the inductive case (4.3) is slightly more involved. Since the construction of $r.h$ for a new node r uses the `max` function and the construction of $r.c$ takes the absolute value of an expression, we use case distinction in our proof for $r_1.h \leq r_2.h$ and $r_1.h > r_2.h$. Note that for the first case, we have $(r_1 \boxed{\circ} r_2).h = r_2.h + 1$ and $(r_1 \boxed{\circ} r_2).c = r_2.c + 2^{r_2.h - r_1.h} * r_1.c$, since we take the maximum height plus one and have to adjust the minimum count of r_1 to the depth of r_2.

The correctness property P is as follows:

$P(n, r) \coloneqq \texttt{sat}(n, r) = \texttt{satnaive}(n, r)$

Induction on r yields the following base and inductive cases.

Base Cases:

$$\forall n : P(n, 0) \tag{4.1}$$
$$\forall n : P(n, 1) \tag{4.2}$$

Inductive Case:

$$(\forall n : P(n, r_1)) \wedge (\forall n : P(n, r_2)) \implies (\forall n : P(n, (r_1 \boxed{\circ} r_2))) \tag{4.3}$$

Proofs

Proof of 4.1:

$\texttt{sat}(n, 0) = 2^{n-0} * 0 = 0 = \texttt{satnaive}(n, 0)$

Proof of 4.2:

$\texttt{sat}(n, 1) = 2^{n-0} * 2^0 = 2^n = \texttt{satnaive}(n, 1)$

Proof of 4.3, case $r_1.h \leq r_2.h$:

$$\begin{aligned}
\texttt{sat}(n, (r_1 \boxed{\circ} r_2)) &= 2^{n-r_2.h-1} * (r_2.c + 2^{r_2.h-r_1.h} * r_1.c) \\
&= (2^{n-r_2.h-1} * r_2.c) + (2^{n-r_2.h-1} * 2^{r_2.h-r_1.h} * r_1.c) \\
&= (2^{n-1-r_2.h} * r_2.c) + (2^{n-1-r_1.h} * r_1.c) \\
&= \texttt{sat}(n-1, r_2) + \texttt{sat}(n-1, r_1) \\
&= \texttt{satnaive}(n-1, r_2) + \texttt{satnaive}(n-1, r_1) \\
&= \texttt{satnaive}(n, (r_1 \boxed{\circ} r_2))
\end{aligned}$$

Proof of 4.3, case $r_1.h > r_2.h$:

$$\begin{aligned}
\texttt{sat}(n, (r_1 \boxed{\circ} r_2)) &= 2^{n-r_1.h-1} * (r_1.c + 2^{r_1.h-r_2.h} * r_2.c) \\
&= (2^{n-r_1.h-1} * r_1.c) + (2^{n-r_1.h-1} * 2^{r_1.h-r_2.h} * r_2.c) \\
&= (2^{n-1-r_1.h} * r_1.c) + (2^{n-1-r_2.h} * r_2.c) \\
&= \texttt{sat}(n-1, r_1) + \texttt{sat}(n-1, r_2) \\
&= \texttt{satnaive}(n-1, r_1) + \texttt{satnaive}(n-1, r_2) \\
&= \texttt{satnaive}(n, (r_1 \boxed{\circ} r_2))
\end{aligned}$$

Figure 4.4.: Equivalence Proof of Cached Satisfiability Count

4.2. Representing Integer Sets

As outlined before, we use our BDD structure to represent integer sets in BDDStab. Traditionally, BDDs are a data structure to represent Boolean functions. Many structures can be encoded as Boolean functions and can therefore also be stored in a BDD. As is customary, our encoding of integer sets uses the two's complement conversion of integers to bitvectors, of which each bit is treated as an input to the Boolean function, represented by the BDD. This Boolean function returns true exactly when the corresponding bitvector and, therefore, the integer is included. In other words, we take the indicator function for the n-bit integer set, convert it to a Boolean function of the form $\{0,1\}^n \to \{0,1\}$, and represent that function using a BDD. Reduction of the resulting BDDs requires an ordering to be defined on its input variables, which we discuss in the next subsection.

Ordering

Theoretically, one could choose any permutation of the n input variables to represent n-bit integer sets. However, we also have to consider that we want to execute operations on the BDDs and these operation's efficiency depends on the variable ordering. There are two natural choices for the variable ordering: Most significant bit (MSB) to least significant bit (LSB) and LSB to MSB. LSB to MSB ordering, meaning the LSB is the bit associated with the root node of the BDD, simplifies the definition of the addition operation, since the carry bit distributes from LSB to MSB and so does each path in a BDD with LSB to MSB ordering. However, with this ordering, two consecutive integers are not close with respect to the BDD structure. Consider Figure 4.5, which depicts two non-reduced BDDs representing the integer set $\{0,1\}$, the one with LSB to MSB ordering on the left, and the one with MSB to LSB ordering on the right. Even though the represented numbers in the set are consecutive, the paths on the left-hand side BDD are not, meaning the terminal node for one number could be far to the left and the terminal node for the other number could be far to the right even though the numbers are consecutive. In difference, in the BDD on the right-hand side, two numerically close unsigned integers will have paths that are close to each other. In the MSB to LSB representation, it is therefore more likely that a range of integers will have an efficient BDD representation as it is likely that reduction is able to remove nodes where both successors are the true terminal. Figure 4.6 illustrates this fact. Whenever an interval is to be represented in a BDD with MSB to LSB ordering, there are essentially two paths, one corresponding to the lower bound and one corresponding to the upper bound, which border an area of included integers. In other words, the successors of the nodes on those paths that point inwards are set to the true terminal, and the successors pointing to the outside are set to the false terminal. Again, it is likely that there exist nodes that can be removed because both successors point to the terminal true in the inside (green) area. Since it is known that convex shapes, i.e., intervals in the non-relational case, are common during the analysis of software, we choose the MSB to LSB ordering to have a more efficient representation and conversion to and from intervals.

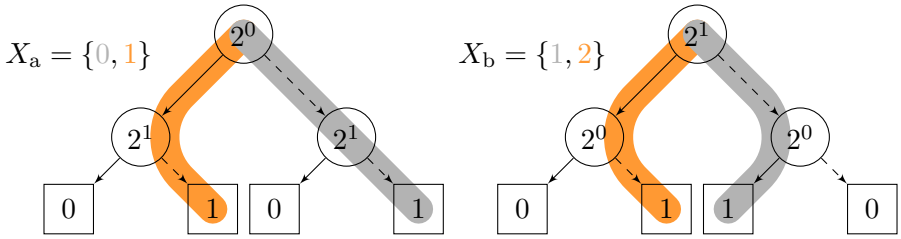


Figure 4.5.: Variable Ordering Options

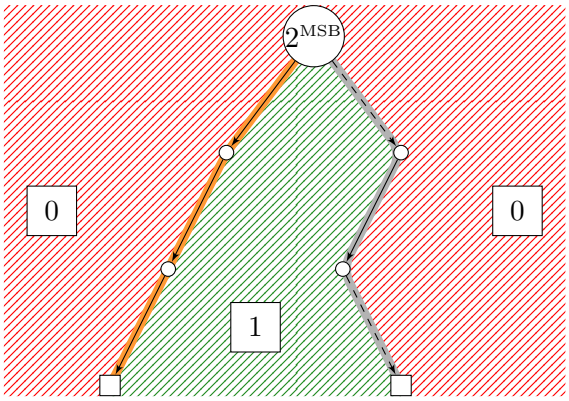


Figure 4.6.: Interval Conversion of $[2\,..\,7]$ with MSB to LSB Ordering

4.3. Conversion Algorithms

In this section, we present algorithms to create BDD-based integer sets from well-known abstractions, namely singleton integers, intervals, and strided intervals. Additionally, we provide an approximating algorithm to convert a BDD-based integer set to a set of intervals, which can be used to synthesize transfer functions on BDD-based integer sets from their corresponding counterparts in the interval abstract domain. Most of the algorithms are recursive, and many count their recursion depth in a parameter, which has to be set to 0 or a constant at the outer-most recursion. To avoid the explicit handling of this parameter in the presentation of an algorithm, we use named parameters that must be associated with a default value and must be set by name should this default value be overwritten. Note that to avoid ambiguities, no regular parameters are allowed after a named parameter in the parameter lists. The parameter d in Algorithm 3 is such a named parameter, denoted using $=$ in the algorithm's parameter list. Its default value is $n-1$, and it is explicitly set to $d-1$ in the recursive calls to `fromNumber`.

4.3.1. Configurable BDD Construction from Singletons

Algorithm 3 takes a singleton integer, and creates a BDD by creating the path described by the corresponding bitvector. This algorithm is configurable in the BDDs it places outside of the path described by the given integer, and by the terminal node it places

at the end of the path. It can therefore be used to convert the unsigned n-bit integer i not only to a singleton BDD set $\{i\}$ using `fromNumber(`i`, `$\boxed{0}$`, `$\boxed{0}$`, `$\boxed{1}$`)`, but also to a BDD-based integer set $\{e \mid e \geq i\}_{\{n\}}$ using `fromNumber(`i`, `$\boxed{1}$`, `$\boxed{0}$`, `$\boxed{0}$`)`. A set with all integers smaller than i can be created similarly.

Algorithm `fromNumber(`a`, `S`, `U`, `T`, `$d = n - 1$`)` **is**
  Input: n-bit integer a, depth d, set BDD S, unset BDD U, terminal BDD T
  Result: BDD with path given by a, terminated by T, all true successors leaving path to S, all false successors leaving path to U
1 **if** $d < 0$ **then return** T
2 **if** $a_{\{d\}}$ **then**
3 **return** (`fromNumber(`a`, `S`, `U`, `T`, `$d = d - 1$`)` $\boxed{\circ} U$)
  else
4 **return** ($S \boxed{\circ}$ `fromNumber(`a`, `S`, `U`, `T`, `$d = d - 1$`)`)
  end
end

Algorithm 3: Configurable Construction from Integer to BDD

As an example, let us use Algorithm 3 to create the set of all 3-bit unsigned integers greater or equal than 2. From Figure 4.6, we know that the corresponding BDD must contain the path corresponding to the bitvector of 2, i.e., 010, and all BDDs to the left or greater side must be set to $\boxed{1}$ and all BDDs to the right or smaller side must be set to $\boxed{0}$. This BDD can be constructed using `fromNumber(2, `$\boxed{1}$`, `$\boxed{0}$`, `$\boxed{1}$`, `$d = 2$`)`:

`fromNumber(2, `$\boxed{1}$`, `$\boxed{0}$`, `$\boxed{1}$`, `$d = 2$`)` =
  $(\boxed{1} \boxed{\circ}$ `fromNumber(2, `$\boxed{1}$`, `$\boxed{0}$`, `$\boxed{1}$`, `$d = 1$`)`$) =$
  $(\boxed{1} \boxed{\circ} ($`fromNumber(2, `$\boxed{1}$`, `$\boxed{0}$`, `$\boxed{1}$`, `$d = 0$`)` $\boxed{\circ} \boxed{0})) =$
  $(\boxed{1} \boxed{\circ} ((\boxed{1} \boxed{\circ}$ `fromNumber(2, `$\boxed{1}$`, `$\boxed{0}$`, `$\boxed{1}$`, `$d = -1$`)`$) \boxed{\circ} \boxed{0})) =$
  $(\boxed{1} \boxed{\circ} ((\boxed{1} \boxed{\circ} \boxed{1}) \boxed{\circ} \boxed{0})) =$
  $(\boxed{1} \boxed{\circ} (\boxed{1} \boxed{\circ} \boxed{0}))$

The resulting BDD is true if the 2^2 bit is set or if the 2^1 bit is set, otherwise it is false. This represents exactly the set of all 3-bit integers greater than 2.

4.3.2. BDD Creation from Standard Intervals

Algorithm 4 performs the conversion from an integer interval to a BDD-based integer set. The algorithm takes two $(d + 1)$-bit unsigned integers a and b, with $a \leq b$, and creates a BDD-based integer set $\{i \mid a \leq i \leq b\}$. Even though this algorithm operates on unsigned integers only, it is easily possible to use it for signed integers, by calling it once for all negative, and once for all positive integers and union the results. The algorithm uses `fromNumber` to create a set of all unsigned integers smaller than b and a set of all unsigned integers greater than a. The desired result is then the intersection of these two BDDs, which is computed using the well known if-then-else algorithm (`ite`) [56].

Algorithm `ival2bdd(a, b, d)` **is**
  Input: $(d+1)$-bit unsigned integers a and b, satisfying $a \leq b$
  Result: BDD-based integer set representing $[a..b]$
1 $s \coloneqq$ `fromNumber(`a`,` $\boxed{1}$`,` $\boxed{0}$`,` $\boxed{1}$`,` $d = d$`)`
2 $u \coloneqq$ `fromNumber(`b`,` $\boxed{0}$`,` $\boxed{1}$`,` $\boxed{1}$`)`
3 **return** `ite(`s`,` u`,` $\boxed{0}$`,` $d = d$`)`
end

Algorithm 4: Conversion from Interval to BDD-based Integer Set

As an example, we recreate the BDD from Figure 4.6, i.e., a BDD representing the 3-bit interval $[2\,..\,7]$, which is created using `ival2bdd(2, 7, 2)`:

`ival2bdd(2, 7, 2)` =
`ite(fromNumber(2,` $\boxed{1}$`,` $\boxed{0}$`,` $\boxed{1}$`,` $d = 2$`), fromNumber(7,` $\boxed{0}$`,` $\boxed{1}$`,` $\boxed{1}$`),` $\boxed{0}$`,` $d = 2$`)` =
`ite((`$\boxed{1}\odot(\boxed{1}\odot\boxed{0}$`)), fromNumber(7,` $\boxed{0}$`,` $\boxed{1}$`,` $\boxed{1}$`,` $d = 2$`),` $\boxed{0}$`)` =
`ite((`$\boxed{1}\odot(\boxed{1}\odot\boxed{0}$`)), (fromNumber(7,` $\boxed{0}$`,` $\boxed{1}$`,` $\boxed{1}$`,` $d = 1$`)`$\odot\boxed{1}$`),` $\boxed{0}$`)` =
`ite((`$\boxed{1}\odot(\boxed{1}\odot\boxed{0}$`)), ((fromNumber(7,` $\boxed{0}$`,` $\boxed{1}$`,` $\boxed{1}$`,` $d = 0$`)`$\odot\boxed{1}$`)`$\odot\boxed{1}$`),` $\boxed{0}$`)` =
`ite((`$\boxed{1}\odot(\boxed{1}\odot\boxed{0}$`)), (((fromNumber(7,` $\boxed{0}$`,` $\boxed{1}$`,` $\boxed{1}$`,` $d = -1$`)`$\odot\boxed{1}$`)`$\odot\boxed{1}$`)`$\odot\boxed{1}$`),` $\boxed{0}$`)` =
`ite((`$\boxed{1}\odot(\boxed{1}\odot\boxed{0}$`)), (((`$\boxed{1}\odot\boxed{1}$`)`$\odot\boxed{1}$`)`$\odot\boxed{1}$`),` $\boxed{0}$`)` =
`ite((`$\boxed{1}\odot(\boxed{1}\odot\boxed{0}$`)),` $\boxed{1}$`,` $\boxed{0}$`)` =
$(\boxed{1}\odot(\boxed{1}\odot\boxed{0}))$

This result is the same as the BDD for all unsigned 3-bit integers greater than 2, because 7 is the largest unsigned 3-bit number.

4.3.3. Approximated Set of Intervals from BDD

Algorithm 5 performs an approximation from BDD-based integer sets to sets of integer intervals. The precision parameter i determines the maximum depth to which the algorithm performs a precise conversion, before approximating sub-BDDs. When the algorithm has reached $\boxed{1}$, or arrived at the maximum depth, the algorithm creates an interval $[0\,..\,2^d - 1]$, offset by o. Since o tracks the value of the traversed edges within the recursion and $\boxed{1}$ at depth $n - d$ represents 2^d elements, the created interval contains exactly the values represented by the encountered $\boxed{1}$, or the non-terminal node at the maximum depth, replaced with $\boxed{1}$.

As an example, if the BDD from Figure 4.6 gets approximated with a maximum depth $i = 1$, then the result is $\{[0\,..\,3], [4\,..\,7]\}$, because the offset for the false successor of the root node is 0, while the offset for the true successor of the root node is 4. In the second recursion, the algorithm has reached its maximum depth and will therefore treat the sub-BDDs as $\boxed{1}$, giving $\{[0\,..\,0 + 2^2 - 1]\}$ on the false side, and $\{[4\,..\,4 + 2^2 - 1]\}$ on the true side.

The performance of Algorithm 5 can be optimized. As presented, the algorithm is configured by the precision parameter i, which is the maximum depth to which the algorithm will perform precise conversion of the given BDD-based integer set. This is suboptimal

since this simple precision configuration may lead to unacceptable overapproximation. Consider a singleton integer set. The conversion to an interval set should contain exactly one singleton interval. However, Algorithm 5 will produce this result only if given a precision parameter that is the depth of the BDD itself, which is unacceptable for most other BDDs.

An improved algorithm uses the $O(1)$ cardinality, supported by our BDD structure, to judge whether approximation by $\boxed{1}$ is acceptable or not. One possibility is to use i to define the maximum approximation allowed by each interval included in the result set. This behavior can be achieved in Algorithm 5 by replacing the call to `stopDepth` with `stopApprox` in Line 7. Of course it is also possible to combine the two approaches or replace the stopping condition with another heuristics.

```
   Algorithm bdd2ival(A_{n} , i = n/2, o = 0, d = n) is
   |  Input: BDD-based integer set A_{n} , depth d, offset o , precision parameter
   |         i < n with conservative default n/2
   |  Result: Set of intervals that overapproximate A_{n}
   |  Function stopDepth(i, d, x) is
1  |  |  return i = n − d
   |  end
   |  Function stopApprox(i, d, x) is
2  |  |  return 2^d − sat(x) ≤ i
   |  end
3  |  switch A do
4  |  |  case [1] do return {[o .. o + 2^d − 1]}
5  |  |  case [0] do return ∅
6  |  |  case (a [○] b) do
7  |  |  |  if stopDepth(i, d, A) then
8  |  |  |  |  return bdd2ival([1], i = i, o = o, d = d)
   |  |  |  else
9  |  |  |  |  return bdd2ival(a, i = i, o = o + 2^{d−1}, d = d − 1) ∪
   |  |  |  |   bdd2ival(b, i = i, o = o, d = d − 1)
   |  |  |  end
   |  |  end
   |  end
   end
```

Algorithm 5: Approximation from BDD-based Integer Set to Interval Set

4.3.4. BDD Creation from Strided Intervals

As we will motivate in Section 4.5, precise multiplication of a BDD-based integer set with a singleton requires the exact conversion of a strided interval to a BDD-based set. We implement this conversion in terms of Algorithm 7, which takes a maximum depth `maxdepth` that is the bitwidth of the output set and a stride, and creates the set of

`maxdepth`-bit integers that belong to the congruence class described by the stride without any offset. We then use the addition algorithm for BDD-based integer sets, as defined in Section 4.5, to add the required offset and intersect with the interval restriction of the strided interval. In other words, given a set $\{i \mid l \leq i \leq h, \exists k \in \mathbb{Z} : i = l + k * s, s \in \mathbb{N}, s > 0\}$, which is described by the strided interval $s[l\,..\,h]$, BDD creation of this interval works as described in Algorithm 6.

Algorithm `strided(maxdepth,` $s[l\,..\,h]$`)` **is**
  Input: Bitwidth of interval `maxdepth`, strided interval
  Result: BDD correspondence to given `maxdepth`-bit strided interval
1 `congruenceSet` $\coloneqq$ `congruence(maxdepth,` s`)`
2 `isect` $\coloneqq$ `ival2bdd(0,` $h - l$`, maxdepth` $- 1$`)`
3 `offset` $\coloneqq$ `fromNumber(`l`,` $\boxed{0}$`,` $\boxed{0}$`,` $\boxed{1}$`,` $d =$ `maxdepth` $- 1$`)`
4 **return** `add(congruenceSet` $\barwedge$ `isect, offset, maxdepth, fAdder)`
end

Algorithm 6: Conversion of Strided Interval to BDD-based Integer Set

Algorithm 7 performs the heavy lifting when converting strided intervals to BDDs. Given the desired bitwidth of the integer set (`maxdepth`), and a stride (`s`), it creates a BDD-based integer set $\{i \mid \exists k \in \mathbb{N} : i = k * s\}$. The algorithm starts at 0, which is always included, and counts in n how many elements must be skipped, i.e., not be included. Whenever the desired depth is reached, and no elements must be skipped anymore, the algorithm places a $\boxed{1}$ and resets n to `s` $- 1$, since it must skip this many elements before the next included element. Otherwise, the algorithm computes how many elements can be represented at the current position ($2^{\texttt{maxdepth}-d}$). If the algorithm must skip more elements than can be represented by the node at the current position, it places a $\boxed{0}$ and adjusts n accordingly. Else, there exists at least one path to a $\boxed{1}$ from the current position. Hence, the algorithm first creates the false BDD successor, and uses the returned `usetn` as n in the recursive call for the true BDD successor.

Even though this algorithm produces the desired BDD-based integer set, it is very inefficient for small strides, i.e., d-bit congruence sets that are almost full. As an example, consider the creation of a BDD-based congruence set for stride 2. Independent of the integer size, the only decision node that determines whether an element is included or not is 2^0, i.e., the one corresponding to the least significant bit. This simple BDD shape suggests that creating the BDD should be fast. However, building the BDD in a left depth-first way is extremely inefficient, because no sub-BDD can be skipped by placing $\boxed{0}$. Figure 4.7 shows the construction of a BDD for stride 2, and additionally lists at each position the algorithm's only non-constant d and n parameters. The final reduced BDD will have the true successor edge from the root not pointing to the 2^0 sub-BDD, however, as is, the algorithm will create this result inefficiently. A closer look at the parameters shows immediately that since the d and n parameters are equal for both successors of the root node, they must point to the same sub-BDD. Memoizing the algorithm, i.e., caching previously computed results, would avoid recomputation, with the effect that even BDD-based integer sets with a bigger bit size can be constructed efficiently. In the

stride 2 example, all successors of all decision nodes except the deepest one will each have the same n and d parameters, and can therefore use precomputed results. Essentially, for each depth d, there are $\texttt{stride} - 1$ possible different inputs to the algorithm, suggesting that the memoization will be less efficient for larger strides. However, for larger strides, there are more opportunities for the algorithm to place a $\boxed{0}$ terminal at a higher level, which increases efficiency. We therefore expect the algorithm to perform well for large and small strides.

Algorithm `congruence(maxdepth, stride,` $d = 0$`,` $n = 0$`)` **is**
  Input: Maximum BDD depth d, stride s, current depth d, values to skip n
  Result: BDD representing unsigned set $\{i \mid 0 \leq i \leq 2^d - 1, \exists k : i = k * s\}$, next n value
1 **if** $d = \texttt{maxdepth} \wedge n = 0$ **then**
2 **return** $(\boxed{1}, \texttt{stride} - 1)$
  else
3 $v \coloneqq 2^{\texttt{maxdepth}-d}$
4 **if** $v \leq n$ **then**
5 **return** $(\boxed{0}, n - v)$
  else
6 $(\texttt{uset}, \texttt{usetn}) \coloneqq \texttt{congruence(maxdepth, stride,}\ d = d + 1,\ n = n\texttt{)}$
7 $(\texttt{set}, \texttt{setn}) \coloneqq \texttt{congruence(maxdepth, stride,}\ d = d + 1,\ n = \texttt{usetn}\texttt{)}$
8 **return** $((\texttt{uset}\ \boxed{\circ}\ \texttt{set}), \texttt{setn})$
  end
  end
end

Algorithm 7: Congruence BDD Creation

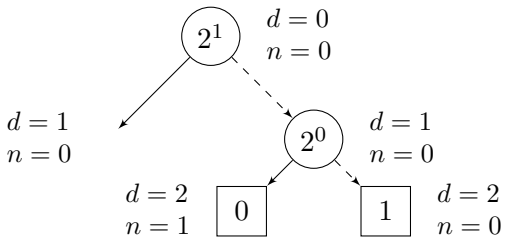


Figure 4.7.: 2-bit Congruence Set for Stride 2

4.3.5. BDD Transfer Functions from Interval Transfer Functions

Using Algorithms 4 and 5, together with the well known transfer function on intervals ($\circ^{\#[]}$), it is easy to define approximative transfer functions for BDD-based integer sets by first converting to interval sets, then applying the interval transfer function in a Cartesian fashion, and finally converting the result back to a BDD-based integer set as follows:

$$A \circ^{\#} B = \bigsqcup\!\!\!\bigsqcup_{\square} \{\texttt{ival2bdd}(a \circ^{\#[]} b) \mid a \in \texttt{bdd2ival}(A), b \in \texttt{bdd2ival}(B)\}$$

Transfer functions for unary operators can be derived similarly. We use derived interval functions whenever no direct algorithm on BDDs is available, e.g., in division as well as multiplication of large sets.

4.4. Higher-Order Algorithm for Binary Bitwise Operations

The binary bitwise operations, namely bitwise and, or, and xor, have in common that every nth output bit is determined by the two input bits at position n. This characteristic implies that, as long as both input BDDs adhere to the same ordering, which we assume in all operations since we always use MSB to LSB ordering, the recursion is not required to communicate information other than the computed sub-BDDs for the corresponding operation.

We will first present a configurable algorithm that supports all binary bitwise operations, and subsequently optimize it using additional base cases for the specific operations.

To develop the algorithm, we will first look at the bitwise variants of traditional binary Boolean operators, i.e., $\wedge$, $\vee$, $\veebar$, and $\neg$, as well as their bitwise counterparts. The property that we will use in designing our configurable algorithm is that these operators do not introduce dependencies between the bit positions, i.e., an output bit at position n depends only on the input bits at position n. We exploit this property in the recursive formulation of the algorithm, where we assume that we only compute one bit at position n, given that we already have a result for the bits at positions $p \leq n$.

Algorithm `algo`$(A_{\{n\}}$, $B_{\{n\}}$, $\circ$`)` **is**
  Input: Operand BDDs representing n-bit integer sets $A_{\{n\}}$ and $B_{\{n\}}$, binary Boolean operator $\circ$
  Result: BDD-based integer set containing $\{a \circ b \mid a \in A_{\{n\}}, b \in B_{\{n\}}\}$
  Function `extract`(i) **is**
1 **switch** i **do**
2 **case** $\boxed{1}$ **do return** $(\boxed{1}, \boxed{1})$
3 **case** $\boxed{0}$ **do return** $(\boxed{0}, \boxed{0})$
4 **case** $(a \boxed{\circ} b)$ **do return** (a, b)
  end
  end
5 **switch** $(A_{\{n\}} \boxed{\circ} B_{\{n\}})$ **do**
6 **case** $(\boxed{0}, \_)$ **do return** $\boxed{0}$
7 **case** $(\_, \boxed{0})$ **do return** $\boxed{0}$
8 **case** $(\boxed{1}, \boxed{1})$ **do return** $\boxed{1}$
9 **case** (a, b) **do**
10 $(a_\mathrm{t}, a_\mathrm{f}) \coloneqq$ `extract(a)`
11 $(b_\mathrm{t}, b_\mathrm{f}) \coloneqq$ `extract(b)`
12 $c \coloneqq \{((a_\mathrm{t}, 1), (b_\mathrm{t}, 1)), ((a_\mathrm{t}, 1), (b_\mathrm{f}, 0)), ((a_\mathrm{f}, 0), (b_\mathrm{t}, 1)), ((a_\mathrm{f}, 0), (b_\mathrm{f}, 0))\}$
13 $t \coloneqq \biguplus\boxed{\cup}\{$`algo(`$x_\mathtt{a}$`,` $y_\mathtt{a}$`,` $\circ$`)` $\mid ((x_\mathrm{a}, x_\mathrm{b}), (y_\mathrm{a}, y_\mathrm{b})) \in c, x_\mathrm{b} \circ y_\mathrm{b}\}$
14 $f \coloneqq \biguplus\boxed{\cup}\{$`algo(`$x_\mathtt{a}$`,` $y_\mathtt{a}$`,` $\circ$`)` $\mid ((x_\mathrm{a}, x_\mathrm{b}), (y_\mathrm{a}, y_\mathrm{b})) \in c, \neg(x_\mathrm{b} \circ y_\mathrm{b})\}$
15 **return** $(t \boxed{\circ} f)$
  end
  end
end

Algorithm 8: Configurable Bitwise Algorithm

Algorithm 8 stops under the following conditions:

1. Either operand is the terminal false node (Lines 6 and 7)
2. Both operands are the terminal true node (Line 8)

In Case 1, the terminal false operand represents an empty integer set and, therefore, there is no combination of integers that the bitwise operation needs to be applied to to form the result. In this case, the result is therefore the terminal false node.

In Case 2, the terminal true operands represent a n-bit integer set that is full, meaning that it consists of all n-bit integers. All required operations will produce all n-bit integers if applied to all combinations of all n-bit integers, because the supported Boolean operators will produce both 0 and 1, if applied to all possible inputs. Therefore, the result is the terminal true node.

If only one operand is the terminal true node, then it is treated as if it would be a non-reduced BDD internal node with both successors set to the terminal true node, which is done in the `extract` function in Lines 1 to 4.

The only missing case is when both operands are non-terminal. In this case (Lines 9 to 15), each operand node can be deconstructed into a true and false successor sub-BDD using the `extract` function. The successor sub-BDDs under the true edge, i.e., a_t and b_t in this case, represent the sets of integers with the corresponding bit set, and the false successors, i.e., a_f, and b_f, represent the sets with the bit unset. Since the algorithm must compute a result in Cartesian product fashion, it must consider all combinations of the successors of both inputs $A_{\{n\}}$ and $B_{\{n\}}$, hence the construction of c in Line 12. This c not only contains tuples of all combinations of sub-BDDs, but additionally a Boolean that indicates whether this sub-BDD was a true or a false successor, which is equivalent to the value of the bit at the current position. These additional Booleans are used in Lines 13 and 14, where the true and false successor of the result node are constructed. The construction of the true successor, in Line 13, only uses those combinations from c where the given Boolean operator $\circ$, applied to the corresponding Booleans, yields true. The algorithm is called recursively on all these combinations and the results of these calls are merged using integer set union, i.e., a disjunction at BDD level. The construction of the false successor in Line 14 applies the same operations on those combinations from c where the Boolean operator yields false.

In the following, we will compute $\texttt{algo}(A_2,\ B_2,\ \wedge)$, where $A_2 = \{0, 3\}$ and $B_2 = \{1, 2\}$. Figure 4.8 lists the corresponding BDDs of the two bit integer sets.

$$
\begin{aligned}
&\texttt{algo}(A_{\{2\}},\ B_{\{2\}},\ \wedge) = \\
&\quad (\texttt{algo}(A_{\{1\}},\ B_{\{1\}},\ \wedge) \boxed{\circ} \\
&\quad \biguplus_{\boxed{\cdot}} \{\texttt{algo}(A_{\{1\}},\ B'_{\{1\}},\ \wedge), \texttt{algo}(A'_{\{1\}},\ B_{\{1\}},\ \wedge), \texttt{algo}(A'_{\{1\}},\ B'_{\{1\}},\ \wedge)\})
\end{aligned}
$$

In the first recursive step, only the combination $(A_{\{1\}}, B_{\{1\}})$ gets selected for the true successor of the result node, since only these BDDs are true successors and therefore correspond to sub-BDDs with an incoming true edge, and $a \wedge b$ is only true if both a and b are; all other combinations are selected for the false successor of the result node.

$$
\begin{aligned}
&\texttt{algo}(A_{\{1\}},\ B_{\{1\}},\ \wedge) = \\
&\quad (\texttt{algo}(\boxed{1},\ \boxed{0},\ \wedge) \boxed{\circ} \biguplus_{\boxed{\cdot}} \{\texttt{algo}(\boxed{1},\ \boxed{1},\ \wedge), \texttt{algo}(\boxed{0},\ \boxed{0},\ \wedge), \texttt{algo}(\boxed{0},\ \boxed{1},\ \wedge)\}) = \\
&\quad (\boxed{0} \boxed{\circ} \boxed{1})
\end{aligned}
$$

With the above result, the algorithm has computed that 2 is included in the result set, since $(\boxed{0} \boxed{\circ} \boxed{1})$ is set under the true edge of the result node. In the following, the algorithm computes the node for the false successor of the result node.

$$
\begin{aligned}
&\texttt{algo}(A_{\{1\}},\ B'_{\{1\}},\ \wedge) = \\
&\quad (\texttt{algo}(\boxed{1},\ \boxed{1},\ \wedge) \boxed{\circ} \biguplus_{\boxed{\cdot}} \{\texttt{algo}(\boxed{1},\ \boxed{0},\ \wedge), \texttt{algo}(\boxed{0},\ \boxed{1},\ \wedge), \texttt{algo}(\boxed{0},\ \boxed{0},\ \wedge)\}) = \\
&\quad (\boxed{1} \boxed{\circ} \boxed{0})
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{algo}(A'_{\{1\}},\ B_{\{1\}},\ \wedge) = \\
&\quad (\texttt{algo}(\boxed{0},\ \boxed{0},\ \wedge) \boxed{\circ} \biguplus_{\boxed{\cdot}} \{\texttt{algo}(\boxed{0},\ \boxed{1},\ \wedge), \texttt{algo}(\boxed{1},\ \boxed{0},\ \wedge), \texttt{algo}(\boxed{1},\ \boxed{1},\ \wedge)\}) = \\
&\quad (\boxed{0} \boxed{\circ} \boxed{1})
\end{aligned}
$$

$$\begin{aligned}&\texttt{algo}(A'_{\{1\}}\ ,\ B'_{\{1\}}\ ,\ \wedge) =\\ &\quad(\texttt{algo}(\boxed{0},\ \boxed{1},\ \wedge) \boxdot \biguplus \{\texttt{algo}(\boxed{0},\ \boxed{0},\ \wedge), \texttt{algo}(\boxed{1},\ \boxed{1},\ \wedge), \texttt{algo}(\boxed{1},\ \boxed{0},\ \wedge)\}) =\\ &\quad(\boxed{0} \boxdot \boxed{1})\end{aligned}$$

The false successor of the outer result node is the union of all of the above three intermediate results, i.e., $\boxed{1}$. Hence, the result of the computation is $\{2, 1, 0\}$, just as expected.

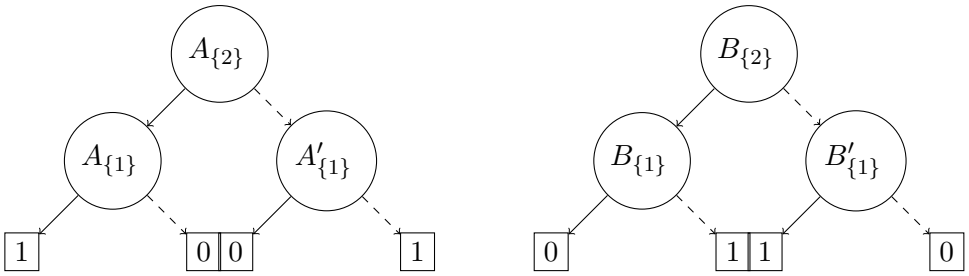


Figure 4.8.: BDDs of $A_{\{2\}}$ on the left, and $B_{\{2\}}$ on the right

Optimization

Algorithm 8 produces the desired results, but it can be improved in terms of efficiency. If c in Line 12 contains two tuples where the BDDs are equal and only the bit information differs, the recursive call must only be applied once and its result can be copied for the other tuple. This optimization is included in the memoization technique. Further, all of the targeted Boolean operators are commutative. Hence, c can be reduced by removing tuples that differ in the order the BDDs appear. This optimization can also be implemented using memoization by checking whether a call to the algorithm with flipped arguments is cached. If these optimizations are to be implemented using memoization it must be ensured that cache entries outlive the call of the algorithm, since otherwise, it could be that cached results are dropped before becoming useful in the next recursive algorithm call.

4.5. Arithmetic Operations

In difference to the bitwise variations of the binary Boolean operators discussed before, in arithmetic operations, an output bit at position n is not uniquely determined by the input bits at position n. Therefore, Algorithm 8 is not directly applicable for addition. In this section, we therefore adapt Algorithm 8 to addition and subtraction, and provide approximating algorithms for multiplication and division, as well as precise versions for special cases.

Addition

Algorithm 8 takes one of the common binary Boolean operators, and computes the bitwise version of the operator on integer sets. The corresponding Boolean operator for addition is the full adder, which we will use to design an algorithm that supports addition. Since overflow behavior is defined on machine code level, this algorithm must precisely simulate overflow.

A full adder takes three Boolean inputs, and produces two boolean outputs, i.e., sum and carry. Equation 4.4 defines the sum (s) and carry (c_{out}) outputs depending on the input Booleans a, b, and c_{in}.

$$\begin{aligned} s(a, b, c_{\text{in}}) &= a \veebar b \veebar c_{\text{in}} \\ c_{\text{out}}(a, b, c_{\text{in}}) &= (a \wedge b) \vee b(\wedge c_{\text{in}}) \vee (a \wedge c_{\text{in}}) \end{aligned} \tag{4.4}$$

To produce an n-bit adder, the full adders are combined as described in Equation 4.5.

$$\begin{aligned} c_{\{0\}} &= 0 \\ c_{\{i\}} &= c_{\text{out}}(a_{\{i-1\}}, b_{\{i-1\}}, c_{\{i-1\}}) \\ o_{\{i\}} &= s(a_{\{i\}}, b_{\{i\}}, c_{\{i\}}) \end{aligned} \tag{4.5}$$

From these equations, we know that to determine a bit at position i, we need not only know the two input bits $a_{\{i\}}$ and $b_{\{i\}}$, but also the carry information as computed from position $i-1$. Therefore, each recursive call must additionally produce carry information. We modify the algorithm to return two BDD-based integer sets instead of one. One set represents the resulting integer sets without values that are the result from overflow, and one set represents only values that result from overflow. This modification allows the communication of the carry information as required, since values that result from overflow are values that have the carry bit set. Unioning the carry and non-carry result produces a correct result if traditional wrap-around semantics is assumed. Since the carry information from bit position n influences the calculation for bit position $n+1$, it can be said to propagate from LSB to MSB. The formulation of an algorithm for addition and subtraction would therefore benefit from LSB to MSB ordering in the BDD. However, our algorithm must support MSB to LSB ordering for the reasons given in Section 4.2.

Algorithm 9 is an adaptation of Algorithm 8 to support a full adder as operator. The algorithm uses the `extract` function from Algorithm 8, as well as the `ival2bdd` function (Algorithm 4) that takes a lower and an upper bound and creates a BDD integer set containing elements within the bounds. In the base cases where one input BDD is $\boxed{0}$ (Lines 2 and 3), the algorithm returns $\boxed{0}$ for the non-overflow BDD in the first element of the returned tuple, and $\boxed{0}$ as the overflow BDD in the second element of the tuple. The reason for returning $\boxed{0}$ is that one of the input integer sets does not contain any elements, therefore the Cartesian product is empty.

In case both input BDDs are $\boxed{1}$ (line 4), i.e., the represented n-bit integer set contains all n-bit integers, then the non-overflow BDD must contain all n-bit integers as well because one such integer set contains 0 and $\{0\} +^{\#} \{i \mid 0 \leq i < 2^n\}$ is $\{i \mid 0 \leq i < 2^n\}$.

The overflow result contains all but the largest n-bit integer, since the largest overflow result is $2^n - 1 + 2^n - 1 = 2^{n+1} - 2$ and the MSB is treated as unset to simulate wrap-around.

The recursive case is in Lines 8 to 18. First, the true and false successors of the roots of the input BDDs are extracted (Lines 9 and 10). Similarly to Algorithm 8, all combinations of the successors are built in Lines 11 to 13. In difference to Algorithm 8, each recursive call generates two results, one non-overflow result and one overflow result, which both get stored in c_{ab} together with the input bit information and carry information. Lines 14 to 17 compute the successor for the overflow and non-overflow results by applying the given operator, which for addition is the full adder, and selecting and merging the corresponding BDDs.

Algorithm `add(`$A_{\{n\}}$ `,` $B_{\{n\}}$ `,` $\circ$`,` $d = n$`)` **is**
  Input: Operand BDDs representing n-bit integer sets $A_{\{n\}}$ and $B_{\{n\}}$, ternary Boolean operator $\circ$, depth d
  Result: BDD-based integer sets containing overflow and non-overflow results
1 **switch** $(A_{\{n\}}, B_{\{n\}})$ **do**
2 **case** $(\boxed{0}, \_)$ **do return** $(\boxed{0}, \boxed{0})$
3 **case** $(\_, \boxed{0})$ **do return** $(\boxed{0}, \boxed{0})$
4 **case** $(\boxed{1}, \boxed{1})$ **do**
5 **if** $d = 0$ **then**
6 **return** $(\boxed{1}, \boxed{0})$
else
7 **return** $(\boxed{1}, \texttt{ival2bdd(0, } 2^d - 2\texttt{)})$
end
end
8 **case** (a, b) **do**
9 $(a_{\text{t}}, a_{\text{f}}) \coloneqq \texttt{extract(a)}$
10 $(b_{\text{t}}, b_{\text{f}}) \coloneqq \texttt{extract(b)}$
11 $c_{\text{a}} \coloneqq \{(a_{\text{t}}, 1), (a_{\text{f}}, 0)\}$
12 $c_{\text{b}} \coloneqq \{(b_{\text{t}}, 1), (b_{\text{f}}, 0)\}$
13 $c_{\text{ab}} \coloneqq \bigcup\{\{(z, y_{\text{a}}, y_{\text{b}}, 0), (z_{\text{c}}, y_{\text{a}}, y_{\text{b}}, 1)\} \mid (x_{\text{a}}, y_{\text{a}}) \in c_{\text{a}}, (x_{\text{b}}, y_{\text{b}}) \in c_{\text{b}}, (z, z_{\text{c}}) = \texttt{add(}x_{\texttt{a}}\texttt{, } x_{\texttt{b}}\texttt{, } \circ\texttt{, } d = d - 1\texttt{)}\}$
14 $t \coloneqq \biguplus\{z \mid (z, y_{\text{a}}, y_{\text{b}}, c) \in c_{\text{ab}}, (r, c_{\text{r}}) = \circ(y_{\text{a}}, y_{\text{b}}, c), r, \neg c_{\text{r}}\}$
15 $t_{\text{c}} \coloneqq \biguplus\{z \mid (z, y_{\text{a}}, y_{\text{b}}, c) \in c_{\text{ab}}, (r, c_{\text{r}}) = \circ(y_{\text{a}}, y_{\text{b}}, c), r, c_{\text{r}}\}$
16 $f \coloneqq \biguplus\{z \mid (z, y_{\text{a}}, y_{\text{b}}, c) \in c_{\text{ab}}, (r, c_{\text{r}}) = \circ(y_{\text{a}}, y_{\text{b}}, c), \neg r, \neg c_{\text{r}}\}$
17 $f_{\text{c}} \coloneqq \biguplus\{z \mid (z, y_{\text{a}}, y_{\text{b}}, c) \in c_{\text{ab}}, (r, c_{\text{r}}) = \circ(y_{\text{a}}, y_{\text{b}}, c), \neg r, c_{\text{r}}\}$
18 **return** $((t \boxed{\circ} f), (t_{\text{c}} \boxed{\circ} f_{\text{c}}))$
end
end
end

Algorithm 9: Algorithm Supporting Addition and Subtraction

As an example, we use Algorithm 9 to compute $A_{\{1\}} +^{\#} B_{\{1\}}$, where $A_{\{1\}} = \{1\}$ and $B_{\{1\}} = \{0, 1\}$. In 1-bit unsigned arithmetic with wrap-around, we would expect the result of the above computation to be $\{0, 1\}$, since $1 + 0 = 1$ and $1 + 1 = 2$, which is 0 in one bit arithmetic with overflow. We therefore expect the algorithm to return $(\{1\}, \{0\})$, since $\{1\}$ is the non-overflow part of the result and $\{0\}$ is the overflow part of the result. Unioning of the overflow and non-overflow results will obtain the desired result with overflow simulation. The BDDs of $A_{\{1\}}$ and $B_{\{1\}}$ are shown in Figure 4.9. Note that $B_{\{1\}}$ is the terminal true node, since it represents the set of all 1-bit integers.

Figure 4.9.: BDDs of $A_{\{1\}}$ on the left, and $B_{\{1\}}$ on the right

Algorithm 9, applied to $A_{\{1\}}$ and $B_{\{1\}}$, proceeds as follows:

$$
\begin{aligned}
&\texttt{add}(A_{\{1\}}\texttt{,}\ B_{\{1\}}\texttt{,}\ \texttt{fAdder()},\ d = 1) = \\
&\quad c_{\text{ab}} \coloneqq \{(z_{11}, 1, 1, 0), (z_{11\text{c}}, 1, 1, 1), (z_{10}, 1, 0, 0), (z_{10\text{c}}, 1, 0, 1) \\
&\qquad\quad , (z_{01}, 0, 1, 0), (z_{01\text{c}}, 0, 1, 1), (z_{00}, 0, 0, 0), (z_{00\text{c}}, 0, 0, 1) \\
&\qquad\quad \mid (z_{11}, z_{11\text{c}}) = \texttt{add}(\boxed{1}\texttt{,}\ \boxed{1}\texttt{,}\ \texttt{fAdder()},\ d = 0) = (\boxed{1}, \boxed{0}) \\
&\qquad\quad , (z_{10}, z_{10\text{c}}) = \texttt{add}(\boxed{1}\texttt{,}\ \boxed{1}\texttt{,}\ \texttt{fAdder()},\ d = 0) = (\boxed{1}, \boxed{0}) \\
&\qquad\quad , (z_{01}, z_{01\text{c}}) = \texttt{add}(\boxed{0}\texttt{,}\ \boxed{1}\texttt{,}\ \texttt{fAdder()},\ d = 0) = (\boxed{0}, \boxed{0}) \\
&\qquad\quad , (z_{00}, z_{00\text{c}}) = \texttt{add}(\boxed{0}\texttt{,}\ \boxed{1}\texttt{,}\ \texttt{fAdder()},\ d = 0)\} = (\boxed{0}, \boxed{0}) \\
&\qquad = \{(\boxed{1}, 1, 1, 0), (\boxed{0}, 1, 1, 1), (\boxed{1}, 1, 0, 0), (\boxed{0}, 1, 0, 1) \\
&\qquad\quad , (\boxed{0}, 0, 1, 0), (\boxed{0}, 0, 1, 1), (\boxed{0}, 0, 0, 0), (\boxed{0}, 0, 0, 1)\} \\
&\quad t \coloneqq \biguplus\{\boxed{1}, \boxed{0}, \boxed{0}\} = \boxed{1} \\
&\quad t_{\text{c}} \coloneqq \biguplus\{\boxed{0}\} = \boxed{0} \\
&\quad f \coloneqq \biguplus\{\boxed{0}\} = \boxed{0} \\
&\quad f_{\text{c}} \coloneqq \biguplus\{\boxed{1}, \boxed{0}, \boxed{0}\} = \boxed{1} \\
&\quad \textbf{return}((\boxed{1} \boxed{\circ} \boxed{0}), (\boxed{0} \boxed{\circ} \boxed{1}))
\end{aligned}
$$

When applied to $A_{\{1\}}$, $B_{\{1\}}$, and depth 1 corresponding to the bitwidth of the input integer sets, the algorithm executes the recursive switch case (Lines 8 to 18). Consequently, the successors of the input BDDs are extracted and all required combinations are built with the corresponding results of the recursive addition calls in c_{ab}, while retaining the information from which edges the combination was built. The recursive addition calls with $\boxed{0}$ as one of the arguments will return $(\boxed{0}, \boxed{0})$ according to the switch cases in Lines 2 and 3. The recursive addition calls with $\boxed{1}$ for both arguments will return $(\boxed{1}, \boxed{0})$ according to the switch case in Lines 4 to 7 since d is 0. The non-overflow true edge BDD (t) gets constructed from those tuples in c_{ab} that have one set bit in the last three tuple elements, because only then will the full adder return 1 for the sum and 0 for the

carry. Likewise, only the tuple with all bits set is selected for t_c, since only then will the full adder return 1 for the sum and carry. The non-overflow false edge BDD (f) is built from the tuple with all bits set to 0, all other tuples are selected for the remaining false edge BDD of the overflow result. Hence, the result is $((\boxed{1}\boxdot\boxed{0}), (\boxed{0}\boxdot\boxed{1}))$, which is the expected $(\{1\}, \{0\})$ in integer set-interpretation.

Optimizations

Algorithm 9 can be optimized, but one must be careful not to change the function of the algorithm. For example, it may be tempting to replace the case where both input BDDs are equal, which can, of course, be checked in $O(1)$, by multiplication of two or shift left of one. However, this would be incorrect, since the algorithm must compute its result in the previously outlined Cartesian product fashion. In other words, equality of two variation domains does not imply that the corresponding variables, for which the VDs were computed, are equal. Therefore, such optimization can only be done outside of the algorithm, if equality can be proven.

More optimization potential can be found in the unification of the BDDs for t and f_c. The full adder will always select three BDDs for t and f_c, since there are three inputs with one set bit, and also three inputs with two set bits. Therefore, instead of defining unification of a set of BDD-based integer sets using repeated application of binary unification, which requires two BDD traversals, we specialize union of three BDD-based integers sets in Algorithm 10.

Algorithm `merge3`$(A,\ B,\ C)$ **is**
  Input: Three BDD-based integer sets
  Result: Union of given BDD-based integer sets
1 **switch** (A, B, C) **do**
2 **case** $(\boxed{1}, \_, \_)$ **do return** $\boxed{1}$
3 **case** $(\_, \boxed{1}, \_)$ **do return** $\boxed{1}$
4 **case** $(\_, \_, \boxed{1})$ **do return** $\boxed{1}$
5 **case** $(\boxed{0}, \_, \_)$ **do return** $B \cup C$
6 **case** $(\_, \boxed{0}, \_)$ **do return** $A \cup C$
7 **case** $(\_, \_, \boxed{0})$ **do return** $A \cup B$
8 **case** $((A_a \boxdot A_b), (B_a \boxdot B_b), (C_a \boxdot C_b))$ **do**
9 $t \coloneqq$ `merge3`$(A_a,\ B_a,\ C_a)$
10 $f \coloneqq$ `merge3`$(A_b,\ B_b,\ C_b)$
11 **return** $(t \boxdot f)$
  end
end

Algorithm 10: Algorithm for Unification of Three BDD-based Integer Sets

Another substantial optimization for Algorithm 9 is the addition of base cases specific to addition. Currently, the algorithm only stops its recursion if at least one operand

is $\boxed{0}$, or both operands are $\boxed{1}$. However, it is possible to extend the algorithm to also stop its recursion when at least one operand is $\boxed{1}$. Whenever one of the arguments is $\boxed{1}$, this $\boxed{1}$ represents an interval of all n-bit integers ($A = [0\,..\,2^n - 1]$), whereas the other represents a set of n-bit integers B. Since A is the set of all n bit integers, we conclude $B \subseteq A$. The distance between any two elements in B is less than the width of A, and hence all intervals that result from adding a number from B to each element in A will overlap. Therefore, the non-overflow result will be an interval, starting at the least element in B (`min(B)` = `min(B)` + `min(A)`) to the maximum n-bit integer (`max(A)`). The overflow result will be an interval from 0 to the greatest element in B minus one, since `max(B)` + `max(A)` = `max(B)` + $2^n - 1$ = `max(B)` $- 1 (\texttt{mod}\ 2^n)$. Algorithm 11 lists the optimized addition algorithm.

Algorithm `add`($A_{\{n\}}$, $B_{\{n\}}$, d) **is**
  Input: Operand BDDs representing n-bit integer sets $A_{\{n\}}$ and $B_{\{n\}}$
  Result: BDD-based integer sets containing overflow and non-overflow results
1 **switch** $(A_{\{n\}}, B_{\{n\}})$ **do**
2 **case** $(\boxed{0}, \_)$ **do return** $(\boxed{0}, \boxed{0})$
3 **case** $(\_, \boxed{0})$ **do return** $(\boxed{0}, \boxed{0})$
4 **case** $(\boxed{1}, \_)$ **do return**
   (`ival2bdd(min(`$B_{\{n\}}$`), max(`$A_{\{n\}}$`)), ival2bdd(0, max(`$B_{\{n\}}$`) − 1))`
5 **case** $(\_, \boxed{1})$ **do return** `add`($B_{\{n\}}$, $A_{\{n\}}$, d)
6 **case** $((A_{\mathtt{t}} \boxed{\circ} A_{\mathtt{f}}), (B_{\mathtt{t}} \boxed{\circ} B_{\mathtt{f}}))$ **do**
7 $(x_{11}, x_{\mathrm{c}11}) \coloneqq$ `add`($A_{\mathtt{t}}$, $B_{\mathtt{t}}$, $d-1$)
8 $(x_{10}, x_{\mathrm{c}10}) \coloneqq$ `add`($A_{\mathtt{t}}$, $B_{\mathtt{f}}$, $d-1$)
9 $(x_{01}, x_{\mathrm{c}01}) \coloneqq$ `add`($A_{\mathtt{f}}$, $B_{\mathtt{t}}$, $d-1$)
10 $(x_{00}, x_{\mathrm{c}00}) \coloneqq$ `add`($A_{\mathtt{f}}$, $B_{\mathtt{f}}$, $d-1$)
11 $t \coloneqq$ `merge3`(x_{10}, x_{01}, $x_{\mathrm{c}00}$)
12 $t_{\mathrm{c}} \coloneqq x_{\mathrm{c}11}$
13 $f \coloneqq x_{00}$
14 $f_{\mathrm{c}} \coloneqq$ `merge3`(x_{11}, $x_{\mathrm{c}10}$, $x_{\mathrm{c}01}$)
15 **return** $((t \boxed{\circ} f), (t_{\mathrm{c}} \boxed{\circ} f_{\mathrm{c}}))$
  end
end

Algorithm 11: Optimized Algorithm for Addition

Correctness of Addition

In this paragraph, we will discuss the correctness of Algorithm 11. We will assume correctness of the `ival2bdd` algorithm that constructs BDD representation of intervals, as well as correctness of `merge3`, which unifies three BDD-based sets. In the proof, we use the induction scheme as described in Section 4.1.2.

First, let us define the correctness property for addition (P in the induction scheme).

As previously discussed, abstract interpretation demands that if a correctness relation exists between the concrete and abstract values for the input of an operation, then the outputs of the corresponding transfer function must also be correct with respect to the output of the operation. For the addition transfer function on non-relational, unsigned n-bit integer sets $A_{\{n\}}$ and $B_{\{n\}}$ this requirement applies as well. However, our addition algorithm returns a tuple of two BDDs, one for the non-overflow and one for the overflow result. Therefore, we first define the functions h and g that convert between a tuple of non-overflow ($R_{\{n\}}^{\overline{\texttt{carry}}}$) and overflow result ($R_{\{n\}}^{\texttt{carry}}$) and addition via the Cartesian product:

$$h(R_{\{n+1\}}) = (\{r \mid r \in R_{\{n+1\}}, r < 2^n\}, \{r - 2^n \mid r \in R_{\{n+1\}}, r \geq 2^n\})$$
$$g((R_{\{n\}}^{\overline{\texttt{carry}}}, R_{\{n\}}^{\texttt{carry}})) = R_{\{n\}}^{\overline{\texttt{carry}}} \cup \{r + 2^n \mid r \in R_{\{n\}}^{\texttt{carry}}\}$$

Next, we prove that h and g together form a bijection by showing $h \circ g = \text{id}$ and $g \circ h = \text{id}$.

Assuming $R = L \cup B$ with $\forall l \in L : 0 \leq l < 2^n$ and $\forall b \in B : 2^n \leq b < 2^{n+1}$ we get:
$h(R) = h(L \cup B) = (L, \{r - 2^n \mid r \in B\})$ and
$g((L, \{r - 2^n \mid r \in B\}))$
$= L \cup \{r' + 2^n \mid r' \in \{r - 2^n \mid r \in B\}\}$
$= L \cup \{r' \mid r' \in B\}$
$= L \cup B$
$= R$
Hence, $g \circ h = \text{id}$.

Assuming L and B with $\forall l \in L : 0 \leq l < 2^n$ and $\forall b \in B : 0 \leq b < 2^n$ we get:
$g((L, B)) = g((\{l \mid l \in L, 0 \leq l < 2^n\}, \{b \mid b \in B, 0 \leq b < 2^n\}))$
$= \{l \mid l \in L, 0 \leq l < 2^n\} \cup \{r + 2^n \mid r \in \{b \mid b \in B, 0 \leq b < 2^n\}\}$
$= \{l \mid l \in L, 0 \leq l < 2^n\} \cup \{b \mid b \in \{r + 2^n \mid r \in B, 0 \leq r < 2^n\}, 2^n \leq b < 2^{n+1}\}$
$h(\{l \mid l \in L, 0 \leq l < 2^n\} \cup \{b \mid b \in \{r + 2^n \mid r \in B, 0 \leq r < 2^n\}, 2^n \leq b < 2^{n+1}\})$
$= (L, \{r - 2^n \mid r \in \{b + 2^n \mid b \in B\}\})$
$= (L, \{r - 2^n + 2^n \mid r \in B\})$
$= (L, B)$
Hence, $h \circ g = \text{id}$.

■

We further observe the following equalities:
$(A_{\{n\}} \boxdot B_{\{n\}}) = \{a + 2^n \mid a \in A_{\{n\}}\} \cup B_{\{n\}} = g((B_{\{n\}}, A_{\{n\}}))$
$\boxed{0}_{\{n\}} = \emptyset$
$\boxed{1}_{\{n\}} = \{e \mid 0 \leq e < 2^n\}$
$\texttt{merge3}(A_{\{n\}}, B_{\{n\}}, C_{\{n\}}) = A_{\{n\}} \cup B_{\{n\}} \cup C_{\{n\}}$

The correctness property P for addition is defined as follows:

$$P(X_{\{n\}}, Y_{\{n\}}) \iff h(\{x + y \mid x \in X_{\{n\}}, y \in Y_{\{n\}}\}) = \texttt{add}(X_{\{n\}}, Y_{\{n\}}, n)$$

In other words, if addition via the Cartesian product is executed and the result, using h, is partitioned into values that fit into the original n-bit range and those that do not, then this partition must be returned by the addition algorithm. In the following, we prove P for arbitrary tuples of BDDs using the induction scheme described in Section 4.1.2. Hence, we have to prove the following:

Base Cases:

$$\forall A_{\{n\}} : P(A_{\{n\}}, \boxed{0}) \tag{4.6}$$

$$\forall B_{\{n\}} : P(\boxed{0}, B_{\{n\}}) \tag{4.7}$$

$$\forall A_{\{n\}} : P(A_{\{n\}}, \boxed{1}) \tag{4.8}$$

$$\forall B_{\{n\}} : P(\boxed{1}, B_{\{n\}}) \tag{4.9}$$

Inductive Case:

$$\begin{aligned} &P(A_{\{n\}}, C_{\{n\}}) \wedge P(A_{\{n\}}, D_{\{n\}}) \wedge P(B_{\{n\}}, C_{\{n\}}) \wedge P(B_{\{n\}}, D_{\{n\}}) \\ &\quad \implies P((A_{\{n\}} \boxed{\circ} B_{\{n\}}), (C_{\{n\}} \boxed{\circ} D_{\{n\}})) \end{aligned} \tag{4.10}$$

Proofs

Proof of 4.6:

$h(\{a + b \mid a \in A_{\{n\}}, b \in \boxed{0}\}) = \texttt{add}(A_{\{n\}}, \boxed{0}, n)$
$\implies \emptyset = \texttt{add}(A_{\{n\}} \boxed{0}, n)$
$\implies \emptyset = \boxed{0}_{\{n\}}$
$\implies \emptyset = \emptyset$

Proof 4.7: Analog to $\forall A_{\{n\}} : P(A_{\{n\}}, \boxed{0})$.

Proof of 4.8:

$h(\{a+b \mid a \in A_{\{n\}}, b \in \boxed{1}\}) = \texttt{add}(A_{\{n\}}, \boxed{1}, n)$
$\iff h(\{a+b \mid a \in A_{\{n\}}, b \in \boxed{1}_{\{n\}}\}) = \texttt{add}(A_{\{n\}}, \boxed{1}, n)$
$\iff h(\{a+b \mid a \in A_{\{n\}}, b \in \{e \mid 0 \leq e < 2^n\}\}) = \texttt{add}(A_{\{n\}}, \boxed{1}, n)$

Hence, $A_{\{n\}} \in \{e \mid 0 \leq e < 2^n\}$, and since $\{e \mid 0 \leq e < 2^n\}$ is convex, $\{a+b \mid a \in A_{\{n\}}, b \in \{e \mid 0 \leq e < 2^n\}\}$ is convex as well.
Setting $X_{\{n+1\}} = \{a+b \mid a \in A_{\{n\}}, b \in \{e \mid 0 \leq e < 2^n\}\}$ we get:
$h(\{e \mid \texttt{min}(X_{\{n+1\}}) \leq e \leq \texttt{max}(X_{\{n+1\}})\}) = \texttt{add}(A_{\{n\}}, \boxed{1}, n)$
$\iff h(\{e \mid \texttt{min}(A_{\{n\}}) \leq e \leq \texttt{max}(A_{\{n\}}) + 2^n - 1\}) = \texttt{add}(A_{\{n\}}, \boxed{1}, n)$
$\iff \{e \mid \texttt{min}(A_{\{n\}}) \leq e \leq \texttt{max}(A_{\{n\}}) + 2^n - 1\} = g(\texttt{add}(A_{\{n\}}, \boxed{1}, n))$

Assuming correctness of `ival2bdd`, we have:
$g(\texttt{add}(A_{\{n\}}, \boxed{1}, n))$
$= g((\texttt{ival2bdd}(\texttt{min}(A_{\{n\}}), \texttt{max}(\boxed{1})), \texttt{ival2bdd}(0, \texttt{max}(A_{\{n\}}) - 1)))$
$= g((\texttt{ival2bdd}(\texttt{min}(A_{\{n\}}), 2^n - 1), \texttt{ival2bdd}(0, \texttt{max}(A_{\{n\}}) - 1)))$
$= g((\{e \mid \texttt{min}(A_{\{n\}}) \leq e \leq 2^n - 1\}, \{e \mid 0 \leq e \leq \texttt{max}(A_{\{n\}}) - 1\}))$
$= \{e \mid \texttt{min}(A_{\{n\}}) \leq e \leq 2^n - 1\} \cup \{e + 2^n \mid 0 \leq e \leq \texttt{max}(A_{\{n\}}) - 1\}$
$= \{e \mid \texttt{min}(A_{\{n\}}) \leq e \leq 2^n - 1\} \cup \{e \mid 2^n \leq e \leq \texttt{max}(A_{\{n\}}) + 2^n - 1\}$
$= \{e \mid \texttt{min}(A_{\{n\}}) \leq e \leq \texttt{max}(A_{\{n\}}) + 2^n - 1\}$

Proof of 4.9: Analog to $\forall A_{\{n\}} : P(A_{\{n\}}, \boxed{1})$.

Inductive Case

Proof of 4.10:

$$
\begin{array}{lll}
(& & P(A_{\{n\}}, C_{\{n\}}) \wedge P(A_{\{n\}}, D_{\{n\}}) \wedge P(B_{\{n\}}, C_{\{n\}}) \wedge P(B_{\{n\}}, D_{\{n\}}) \\
\implies & & P((A_{\{n\}} \boxed{\circ} B_{\{n\}}), (C_{\{n\}} \boxed{\circ} D_{\{n\}})) \\
) \iff (& & h(\{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}) = \texttt{add}(A_{\{n\}}, C_{\{n\}}, n) \\
 & & \wedge h(\{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}) = \texttt{add}(A_{\{n\}}, D_{\{n\}}, n) \\
 & & \wedge h(\{b+c \mid b \in B_{\{n\}}, c \in C_{\{n\}}\}) = \texttt{add}(B_{\{n\}}, C_{\{n\}}, n) \\
 & & \wedge h(\{b+d \mid b \in B_{\{n\}}, d \in D_{\{n\}}\}) = \texttt{add}(B_{\{n\}}, D_{\{n\}}, n) \\
\implies & & h(\{s+u \mid s \in (A_{\{n\}} \boxed{\circ} B_{\{n\}}), u \in (C_{\{n\}} \boxed{\circ} D_{\{n\}})\}) \\
 & & = \texttt{add}((A_{\{n\}} \boxed{\circ} B_{\{n\}}), (C_{\{n\}} \boxed{\circ} D_{\{n\}}), n+1) \\
) & &
\end{array}
\tag{4.11}
$$

First, we operate on the right-hand side of the consequent in Equation 4.11 and expand the recursive calls to `add` using the induction hypothesis:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\texttt{add}((A_{\{n\}} \boxed{\circ} B_{\{n\}}), (C_{\{n\}} \boxed{\circ} D_{\{n\}}), n+1) =} \\
& (x_{11}, x_{\mathrm{c}11}) \coloneqq & \texttt{add}(A_{\{n\}}, C_{\{n\}}, n+1) \\
& (x_{10}, x_{\mathrm{c}10}) \coloneqq & \texttt{add}(A_{\{n\}}, D_{\{n\}}, n+1) \\
& (x_{01}, x_{\mathrm{c}01}) \coloneqq & \texttt{add}(B_{\{n\}}, C_{\{n\}}, n+1) \\
& (x_{00}, x_{\mathrm{c}00}) \coloneqq & \texttt{add}(B_{\{n\}}, D_{\{n\}}, n+1) \\
= & (x_{11}, x_{\mathrm{c}11}) \coloneqq & h(\{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}) \\
& (x_{10}, x_{\mathrm{c}10}) \coloneqq & h(\{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}) \\
& (x_{01}, x_{\mathrm{c}01}) \coloneqq & h(\{b+c \mid b \in B_{\{n\}}, c \in C_{\{n\}}\}) \\
& (x_{00}, x_{\mathrm{c}00}) \coloneqq & h(\{b+d \mid b \in B_{\{n\}}, d \in D_{\{n\}}\}) \\
= & (x_{11}, x_{\mathrm{c}11}) \coloneqq & (\{r \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
& & , \{r - 2^n \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\}) \\
& (x_{10}, x_{\mathrm{c}10}) \coloneqq & (\{r \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
& & , \{r - 2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\}) \\
& (x_{01}, x_{\mathrm{c}01}) \coloneqq & (\{r \mid r \in \{b+c \mid b \in B_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
& & , \{r - 2^n \mid r \in \{b+c \mid a \in B_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\}) \\
& (x_{00}, x_{\mathrm{c}00}) \coloneqq & (\{r \mid r \in \{b+d \mid b \in B_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
& & , \{r - 2^n \mid r \in \{b+d \mid a \in B_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\})
\end{array}
\tag{4.12}
$$

After the expansion of the recursive calls to `add`, we now expand the definitions of t, t_c, f, and f_c in Equation 4.12, assuming a correct implementation of `merge3`:

$$
\begin{array}{lll}
& t \coloneqq & \texttt{merge3}(x_{10}, x_{01}, x_{\mathrm{c}00}) \\
& t_\mathrm{c} \coloneqq & x_{\mathrm{c}11} \\
& f \coloneqq & x_{00} \\
& f_\mathrm{c} \coloneqq & \texttt{merge3}(x_{11}, x_{\mathrm{c}10}, x_{\mathrm{c}01}) \\
= & t \coloneqq & x_{10} \cup x_{01} \cup x_{\mathrm{c}00} \\
& t_\mathrm{c} \coloneqq & x_{\mathrm{c}11} \\
& f \coloneqq & x_{00} \\
& f_\mathrm{c} \coloneqq & x_{11} \cup x_{\mathrm{c}10} \cup x_{\mathrm{c}01} \\
= & t \coloneqq & \{r \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
& & \cup \{r \mid r \in \{b+c \mid b \in B_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
& & \cup \{r - 2^n \mid r \in \{b+d \mid a \in B_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\} \\
& t_\mathrm{c} \coloneqq & \{r - 2^n \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\} \\
& f \coloneqq & \{r \mid r \in \{b+d \mid b \in B_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
& f_\mathrm{c} \coloneqq & \{r \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
& & \cup \{r - 2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\} \\
& & \cup \{r - 2^n \mid r \in \{b+c \mid a \in B_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\}
\end{array}
\tag{4.13}
$$

Next, we expand the non-overflow result, i.e., $((t \boxed{\circ} f))$, with t and f computed in Equation 4.13:

$$
\begin{aligned}
&(t \boxdot f) = g(f,t) = \\
&\quad \{r \mid r \in \{b+d \mid b \in B_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
&\quad \cup \{r' + 2^n \mid r' \in \\
&\quad \{r \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
&\quad \cup \{r \mid r \in \{b+c \mid b \in B_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
&\quad \cup \{r - 2^n \mid r \in \{b+d \mid a \in B_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\} \\
&\quad \} \\
&= \quad \{r \mid r \in \{b+d \mid b \in B_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
&\quad \cup \{r + 2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
&\quad \cup \{r + 2^n \mid r \in \{b+c \mid b \in B_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
&\quad \cup \{r \mid r \in \{b+d \mid a \in B_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\}
\end{aligned}
\tag{4.14}
$$

Now, we expand the overflow result, i.e., $((t_\mathrm{c} \boxdot f_\mathrm{c}))$, with t_c and f_c given by Equation 4.13:

$$
\begin{aligned}
&(t_\mathrm{c} \boxdot f_\mathrm{c}) = g(f_\mathrm{c}, t_\mathrm{c}) = \\
&\quad \{r \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
&\quad \cup \{r - 2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\} \\
&\quad \cup \{r - 2^n \mid r \in \{b+c \mid a \in B_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\} \\
&\quad \cup \{r - 2^n + 2^n \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\} \\
&= \quad \{r \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
&\quad \cup \{r - 2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\} \\
&\quad \cup \{r - 2^n \mid r \in \{b+c \mid a \in B_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\} \\
&\quad \cup \{r \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\}
\end{aligned}
\tag{4.15}
$$

In the following, we use the fact that $g \circ h = \mathrm{id}$ and insert the results, obtained using Equation 4.14 and 4.15, into the consequent of Equation 4.11:

$$
\begin{aligned}
& h(\{s+u \mid s \in (A_{\{n\}} \boxdot B_{\{n\}}), u \in (C_{\{n\}} \boxdot D_{\{n\}})\}) = \\
& \qquad \mathtt{add}((A_{\{n\}} \boxdot B_{\{n\}}), (C_{\{n\}} \boxdot D_{\{n\}}), n+1) \\
\iff \quad & \{s+u \mid s \in (A_{\{n\}} \boxdot B_{\{n\}}), u \in (C_{\{n\}} \boxdot D_{\{n\}})\} = \\
& \qquad g(\mathtt{add}((A_{\{n\}} \boxdot B_{\{n\}}), (C_{\{n\}} \boxdot D_{\{n\}}), n+1)) \\
\iff \quad & \{s+u \mid s \in (A_{\{n\}} \boxdot B_{\{n\}}), u \in (C_{\{n\}} \boxdot D_{\{n\}})\} = \\
& \qquad g((g(f,t), g(f_\mathrm{c}, t_\mathrm{c})))
\end{aligned}
\tag{4.16}
$$

Next, we expand and rewrite the right-hand side of Equation 4.16. Note that the colors denote elements of a partition, i.e., subsets $X \in Z$ and $Y \in Z$ such that $X \cap Y = \emptyset$ and $X \cup Y = Z$. Furthermore $\exists p : (\forall x \in X : p(x) \wedge \forall y \in Y : \neg p(y))$, whenever we may unify the colored sets and drop the predicate:

$$
\begin{aligned}
& g(g(f,t),g(f_{\mathrm{c}},t_{\mathrm{c}})) = \\
& \quad \{r \mid r \in \{b+d \mid b \in B_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r+2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r+2^n \mid r \in \{b+c \mid b \in B_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r \mid r \in \{b+d \mid a \in B_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\} \\
& \quad \cup \{r'+2^{n+1} \mid r' \in \\
& \quad \{r \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r-2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\} \\
& \quad \cup \{r-2^n \mid r \in \{b+c \mid a \in B_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\} \\
& \quad \cup \{r \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\} \\
& \quad \} \\
= \quad & \{r \mid r \in \{b+d \mid b \in B_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r+2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r+2^n \mid r \in \{b+c \mid b \in B_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r \mid r \in \{b+d \mid a \in B_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\} \\
& \quad \cup \{r+2^{n+1} \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r+2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\} \\
& \quad \cup \{r+2^n \mid r \in \{b+c \mid a \in B_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\} \\
& \quad \cup \{r+2^{n+1} \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\} \\
= \quad & \{r+2^{n+1} \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r+2^{n+1} \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\} \\
& \quad \cup \{r+2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r+2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\} \\
& \quad \cup \{r+2^n \mid r \in \{b+c \mid b \in B_{\{n\}}, c \in C_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r+2^n \mid r \in \{b+c \mid a \in B_{\{n\}}, c \in C_{\{n\}}\}, r \geq 2^n\} \\
& \quad \cup \{r \mid r \in \{b+d \mid b \in B_{\{n\}}, d \in D_{\{n\}}\}, r < 2^n\} \\
& \quad \cup \{r \mid r \in \{b+d \mid a \in B_{\{n\}}, d \in D_{\{n\}}\}, r \geq 2^n\} \\
= \quad & \{r+2^{n+1} \mid r \in \{a+c \mid a \in A_{\{n\}}, c \in C_{\{n\}}\}\} \\
& \quad \cup \{r+2^n \mid r \in \{a+d \mid a \in A_{\{n\}}, d \in D_{\{n\}}\}\} \\
& \quad \cup \{r+2^n \mid r \in \{b+c \mid b \in B_{\{n\}}, c \in C_{\{n\}}\}\} \\
& \quad \cup \{r \mid r \in \{b+d \mid b \in B_{\{n\}}, d \in D_{\{n\}}\}\} \\
= \quad & \{(a+c+2^{n+1})_{\mathrm{n}+2} \mid a \in A_{\{n\}}, c \in C_{\{n\}}\} \\
& \quad \cup \{(a+d+2^n)_{\mathrm{n}+2} \mid a \in A_{\{n\}}, d \in D_{\{n\}}\} \\
& \quad \cup \{(b+c+2^n)_{\mathrm{n}+2} \mid b \in B_{\{n\}}, c \in C_{\{n\}}\} \\
& \quad \cup \{b+d \mid b \in B_{\{n\}}, d \in D_{\{n\}}\}
\end{aligned}
\tag{4.17}
$$

Lastly, we rewrite the left-hand side of Equation 4.16 to show that it is equal to that of Equation 4.17.

$$
\begin{aligned}
&\{s+u \mid s \in (A_{\{n\}} \boxdot B_{\{n\}}), u \in (C_{\{n\}} \boxdot D_{\{n\}})\} \\
&\quad = \{s+u \mid s \in g(B_{\{n\}}, A_{\{n\}}), u \in g(D_{\{n\}}, C_{\{n\}}) \\
&\quad = \{s+u \mid s \in \{r+2^n \mid r \in A_{\{n\}}\} \cup B_{\{n\}}, u \in \{r+2^n \mid r \in C_{\{n\}}\} \cup D_{\{n\}}\} \\
&\quad = \{s+u \mid s \in \{r+2^n \mid r \in A_{\{n\}}\}, u \in \{r+2^n \mid r \in C_{\{n\}}\}\} \\
&\quad \cup \{s+u \mid s \in \{r+2^n \mid r \in A_{\{n\}}\}, u \in D_{\{n\}}\} \\
&\quad \cup \{s+u \mid s \in B_{\{n\}}, u \in \{r+2^n \mid r \in C_{\{n\}}\}\} \\
&\quad \cup \{s+u \mid s \in B_{\{n\}}, u \in D_{\{n\}}\} \\
&\quad = \{s+u+2^{n+1} \mid s \in A_{\{n\}}, u \in C_{\{n\}}\} \\
&\quad \cup \{s+u+2^n \mid s \in A_{\{n\}}, u \in D_{\{n\}}\} \\
&\quad \cup \{s+u+2^n \mid s \in B_{\{n\}}, u \in C_{\{n\}}\} \\
&\quad \cup \{s+u \mid s \in B_{\{n\}}, u \in D_{\{n\}}\}
\end{aligned}
\tag{4.18}
$$

Hence, the two left- and right-hand sides of the consequent in Equation 4.11 are equal, which concludes our proof. ■

Multiplication

In the algorithms for binary bitwise operations and addition, we used the fact that each output bit at position n is determined by input bits at position n plus a carry bit computed at position $n-1$. The fact that at most one bit of information must be communicated from position $n-1$ to n resulted in our algorithm returning two BDDs, one for a set carry bit and one for an unset carry bit, which in turn means that we only have to do 2^1 recursive calls in the worst case. With multiplication, however, the number of additional information that must be transported from one recursive call to another is not constant, but instead grows with the length of the input bitvectors. Hence, the number of required recursive calls in the worst case grows with the length of the bitvectors, which is not acceptable.

An alternative idea is to express multiplication using shift left and addition, as is done in the Egyptian method [72]. The assumption of this method is that addition and multiplication by 0, 1, and 2 (shift left) are cheaper operations than general multiplication. These assumptions are true for BDD-based integer sets, as multiplication by 0 and 1 are trivial and we support addition as well as shift left (see Section 4.6).

Figure 4.10 shows how two 3-bit integers are multiplied using the Egyptian method. First, the integer b is shifted by the bit position of integer a, and then multiplied by the corresponding bit in a, i.e., if the bit of a at position n is set, then $b << n$ is added to the result, otherwise it is not. This multiplication is an example of more than one carry bit influencing one output bit. Both the carries produced from positions 2 (value 2^2) and 3 influence the value of the output at position 4, since at position 2, there are up to 4 bits that are added, giving a maximum result of 4, which is 100 in binary representation. Therefore, not only the next bit position is influenced, but also the one beyond that.

$$
\begin{array}{cccccccc}
 & a_0 & \times & & 0 & 0 & b_2 & b_1 & b_0 \\
+ & a_1 & \times & & 0 & b_2 & b_1 & b_0 & 0 \\
+ & a_2 & \times & & b_2 & b_1 & b_0 & 0 & 0 \\
+ & & & c_4 & c_{2,3} & c_2 & c_1 & 0 & 0 \\
\hline
\end{array}
$$

Figure 4.10.: Multiplication of a and b

Since we have shift and addition on sets of integers, we could define multiplication using the Egyption method on this higher abstraction. This approach, however, does not work because our addition algorithm computes addition via the Cartesian product and ignores relations that may exist between the operands. The desired result for a multiplication of integer sets $A_{\{n\}}$ and $B_{\{n\}}$ is $\{a_{\{n\}} * b_{\{n\}} \mid a_{\{n\}} \in A_{\{n\}}, b_{\{n\}} \in B_{\{n\}}\}$. Designing a multiplication algorithm that follows the Egyptian multiplication method would, however, not produce this result since it involves a sum over n integer sets. Since computing the sum over n integer sets requires $n-1$ binary additions, and each addition is computed via the Cartesian product, the resulting multiplication algorithm will exhibit an unacceptable ratio between precision and efficiency. It is, however, possible to design a multiplication algorithm following the Egyptian method for the special case where one input set is a singleton, as shown in Algorithm 12. This algorithm has been implemented in the context of a bachelor thesis, which also contains an evaluation that shows that most multiplications fall into the singleton case [73].

The algorithm for set and singleton multiplication traverses the non-singleton BDD, while keeping track of the current depth in $n - d$. When the algorithm reaches a $\boxed{0}$ node, then it returns an empty integer set ($\boxed{0}$), as the traversed path does not represent any included element. Should the algorithm reach a $\boxed{1}$ node, then it returns the interval represented by this node, multiplied by the singleton using interval multiplication. Since interval multiplication with a singleton is not exact without congruence information, the BDD that is constructed from the interval multiplication is overapproximated. This overapproximation motivates an algorithm to precisely convert an interval with congruence information (strided interval) to BDD form. We described this subalgorithm in Section 4.3.4. For each decision node, the multiplication algorithm calls itself recursively for the true and false successors, which results in two BDD-based integer sets. The algorithm then adds the singleton, shifted according to the node's depth to the BDD-based integer set from the true successor recursive call, since the corresponding edge represents a set bit (see Egyptian method). Finally, the algorithm returns the unification of the addition result and the false successor BDD-based integer set.

Algorithm `mulSingl`$(A_{\{n\}}\ ,\ b_{\{n\}}\ ,\ d = n)$ **is**
  Input: BDD representing an n-bit integer set $A_{\{n\}}$, an n-bit singleton $b_{\{n\}}$, depth d
  Result: BDD-based $2n$-bit integer set containing multiplication result
1 **switch** $A_{\{n\}}$ **do**
2 **case** $\boxed{1}$ **do**
3 $[l\,..\,h] \coloneqq [0\,..\,2^d - 1]\ *^{\#\square}\ b_{\{n\}}$
4 **return** `ival2bdd`$(l,\ h,\ 2n)$
  end
5 **case** $\boxed{0}$ **do return** $\boxed{0}$
6 **case** $(s \boxed{\circ} u)$ **do**
7 $l \coloneqq$ `add`$($`mulSingl`$(s,\ b_{\{n\}}\ ,\ d = d-1),$ `fromNumber`$(b_{\{n\}} << n-d,\ \boxed{0},\ \boxed{0},\ \boxed{1},\ d = 2n), 2n)$
8 $r \coloneqq$ `mulSingl`$(u,\ b_{\{n\}}\ ,\ d = d-1)$
9 **return** `ite`$(l,\ \boxed{1},\ r)$
  end
end

Algorithm 12: Multiplication with Singleton

For the cases where neither input set is a singleton, we use an approximative algorithm that converts the input BDD-based integer sets to sets of intervals, performs interval multiplication in a Cartesian fashion, and converts the resulting set of intervals back to a BDD-based integer set.

Division

Algorithm 13 uses the transfer function for division of the interval abstract domain ($/^{\#\square}$) in combination with the approximative conversion from BDD-based integer sets to sets of intervals. The algorithm's precision can be adjusted using the parameter i, which configures the precision of the conversion to interval sets.

Algorithm `div`$(A_{\{n\}}\ ,\ B_{\{n\}}\ ,\ i)$ **is**
  Input: BDDs $A_{\{n\}}$ and $B_{\{n\}}$, representing n-bit integer sets, precision parameter $i < n$
  Result: BDD-based $2n$-bit integer set containing division result
1 **return** $\bigsqcup \{$`ival2bdd`$(A /^{\#\square} B) \mid A \in$ `bdd2ival`$(A_{\{n\}}\ ,\ i = i)$ $, B \in$ `bdd2ival`$(B_{\{n\}}\ ,\ i = i)\}$
end

Algorithm 13: Division via Interval Transfer Function

Negation

Most modern machines use two's complement to support negative integers. One advantage of the two's complement is that arithmetic operations do not have to distinguish between signed and unsigned integers. A simple way to negate an integer in two's complement is to add one to the integer's ones' complement. Since we have precise algorithms for computing the ones' complement (`not`, see Section 4.6) and Cartesian product-based addition (`add`), we use these two algorithms to implement negation in Algorithm 14.

Algorithm `negate`($A_{\{n\}}$) **is**
  Input: Operand BDD representing n-bit integer set $A_{\{n\}}$
  Result: BDD-based integer set containing $\{-a \mid a \in A_{\{n\}}\}$
1 **return** `add(not(`$A_{\{n\}}$`), fromNumber(1,` $\boxed{0}$`,` $\boxed{0}$`,` $\boxed{1}$`,` $d = n$`),` n`)`
end

Algorithm 14: Negation Algorithm

4.6. Bitwise Operations on one BDD

In this section, we present algorithms for bitwise transfer functions on one BDD. Even though rotations and shifts are usually binary operations, we only provide algorithms for shifts by a singleton, hence they operate on one BDD and are presented in this section.

Bitwise Not

The bitwise `not` applies the logic negation to each bit of an input integer individually. Since the bits of the included integers of a BDD-based integer set at a specific position are represented by the BDD edges at a corresponding depth, each true edge in the BDD must be turned into a false edge and vice versa. Algorithm 15 does exactly that by recursively traversing the BDD to the terminals, and reconstructing each node with opposite true and false successors.

Algorithm `not`(A) **is**
  Input: BDD-based integer set A
  Result: BDD-based integer set $\{\neg a \mid a \in A\}$
1 **switch** A **do**
2 **case** $\boxed{0}$ **do return** $\boxed{0}$
3 **case** $\boxed{1}$ **do return** $\boxed{1}$
4 **case** $(a \boxed{\circ} b)$ **do return** $(\texttt{not}(b) \boxed{\circ} \texttt{not}(a))$
  end
end

Algorithm 15: Bitwise Not

There is potential for optimization in Algorithm 15. Consider an integer set containing

all even integers of a fixed bitwidth. Assuming n-bit integers, our BDD representation of this set consists of n decision nodes. Each of these decision nodes has its true and false successors set to the same sub-BDD. Hence, Algorithm 15 executes two recursive calls per decision node, yielding exponential runtime even though n recursive calls are sufficient. Introducing a case that checks for equivalence of the two input BDDs, and uses only one recursive call in case of equivalence, alleviates this inefficiency.

Sized, Drop, and Fill

In this section, we first present helper algorithms that we will use later in this section to define various shift and rotation algorithms.

The `sized` algorithm (Algorithm 16) takes a BDD A, a target height d, and another BDD T, and cuts A to a depth d, and places T at all cutting points. This algorithm is used to limit the height of A.

```
Algorithm sized(A, d, T) is
    Input: BDD A, target height d, path end BDD T
    Result: BDD with maximum height d + depth(T)
1   if d = 0 then return T
2   switch A do
3       case 0̲ do 0̲
4       case 1̲ do 1̲
5       case (S ⊡ U) do
6           return (sized(S, d − 1, T) ⊡ sized(U, d − 1, T))
        end
    end
end
```

Algorithm 16: Sized

Algorithm 17 removes the first n decision nodes from a given BDD A, by computing the disjunction of all sub-BDDs at depth n.

Algorithm `drop`($A_{\{n\}}$, d) **is**
  Input: BDD $A_{\{n\}}$, bits to drop d
  Result: BDD with maximum height $n - d$, with the first d bits dropped
1 **if** $n = 0$ **then return** $\{A\}$
2 **switch** A **do**
3 **case** $\boxed{0}$ **do return** $\boxed{0}$
4 **case** $\boxed{1}$ **do return** $\boxed{1}$
5 **case** $(S \boxed{\circ} U)$ **do**
6 `ite(drop(S, n - 1), 1, drop(U, n - 1))`
  end
  end
end

Algorithm 17: Drop

The `fill` algorithm (Algorithm 18) takes a depth d and a BDD T, and places T at the end of a chain of decision nodes where both outgoing edges point to the next element in the chain. This operation is therefore equivalent to shifting the ordering of the variables in the BDD, because the root node of T becomes a decision node at depth d in the resulting BDD.

Algorithm `fill`(d, T) **is**
  Input: Number of bits to ignore d, BDD to end paths T
  Result: BDD with maximum height $d +$ `depth`(T), ignoring the first d bits
1 **if** $d = 0$ **then**
2 **return** T
  else
3 $temp \coloneqq$ `fill`$(d - 1, T)$
4 **return** $(temp \boxed{\circ} temp)$
  end
end

Algorithm 18: Fill

The `bitExtract` algorithm (Algorithm 19) combines `sized` and `drop` to perform slicing at the bit level. It takes an integer set $A_{\{n\}}$, a high bit position h, and a low bit position l, and returns an $h - l$ bit integer set that contains only the bits between h and l of all integers in $A_{\{n\}}$. The algorithm can be used to model shadow registers, used in x86 assembly.

Algorithm `bitExtract`($A_{\{n\}}$, h, l) **is**
  Input: n-bit integer set $A_{\{n\}}$, start of slice h, end of slice l
  Result: $h - l$ bit BDD integer set containing slices of integer
1 **return** `sized(drop(`$A_{\{n\}}$, $n - h$`)`, $h - l$, $\boxed{1}$`)`
end

Algorithm 19: bitExtract

Shift Left

A shift left moves the bits of a bitvector to the left (MSB) a given number of times. Bits that are shifted outside the range of the vector are discarded, new bits on the right-hand side are filled with zero.

In our BDDs, the MSB is represented by the root decision node's edges. Hence, we can use `drop` to discard the desired number of bits, ending up with a BDD that represents the remaining bits only. In the next step, we compute a BDD that represents the filled in zeros on the bottom of the BDD, using `fromNumber` to create a BDD representation of a bitvector with only zeros, and `fill` to shift this bitvector to the right position. Lastly, we compute the conjunction of the two BDDs using `ite`.

Algorithm `shiftl`$(A_{\{n\}}, s)$ **is**
  Input: BDD-based n-bit integer set $A_{\{n\}}$, steps to shift s
  Result: BDD-based integer set $\{a << s \mid a \in A\}$
1 `dropped` $\coloneqq$ `drop`$(A_{\{n\}}, s)$
2 `zeros` $\coloneqq$ `fill`$(s,$ `fromNumber`$(0, \boxed{0}, \boxed{0}, \boxed{1}, d = n - s))$
3 **return** `ite`$($`dropped, zeros,` $\boxed{0})$
end

Algorithm 20: Shift Left

Shift Right

Algorithm 21 implements shift right by adding a bitvector of zeros above the MSB of $A_{\{n\}}$ using `fromNumber`, and cutting the resulting BDD to the required height using `sized`.

Algorithm `shiftr`$(A_{\{n\}}, s)$ **is**
  Input: BDD-based n-bit integer set $A_{\{n\}}$, steps to shift s
  Result: BDD-based integer set $\{a >>> s \mid a \in A\}$
1 **return** `sized`$($`fromNumber`$(0, \boxed{0}, \boxed{0}, A_{\{n\}}, d = s), n, \boxed{1})$
end

Algorithm 21: Shift Right

Shift Right Arithmetic

In difference to shift right on bit vectors, arithmetic shift right does not always fill with 0 at the MSB position, but instead uses the value of the sign before shifting to fill at MSB. As a result, after executing an arithmetic shift right of l positions, the $l + 1$ most significant bits will have the same value. An arithmetic shift on integer sets must therefore treat all those elements that have their MSB set differently from those that do not. Accordingly, Algorithm 22 inspects the MSB in Line 4. After this inspection, S represents all sub-bitvectors with the MSB set, and U all sub-bitvectors with an unset

MSB. Hence, S is shifted with 1 as filler using `fromNumber` with $2^n - 1$, a number with all bits set in unsigned representation, and u is shifted using 0 as filler. The resulting BDDs are restricted to the given integer set's size.

Algorithm `shiftrarith`($A_{\{n\}}$, s) **is**
  Input: BDD-based n-bit integer set $A_{\{n\}}$, steps to shift s
  Result: BDD-based integer set $\{a >> s \mid a \in A\}$
1 **switch** $A_{\{n\}}$ **do**
2 **case** $\boxed{0}$ **do return** $\boxed{0}$
3 **case** $\boxed{1}$ **do return** `sized`$(((\boxed{1} \boxdot \boxed{0}) \boxdot (\boxed{0} \boxdot \boxed{1}))$, n, $\boxed{1})$
4 **case** $(S \boxdot U)$ **do**
5 `negative` $\coloneqq$ `sized(fromNumber(`$2^n - 1$, $\boxed{0}$, $\boxed{0}$, S, $d = s$`)`, n, $\boxed{1})$
6 `positive` $\coloneqq$ `sized(fromNumber(`0, $\boxed{0}$, $\boxed{0}$, U, $d = s$`)`, n, $\boxed{1})$
7 **return** `ite(negative,` $\boxed{1}$`, positive)`
  end
end

Algorithm 22: Shift Right by One Arithmetic

Rotate Left

Rotate left (`rol`) shifts the bits of a bitvector to the left, and fills the LSB position with the bit that was discarded at the MSB during the shift. On BDDs, this operation is essentially equivalent to BDD reordering, which is known to be prohibitively expensive.

We therefore choose to implement the rotate operations in an overapproximating manner. Specifically, we ignore the relations between the bits that are shifted out at the LSB position and those that are added at the MSB position. As an example, consider the set of bit vectors $\{101, 010\}$ that, precisely rolled to the left by one positions would yield $\{011, 100\}$, since in 101 the MSB is shifted outside and therefore added at the LSB and similarly in 010. Our overapproximating variant cannot relate the bits that are shifted out to the part that remains and will therefore return $\{011, 101, 010, 100\}$, i.e., it will combine each remaining part with each shifted out part. Implementing rotation by n positions using n rotations by 1 position would lead to unacceptable loss in precision, since each rotated bit would lose its relation to all other bits. We therefore implement rotations by n positions directly. Consequently, relations are lost only between the sub-bitvector of rotated bits and the remaining bits.

Algorithm `rol`$(A_{\{n\}}$, $l)$ **is**
  Input: BDD-based n-bit integer set $A_{\{n\}}$, number of bits to rotate l
  Result: BDD-based integer set $\{a$ `rol` $l \mid a \in A\}$
1 $t \coloneqq$ `sized`$(A_{\{n\}}$, l, $\boxed{1})$
2 $d \coloneqq$ `drop`$(A_{\{n\}}$, $l)$
3 $s \coloneqq$ `fill`$(n - l$, $t)$
4 **return** `ite`$(d$, s, $\boxed{0})$
end

Algorithm 23: Rotate Left

Rotate Right

The algorithm for rotate right (`ror`), is almost the same as for left rotation and differs only in the boundary where the BDD is split using `sized` and `drop` and reassembled using `ite`.

Algorithm `ror`$(A_{\{n\}}$, $r)$ **is**
  Input: BDD-based n-bit integer set $A_{\{n\}}$, number of bits to rotate r
  Result: BDD-based integer set $\{a$ `ror` $l \mid a \in A\}$
1 $t \coloneqq$ `sized`$(A_{\{n\}}$, $n - r)$
2 $d \coloneqq$ `drop`$(A_{\{n\}}$, $n - r)$
3 $s \coloneqq$ `fill`$(r$, $t)$
4 **return** `ite`$(d$, s, $\boxed{0})$
end

Algorithm 24: Rotate Right

4.7. BDDStab Library Implementation

The BDDStab library (Chapter 4) implements BDD-based integer sets (Sections 4.1, 4.2), as well as transfer functions on these sets (Section 4.4 to 4.6). With it, we target integer value analyses that are implemented on the Java platform, i.e., that are executed on the Java Virtual Machine (JVM). To use pattern matching on BDDs, which we believe not only simplifies the implementation of our library but also makes the code readable, we chose Scala as our implementation language. With Scala, the library consists of about 4000 lines of code. Since Scala provides language constructs that Java does not, we provide Java compatible APIs whenever needed.

Different integer analyses require different abstractions. It is common for most languages to only support a limited set of integer types with constant bitwidth, such as `int` or `long`. For the analysis of these languages, it is sufficient to provide a set abstraction that can only handle exactly these types, i.e., a BDD-based set supporting constant bitwidth integers. However, there exist languages that support integers of varying bitwidth, which demands a set data structure that can support these. In Jakstab, each integer is a combination of a bitwidth and a bitvector, where the bitwidth determines where overflow happens in additions.

We therefore provide different entry points to our BDD-based integer sets, all with associated transfer functions. Figure 4.11 shows an overview of these entry points. In this figure, the arrow denotes composition, i.e., CBDDs are a wrapper around BDDs, IntSets wrap CBDDs, and IntLikeSets wrap IntSets. Most algorithms are defined on the CBDD class, as it represents reduced BDDs with labelless nodes and complementable edges. IntSets and IntLikeSets are integer sets that implement Scala's set trait. The difference between them is that IntSets only supports integers with constant bitwidth, while IntLikeSets supports integers with dynamic bitwidths as required by Jakstab.

In Sections 4.7.1 to 4.7.4, we provide more details for each of the library's entry points and give an overview of their APIs. We finish with a description of our test suite in Section 4.8.

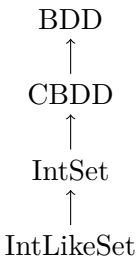


Figure 4.11.: Main Hierarchy in BDDStab Library

4.7.1. The BDD Class

The BDD class implements BDDs with complement edges, but without the complement information that applies to the root of a BDD. Therefore, this class only represents the part of a BDD with complemented edges that is shared using a unique table [56].

One of the challenges when implementing a BDD package is to enforce the BDD's reduction rules. With labelless BDDs, we have to remove redundant sub-BDDs, and replace a decision node by one of its sub-BDDs if both sub-BDDs are the same terminal. We implement both reduction rules in a factory for decision nodes, require that all non-terminal BDDs are built using it, and provide static objects for the terminal. Implementing Rule 1 from Section 4.1.1 is done by only creating a new decision node if not both sub-BDDs are the same terminal. If they are, then the factory returns one of them.

The factory's implementation of Rule 2 from Section 3.9 is done using hash consing, which ensures that each BDD exists at most once in memory. Hence, if two BDDs are equivalent, they are the same object. Hash consing exploits that both argument sub-BDDs have been constructed using the factory, and must therefore be reduced already. The implementation uses a unique table that stores the forest of BDDs that have previously been constructed. If the BDD that the factory must build exists in the forest, then this BDD is returned. Otherwise, a new BDD is constructed and added to the forest. To enable fast lookup in the forest, we use a weak hashmap, which requires fast hash computation and equality checking between BDDs. As is common in hash consing, we make the hash of a decision node dependent only on the hashes of its sub-BDDs, and store the hash of each BDD in its root, thus avoiding recursion. When constructing a decision node with sub-BDDs A and B, the factory creates a hash for $(A \boxed{\circ} B)$, and accesses the corresponding bucket in the hashmap. Within this bucket, the factory searches for the BDD $(A \boxed{\circ} B)$, using the $O(1)$ reference equality on A and B that is established by hash consing, and returns it if it exists. Otherwise, the factory creates a new decision node, adds it to the hashmap and returns it.

The BDD class itself is not intended to be used outside of its wrapper class CBDD, which we present next.

4.7.2. The CBDD Class

The CBDD class implements labelless BDDs with complemented edges, and $O(1)$ satisfiability count as described in Section 4.1. To simplify the use of CBDDs, we provide an API that transparently handles the complement bits, as well as the machinery for the satisfiability count and acts as a simple binary tree with Booleans at the leaves.

The API addresses two concerns: construction and deconstruction. Construction of a CBDD is done using either a terminal node, available via the objects `True` and `False` in Scala, and `True$.MODULE$` and `False$.MODULE$` in Java, or via creating a decision node given two CBDDs. The construction of a decision node with true successor BDD `T` and false successor BDD `F` is done using `Node(T, F)` in Scala, and `Node$.MODULE$.apply(T, F)` in Java. Deconstruction, i.e., extracting the true and false successor BDDs from a

BDD, is done using pattern matching agains `Node(a, b)` in Scala, where `a` will be assigned the true successor BDD, and `b` the false successor BDD. In Java, one uses the `Node$.MODULE$.unapply` method that returns an optional tuple of true and false successor BDDs. This optional value is empty, if the given BDD cannot be deconstructed, i.e., when it is a terminal.

Additionally, CBDDs contain static methods, e.g., for the if-then-else function on BDDs and BDD evaluation, the construction algorithms described in Section 4.3, and the transfer algorithms from Sections 4.4 to 4.6. The presence of these algorithms on the CBDD level facilitates the implementation of new BDD-based integer set data structures for value analysis.

4.7.3. The IntSet Class

IntSets implement BDD-based integer sets for any finite integral type with a constant finite bit length, such as `Integer` and `Long`. Only integer types that support certain functionality can be used with Intsets. These types must be integral, i.e., support common operations on integers such as addition, negation, multiplication and comparison, and must be representable using a constant size bitvector. These type constraints are implemented using Scala type classes, implementations of which must be passed during Intset construction. The BDDStab library provides implementations for Java's `Integer` and `Long` types.

The API supports IntSet creation via the overloaded method `IntSet` in Scala, and `IntSet$.MODULE$.apply` in Java. This method can construct IntSets from Scala integer sets, Java integer sets as well as intervals.

4.7.4. The IntLikeSet Class

IntLikeSets implement BDD-based integer sets for integers up to a finite bitwidth, i.e., finite integers. Additionally to the requirements for IntSets, IntLikeSets require the included type to support an operation that returns their bitwidth, as well as one that converts them into a constant bitwidth type such as `Long`. This abstraction allows us to have the API deal with Jakstab numbers (RTLNumbers) directly instead of converting in the Jakstab Adapter.

Creation of IntLikeSets is done similarly to that of IntSets, i.e., via an overloaded `IntLikeSet` method in Scala, named `IntLikeSet$.MODULE$.apply` in Java. Since the Scala methods require passing of type class implementations, we provide special construction methods for Java's `Long` type so that no passing of type class implementations is necessary in this case.

4.8. Testing

The BDDStab library contains about 60 ScalaCheck tests, testing not only the set types for basic functionality such as adding elements to the set and checking for membership, but also the implementation of all algorithms from Sections 4.3 to 4.6 with particular

focus on `algo`, `add`, and `mulSingl`. These tests are formulated as functions that, given appropriate input data, return true exactly when the implemented test case succeeds. ScalaCheck generates pseudo random input, and calls the tests with it. Most of our tests use an oracle, i.e., a simple but inefficient implementation of an algorithm using explicit sets that predicts the correct result. To facilitate the implementation of such oracles for IntLikeSets, the BDDStab library includes a class for integers with a dynamic behavior, i.e., arbitrary bitwidth and corresponding wrap-around.

4.9. Evaluation

We evaluate the performance of the BDDStab library on integer sets that have exact representations in the common abstractions of integer sets. If these experiments reveal that BDD-based integer sets are suitable for the exact representation of these shapes, then it is likely that they are also suitable for the representation of integer sets that are close to these representations, as the BDD shape will not change drastically by adding or deleting a few elements. In particular, we evaluate the following two set shapes, known from other value analyses:

1. Intervals
2. Congruence Classes

For intervals, we evaluate the efficiency of the BDD representation, and for congrunece classes, we additionally evaluate the construction speed, and the performance of the precise algorithms for abstract evaluation, namely addition and binary bitwise operations.

4.9.1. Measuring Procedure

To determine the efficiency of our BDD-based integer representation and measure the performance of algorithms, we record the following measurements:

1. The number of decision nodes in unique sub-BDDs
2. The number of decision nodes in all sub-BDDs
3. Time spent during execution of an algorithm

Measurement 2 is the number of decision nodes a labelless BDD would include if equivalent sub-BDDs would not be shared. We call the BDD that would result from not sharing equivalent sub-BDDs binary decision tree (BDT). This number is computed during node construction and is therefore easily available. Only Measurement 1 is not available and must be computed. Since we use maximal sharing via hashconsing (Section 4.7.1), we compute this number by constructing the BDD we want to measure, then request the garbage collector to run, wait and query the number of elements in the unique table that stores all unique sub-BDDs. Before construction and after measurement we clear the unique table. Requesting and waiting for the garbage collector is necessary to purge the weak hashmap that implements our unique table of decision nodes that were only constructed temporarily and are not part of the final result. Because only decision nodes are stored in the unique table, this measurement does not include the terminal node. Measurement 1 provides insight into the nodes our BDD-based integer set implementation uses in total to represent integer sets, while the relation between Measurement 2 and 1 shows the effectiveness of sharing equivalent sub-BDDs within one BDD. Note that sharing increases its effectiveness with a growing forest. To measure

the speed of our algorithms, we used an early 2015 MacBook Pro with a 3.1 Ghz i7 processor and 16 GB of DDR3 ram.

In the evaluation of the algorithms for abstract evaluation, we use Measurement 3, i.e., the time spent computing results for a representative subset of a given set shape. As an example, to evaluate the addition algorithm on intervals, we choose a representative subset of intervals, apply the addition algorithm on all elements in this subset and measure its execution time. Note that we do not instruct the garbage collector to run when evaluating algorithm performance, since our algorithms make use of caching to improve performance, and the garbage collector may clear the caches, hence negatively affecting algorithm performance.

4.9.2. Representation Efficiency for Intervals

To evaluate the efficiency of our BDD-based integer set data structure when representing intervals, we construct all 10-bit intervals, represent them with BDD-based integer sets, and record Measurement 1. Using 10-bit intervals results in $2^{10} * 2^{10}/2 = 524288$ intervals, since there exist 2^{10} possible lower and upper bounds, giving 2^{20} possible combinations of lower and upper bounds, half of which have a lower bound that is smaller or equal to the upper bound. Measuring all 11-bit intervals would already have required about 2 million measurements, which would have required an excessive amount of time. Since we spend most of the time waiting for the garbage collector to run, we cannot increase the benchmark speed by using more powerful hardware. Hence, 10-bit intervals is a good compromise between a speedy execution of the benchmark and a representative bitwidth, i.e., one that is wide enough to reveal the behavior of the BDDs on larger bitwidths.

Figure 4.12 shows a histogram of the reduced BDD sizes for all 10-bit intervals. The x-axis shows Measurement 1, i.e., the number of nodes in the reduced BDD, and the y-axis denotes how often this number was measured in logarithmic scale. We observe that the BDD size varies between 23 and 44 nodes, with an average number of 36 nodes.

It may be surprising that we observe BDD sizes greater than twice the bitwidths of the represented intervals, since an n-bit interval is represented by two n-bit numbers. This oddity is explained by the fact that we use the bitwidth of an underlying type internally. In the interval measurements, we used Java `Integer`, i.e., 32-bit as underlying type, meaning that only the last 10 bits of the BDDs represent the intervals, while the upper 22 bits are set to 0. Hence, the theoretical maximum BDD size for 10-bit intervals is 42 because there are $32 - 10 = 22$ bits outside of the 10 bit used to represent the intervals, and $2 * 10$ bits are used for the two interval boundaries. However, the data contains a sample where the maximum measured result is 44, i.e., 2 over the theoretical maximum. Closer investigation reveals that this is the only test case where the number of nodes in the BDT form is smaller than that of the reduced BDD form ($44 > 23$). Recreation of this interval and waiting for garbage collection showed the real number of reduced BDD nodes to be 23. Hence, the result of this test case is an artifact.

With a theoretical maximum BDD size of $2n + c - n$, when representing n-bit intervals in an underlying c bit integer type, where c is usually 64, the size of BDDs representing

intervals remains small. Better yet, further investigation of our benchmark data reveals that sharing equivalent sub-BDDs improves the average BDD size by about 14%, even when representing intervals, which only demand storing two numbers and therefore lead to a simple BDD shape.

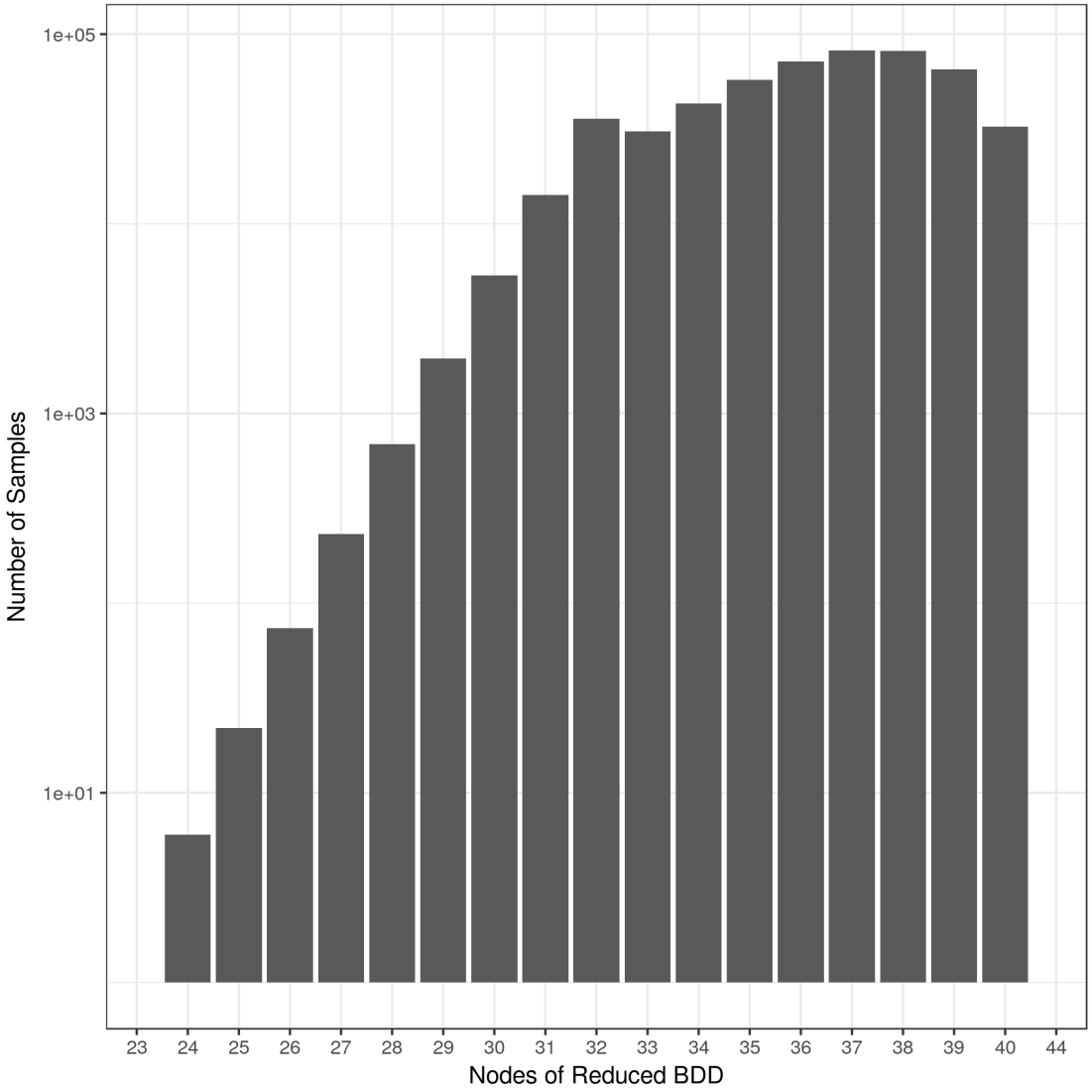


Figure 4.12.: Node Count of Reduced BDD Representing 10-bit Intervals

4.9.3. Representation Efficiency for Congruence Classes

Compared to n-bit intervals, where the BDD representation must at most store two n-bit paths, congruence classes produce BDDs with more internal nodes, as they are non-convex and hence must represent holes between any two included elements. There-

fore, representing congruence classes using BDD-based integer sets is more challenging, and hence we provide a more in-depth evaluation that covers the effectiveness of reduction Rule 1 from Section 4.1.1 (Measurement 2) as well as the performance of various algorithms (Measurement 3). We use Algorithm 6 to construct integer sets with dynamic bitwidth set to 16, and the underlying constant bitwidth 32, for all possible strides. In total, we therefore measure $2^{16} - 1 = 65535$ BDD-based integer sets, since 0 is not a valid stride. We chose 16 bit, as it is a common integer size and still small enough to allow measuring all possible congruence classes. In difference to intervals, which are defined by two parameters, i.e., upper and lower bound, and are therefore hard to plot in two dimensions, congruence classes are defined by one parameter only, i.e., the stride. Hence, we will not use histograms, as we did for intervals, but instead use a regular two-dimensional plot.

Figure 4.13 shows our measurements with the stride on the x-axis in logarithmic scale, and the number of nodes in the reduced BDD on the y-axis. The node count for the reduced BDDs ranges from 16, at stride 1, to 780 for stride 257, and is 77 in average.

It may be surprising that the minimum size of the reduced BDD is 16 and not 0, as stride 1 should generate $\boxed{1}$ as BDD, which is not measured because it is a terminal. Similarly to the interval benchmark, the first $32 - 16 = 16$ bits are set to 0 in our implementation. Hence, 16 is the least number of nodes possible for any non-empty 16-bit integer sets, represented using an underlying 32 bits in the BDD. Apart from the left most point, corresponding to stride 1, the strides producing the smallest BDDs are 2^n strides, which is not surprising because in these strides, the last n bits of all included elements and, therefore, their BDDs are constant. Figure 4.13 shows several sub-plots that behave similarly, i.e., they rise to their peak, and then fall again. We call each of these sub-plots a stride class, distinguish them using color, and number them as given in the legend. Closer investigation reveals that each stride can be assigned to exactly one stride class as it fulfills $s\%2^c == 2^{c-1}$, where s is the stride and c the stride class number, only for exactly one class number. As an example, consider the stride class with the highest peak (outer, enclosing plot). All strides s in this stride class satisfy $s\%2 == 1$, i.e., the strides are all odd. Similarly, the next lower stride class are built from strides s with $s\%4 == 2$. From the classes, we can conclude that the lower the stride class, the less efficient is the BDD representation.The shape of the stride classes, i.e., rising quickly in the beginning then falling off, confirms our expectation about the performance of Algorithm 6: Small strides lead to more sharing (memoizing), while large strides lead to small sets. Both, sharing and small set sizes, can produce minimal BDDs.

The average 77 nodes required to represent 16-bit congruence classes within reduced BDDs with 32 variables is encouraging, especially when considering that the congruence class with stride 2 would have 2^{16} nodes without sharing.

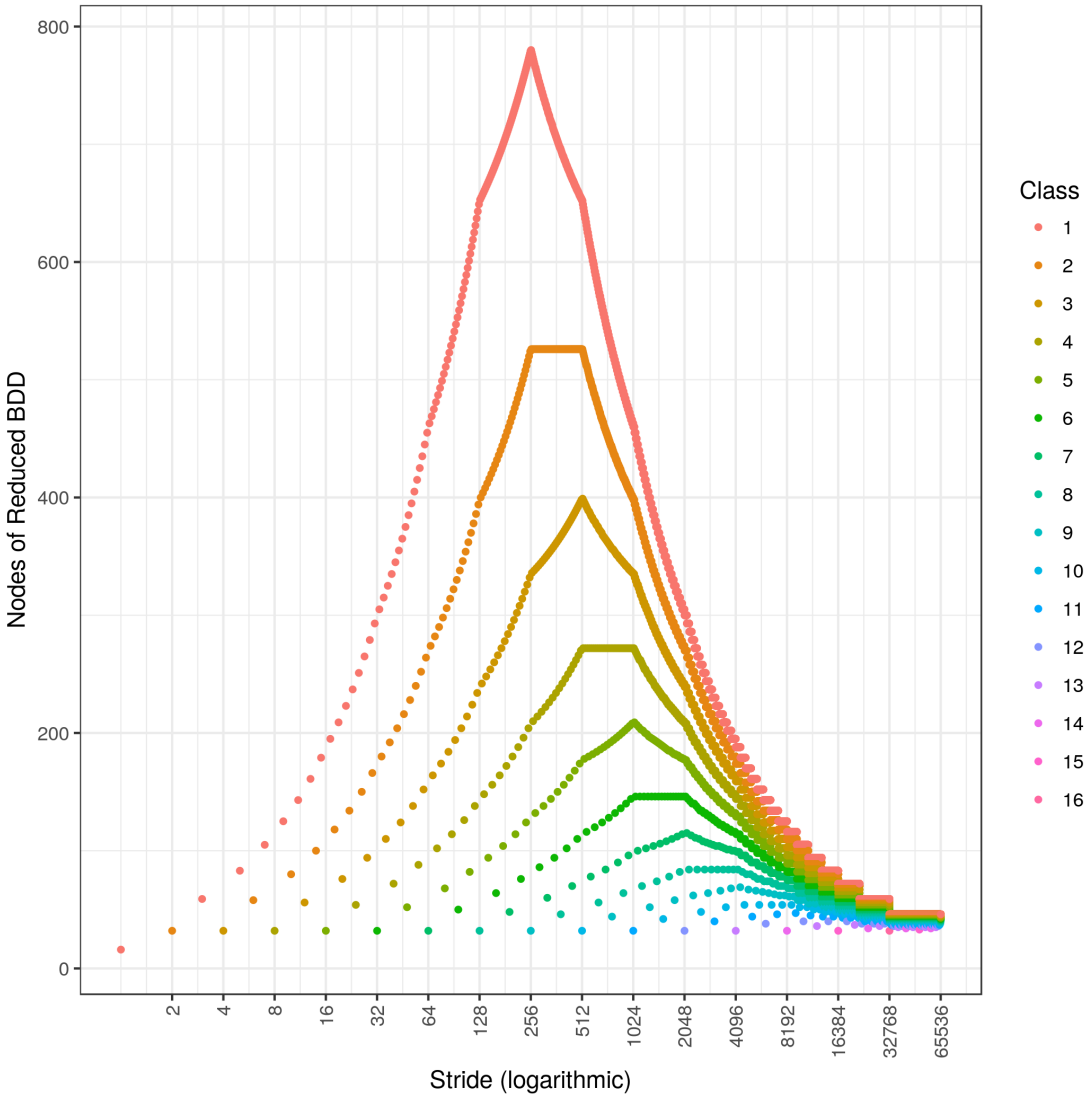


Figure 4.13.: Node Count for all 16-bit Strides in 16-bit Integer Sets

To investigate further the efficiency of BDD reduction when representing congruence classes, we next show the relation between the node count of BDTs (Measurement 2) to that of the reduced BDDs (Measurement 1), i.e., we show the factor by which the reduced BDD contains fewer nodes than the non-reduced BDT. Figure 4.14 contains the plot of the relation, again for all 16-bit congruence classes, with the stride on the logarithmic x-axis, and the relation on the y-axis. We see that the BDD and BDT sizes are essentially equal for stride 1. The reason for equality in this case is that stride 1 leads to a full set of all 16 bit integers, which is represented by just $\boxed{1}$ for the reduced BDD and the BDT that implements only reduction Rule 2. The most efficient representation of each stride class is at a power of 2, since the bitvectors contained in the BDDs have

constant trailing zeros. As an example, the congruence class of 4-bit integers with stride 4 contains 0, 4, 8, and 12, all of which have the two least significant bits set to 0. Above these constant bits, the stride class will contain all bit patterns, which results in an efficient BDD representation. Unsurprisingly, the stride class with the highest node count (Class 1 in Figure 4.13) has the least efficient BDD representation.

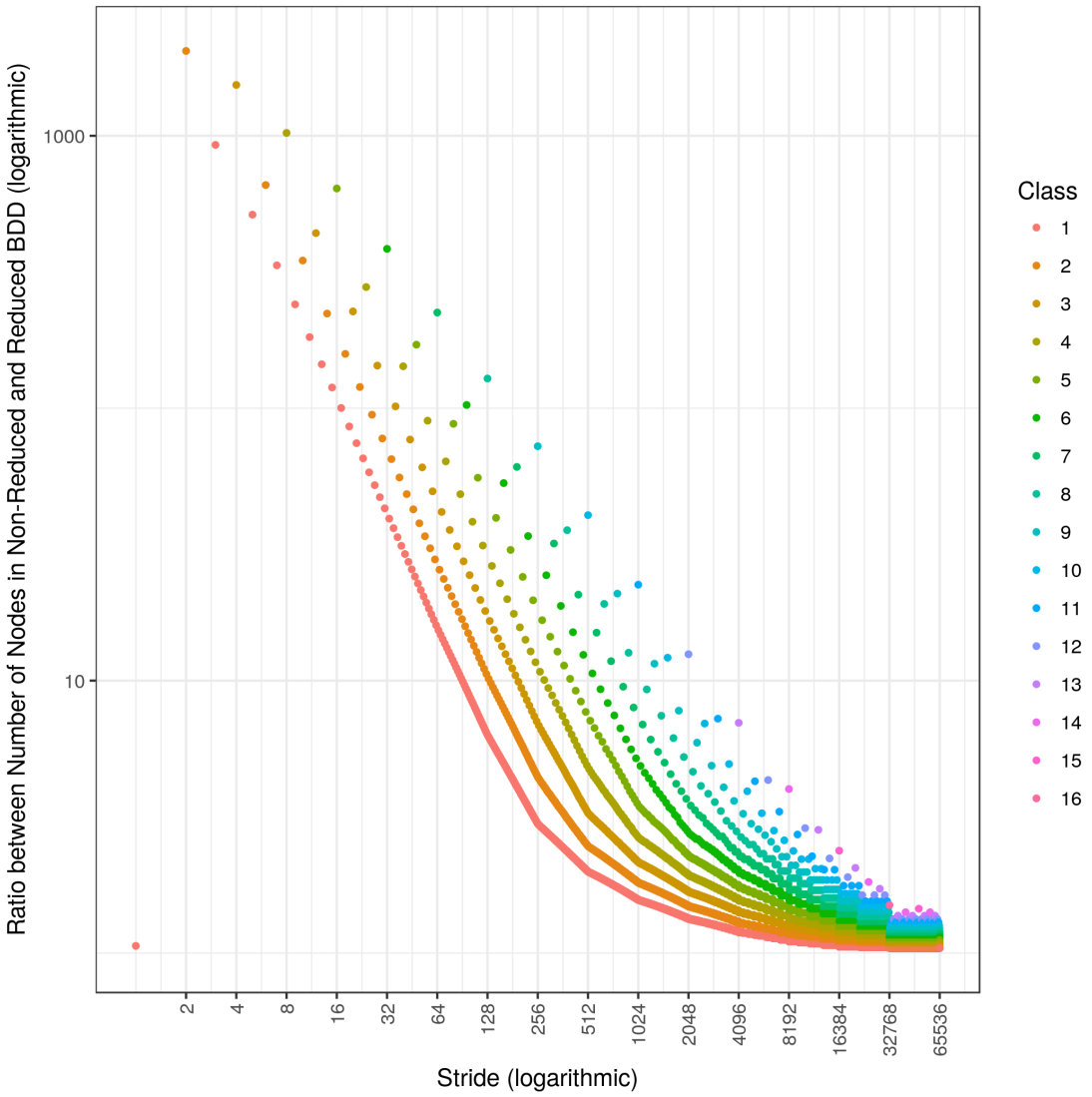


Figure 4.14.: Ratio between Number of Nodes in Non-Reduced and Reduced BDDs for all 16-bit Strides in 16-bit Integer Sets

Construction Performance for Congruence Classes

Figure 4.15 shows the construction times, i.e., Measurement 3, of congruence classes when using Algorithm 6 to construct all congruence classes of 1- to 24-bit integers. For each bitwidth n, we construct all congruence classes with the strides from 1 to $2^n - 1$, meaning we create up to $2^{24} - 1$ sets for one measurement. The x-axis shows the bitwidth of the created congruence classes, the y-axis shows the time in seconds it took to create all congruence classes of the given bitwidth, and the color indicates the maximum construction time of one congruence class of the given bitwidth. Note that the scale of the construction time (y-axis) is logarithmic.

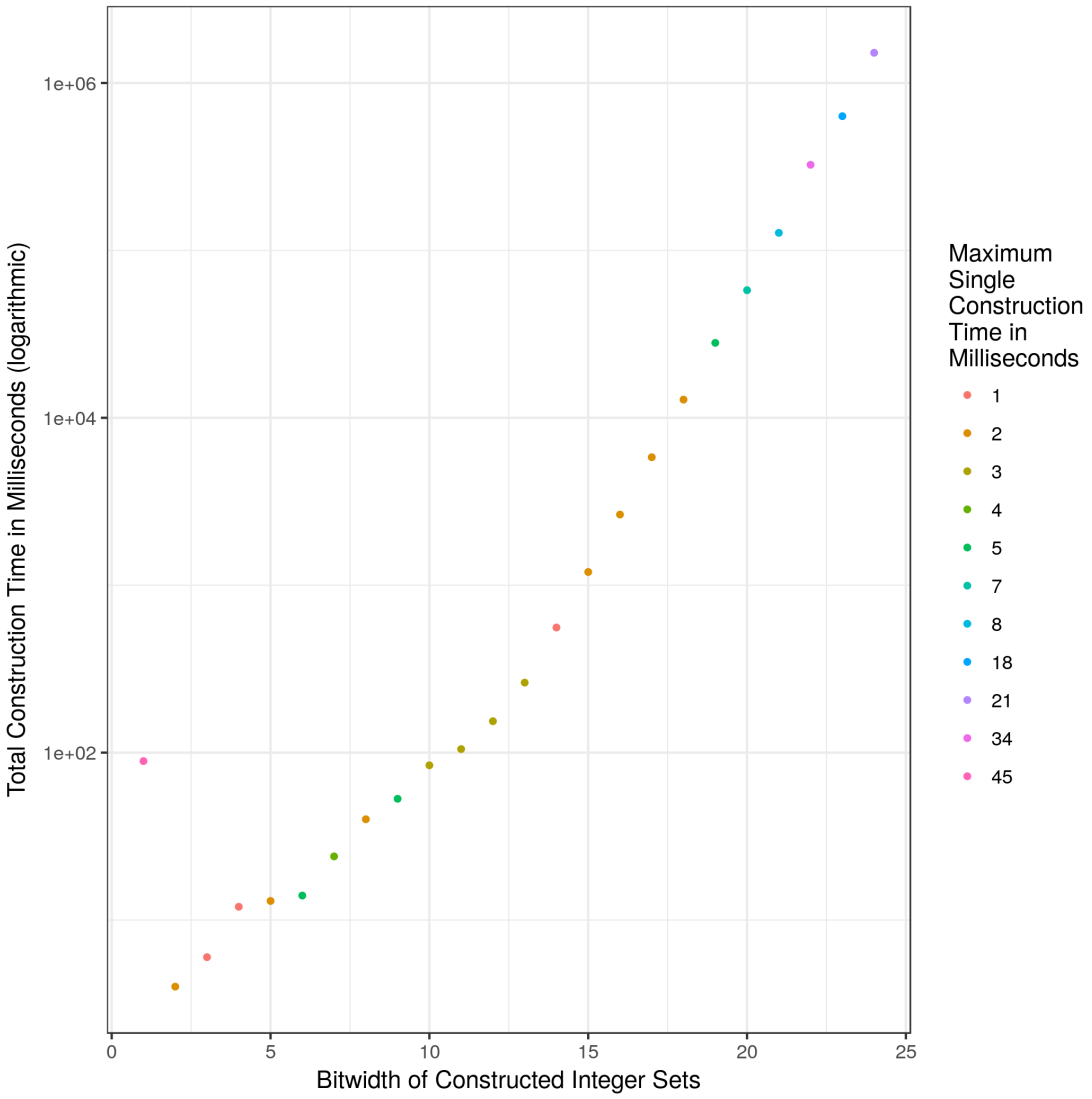


Figure 4.15.: BDD Construction Speed for all Strides in 1- to 24-bit Integer Sets

We observe that the construction of all 1-bit congruence classes, which is only the set $\{0, 1\}$, takes unusually long. We consider this measurement an outlier, most likely caused by the Java runtime system's optimization procedure. It is noticable that, particularly after the 12-bit congruence class constructions, the construction time increases linearly in the logarithmic scale. This progression is not surprising, as we always construct all possible congruence classes and the number of possible congruence classes with offset 0 in n-bit integers is 2^n. Hence, we call the construction algorithm 2^n times for each data point, which suggests that the construction times grow linearly with the bitwidth of the constructed congruence class. The maximum construction times of single congruence classes per bitwidth ranges from 1 millisecond to 34 milliseconds when ignoring the outlier for bitwidth 1. We would expect the maximum single construction time, when ignoring the outlier at 1, to occur at the widest bitwidth, but instead, we measured it at bitwidth 22, which suggests that there is jitter on the measurements. This jitter can, e.g., be caused by the garbage collector.

Algorithm Performance for Congruence Classes

The previous benchmarks have shown that our BDDs efficiently represent congruence classes by sharing common sub-BDDs, which leads to smaller reduced BDDs. However, smaller BDD sizes do not necessarily improve the performance of the algorithms that operate on them. Ignoring memoization, algorithms are defined on the BDT form, and we have shown that there may be huge differences between this non-reduced form and the reduced form. We therefore evaluate the performance of the exact algorithms for binary transfer functions, namely the addition algorithm (Algorithm 11), and the algorithm for binary bitwise operations (Algorithm 8) on BDD representations of congruence classes. Since we expect the same performance for any of the common bitwise operators ($\vee$ and $\wedge$ on the bit level), we only present data for bitwise `and`.

Similarly to the measurements on intervals, we chose 10-bit congruence classes, because we want to evaluate the performance of the algorithms on all unique combinations of all n-bit congruence classes, and that leads to 2^{2n} measurements. For 10-bit, we therefore have to record about 1 million samples, which is acceptable.

Figure 4.16 shows the time (Measurement 3) needed by the addition algorithm when adding two 10-bit congruence class sets, while Figure 4.17 shows the same for the logical `and`. As we can see, their performance is almost equivalent. The maximum time spent for one operation is about 1 second for both. However, the mean of all additions is at about 4 milliseconds, while the mean of all bitwise operations is about 3 milliseconds. All but two additions took less than 550 milliseconds, and all but two bitwise operations took less than 750 milliseconds. Since our measurements contain Java runtime actions, it is possible that the two outliers are caused by garbage collector runs. Further inspection of the data did not reveal that the performance of the algorithms is significantly better or worse for specific congruence classes.

Even though speeds close to 1 second may be excessive for the application of one transfer function, the vast majority of cases show a performance of under 4 milliseconds, which is acceptable, considering that both algorithm produce an exact result.

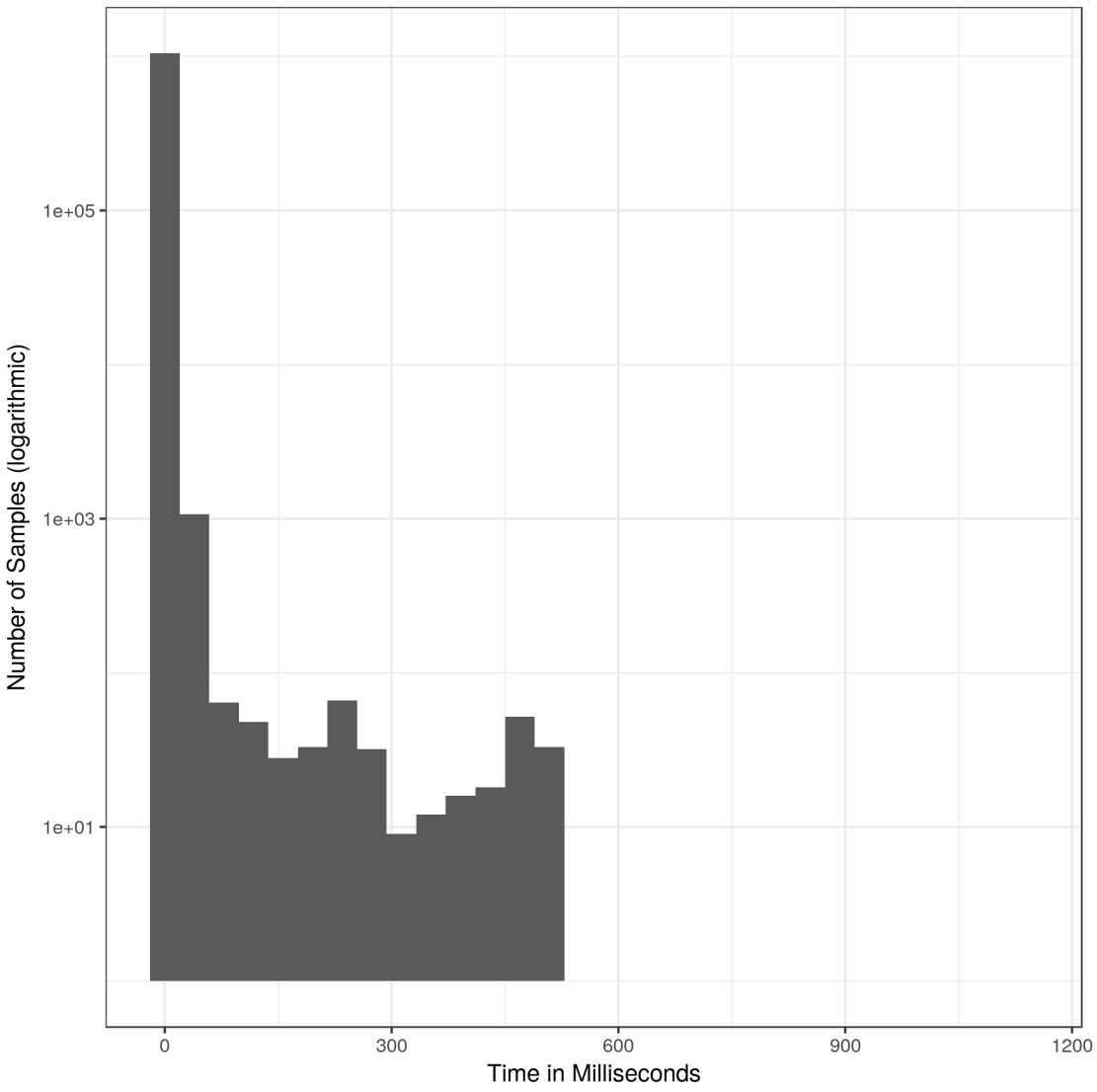


Figure 4.16.: Addition of all Combinations of all 10-bit Congruence Classes

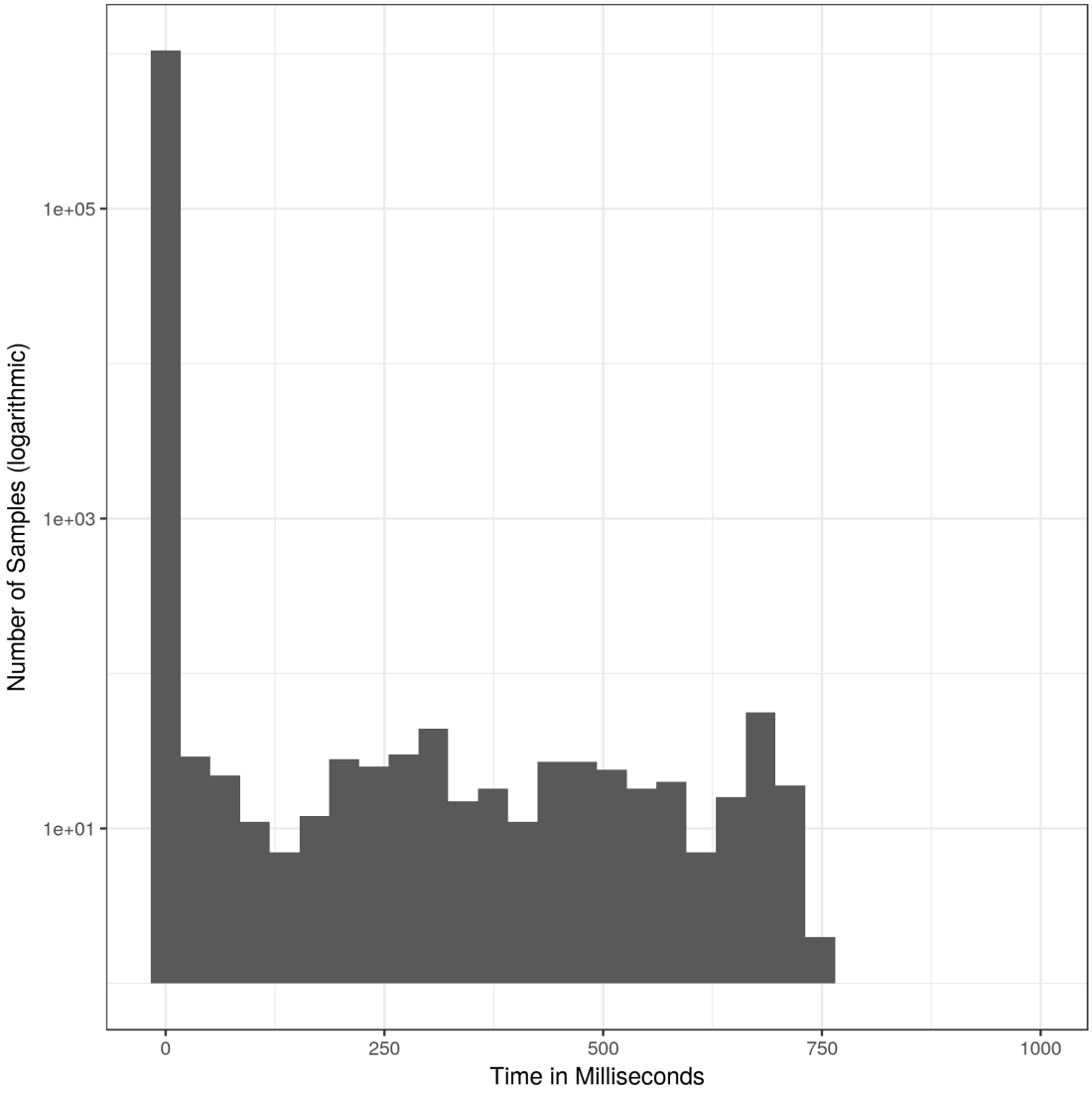


Figure 4.17.: Bitwise And of all Combinations of all 10-bit Congruence Classes

4.9.4. Threats to Validity

We identify two potential threats to the validity of our data:

1. Dirty computed table
2. Tested bitwidths not representative

Threat 1 threatens our measurements because we use `System.gc()` to clean our computed table of non-referenced BDDs and this only suggests to the Java Virtual Machine (VM) that garbage collection may be beneficial, it does not necessarily cause the VM to perform garbage collection. If the production of the BDDs that must be measured causes the creation of temporary BDDs which are not part of the BDD that should be measured, and these temporary BDDs are not dropped by the garbage collector, then the measurement of the number of decision nodes of the BDD will include spurious nodes.

A dirty computed table, however, can only lead to pessimistic values, i.e., to an overestimation of the reduced BDD size. Therefore, the values we report for the reduced BDD sizes must be interpreted as an upper bound, not as the exact size.

Threat 2 may invalidate the conclusions we draw from the measurements as we only tested the BDD sizes for sets of fixed bitwidths. This threat cannot entirely be dismissed. However, we find it unlikely that effects that occur in a certain bitwidth do not occur on wider or narrower sets and vice versa. To double check, we ran a selection of our benchmarks with narrower and wider bitwidth and found the same phenomena, suggesting that our selection of bitwdiths are representative.

5. Jakstab

In Chapter 2, we motivated that reconstructing control flow is the focal task in the analysis of executables, and that this task involves resolving jump targets, which in turn demands the computation of possible sets of integers for the target addresses. Jakstab, a static analysis framework for x86 32-bit executables created by Johannes Kinder [3], provides all the necessary machinery for the flow-based analysis of executables, such as parsing the executable (e.g., ELF or EXE), as well as an implementation of CPA. CPA enables the formulation of an analysis on a dynamically constructed control flow graph, facilitates the implementation of disjunctive refinement via the merge operator, and improves precision via the combination of analyses.

In Jakstab, we distinguish between four different abstract entities: Abstract value, abstract environment, abstract location, and abstract state. An abstract value is a representation of a concrete value, e.g., a representation of Booleans or Integers, while an abstract environment represents an assignment of all storage locations to an abstract value. An abstract location represents a control position in the analyzed program, and an abstract state is the combination of an abstract location and abstract environment. In general, entities in the concrete are singletons, while their abstract counterparts describe a collection of concrete values. As an example, X86 registers may each contain only one bitvector, while in the abstract, each register is associated with a set of bitvectors.

Jakstab draws its strengths from its plug-ins, which implement the core part of the analysis, and communicates with its plug-ins via its intermediate language SSL. Hence, plug-ins must not only support the operators required by CPA, such as merge, stop, and concretization, but also abstract evaluation and transfer functions for all SSL statements and expressions. Soundness requires that these functions are overapproximating (see Chapter 3.5). Jakstab's analysis process is visualized in Figure 5.1. Jakstab keeps a set of active abstract states in the workset and extracts one such state in each analysis iteration. It deconstructs this state into a location in the executable and an abstract environment, lifts the instruction found at the location to SSL, and calls the analysis plug-in with the resulting SSL and the abstract environment. The active abstract states, computed by the plug-in, are added back to the workset and the next analysis iteration is started.

In the remainder of this section, we focus on integer value analysis, i.e., abstract values are sets of n-bit integers.

5.1. Value Analysis with Jakstab

Implementing an integer value analysis plug-in for Jakstab requires the implementation of the abstract environment, abstract values, and transfer and evaluation functions for SSL statements and expressions respectively. In Figure 5.1, a new value analysis plug-in would therefore provide the "Environment" as well as the "Analyze" parts in Jakstab's analysis flow. In Jakstab, abstract locations are hidden from value analysis plug-ins, which therefore operate on abstract environments instead of abstract states. Jakstab

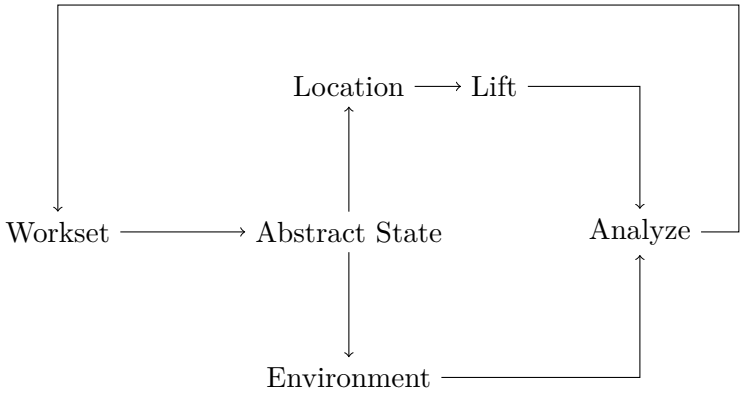


Figure 5.1.: Jakstab Analysis Process

computes the corresponding abstract locations either by using the length and location of the analyzed expression in case of a non-jump instruction, or by calling the abstract evaluation function with the produced abstract environment on the target address of a jump.

In the following sections, we will first focus on the abstract environment (Section 5.2), and then present Jakstab's intermediate language (Section 5.3). Afterwards, in Chapter 6, we describe our BDDStab Jakstab adapter, which uses the BDDStab library to implement BDD-based value analysis for Jakstab.

5.2. Abstract Environment

To define a common abstract environment for a value analysis in Jakstab, let us first consider the shape of a concrete environment as depicted in Figure 5.2. Formally, a state is a combination of a program location together with an environment, i.e., an assignment of each storage location to a value. In the following, we focus on the environment, as the location is part of Jakstab's location CPA and must therefore not be reimplemented by a value analysis plug-in. Further, we only consider integral values, as it is uncommon for floating point values to be converted to integral values and used in indirect jumps. A concrete environment consists of a `RegTable`, which is a function that assigns a value to each register, as well as a `Heap`, which assigns a value to each address. It is possible that there is no meaningful value for a register or memory location, as none might have been assigned. In an actual execution on a computer, accessing this location would most likely yield a random value, a behavior which can be captured in the abstract by initializing that memory location with $\top$.

An abstract environment, as visualized in Figure 5.3, is structurally equivalent to a concrete environment, with concrete integral values replaced by abstract values, which summarize a set of concrete values soundly. While in the concrete, associations between storage locations and values can be modeled by simple updatable associations, modeling `Heap` and `RegTable` in the abstract is slightly more complex.

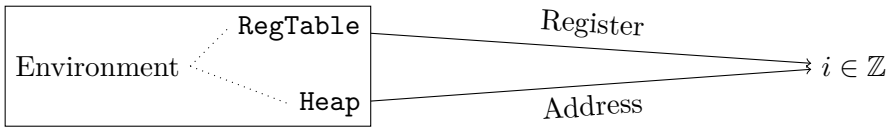


Figure 5.2.: Concrete Environment

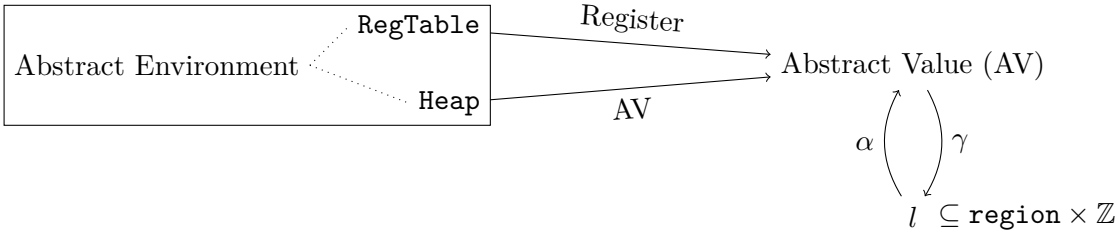


Figure 5.3.: Abstract Environment

Heap

Jakstab's heap model supports abstract base addresses, called regions, which are generated by memory allocation statements. Additionally there always exists a global region that is used by default, e.g., when no dynamic memory management is used or the use of dynamic allocations cannot be identified in the executable. The set of possible regions forms a join semi-lattice, hence a top symbol exists, which signifies that a region could not be determined precisely. Since it is not possible to distinguish pointers from integers on the binary level, all integers are assigned the global region by default, which signifies a base address of zero. It is assumed that abstract regions cannot overlap. Therefore, if an operand address of a memory operation cannot be computed precisely enough or is approximated to $\top$, then the region information may limit the effect of the operation to the region only. The core difference between the operation of the `Heap` in the concrete is that there, values are singletons. In the abstract, each value represents a set of concrete values. Therefore, the `Heap` must support lookup and write operations on collections of addresses. Since these collections of addresses are overapproximations, it cannot be guaranteed that a write operation deletes information stored before under the given addresses. Only when the collection of target addresses for a write operation is a singleton can old information at that address be deleted safely, since even if this singleton would be spurious, execution in the concrete would not be able to perform the write operation.

RegTable

In difference to the `Heap`, the `RegTable` is not addressable, so that values cannot be accessed using computed values. Instead, values are accessed using constant register names. Therefore, write operations can always overwrite present information. X86 has registers that overlap by default, e.g., there are 8- and 16-bit sized registers that contain parts of an underlying 32-bit register.

5.3. Intermediate Language

Similar to other analysis frameworks [33, 74, 75], Jakstab uses an intermediate language to perform the analysis on. Whenever an analysis has determined a new program location as reachable, Jakstab lifts the instruction at that location to the intermediate language and performs a new analysis step which produces new successor states, which in turn contain successor locations. The iterative process of lifting and analysis continues at these new successor locations, and since the successor locations are computed as part of the analysis, the precision of the computed control flow depends on the analysis as well. However, the control flow analysis itself is hidden from the value analysis part and therefore this part does not have to handle control flow instructions such as jumps explicitly.

In Figure 5.4, we present those instructions that are exposed to the value analysis part of the analysis, such as BDDStab. The instructions can generally be categorized into expressions that operate on abstract values, and statements that operate on abstract states. For expressions, Jakstab plug-ins must provide evaluation functions that take abstract values and produce abstract values that correspond to the result. The transfer functions for statements use these evaluation functions to soundly update affected variables and memory locations corresponding to the statement.

Our BDDStab Adapter supports the underlined statement and expression types from Figure 5.4, which we describe in Table 5.1.

$$
\begin{array}{rcl}
\langle \text{Program} \rangle & \models & \langle \text{Statement} \rangle * \\
\langle \text{Statement} \rangle & \models & \begin{array}{ll}
\underline{\langle \text{VarAssignment} \rangle} & | \ \underline{\langle \text{MemAssignment} \rangle} \\
| \ \underline{\langle \text{Assume} \rangle} & | \ \langle \text{Alloc} \rangle \\
| \ \underline{\langle \text{Dealloc} \rangle} & | \ \langle \text{UnknownCall} \rangle \\
| \ \langle \text{Havoc} \rangle & | \ \langle \text{MemSet} \rangle \\
| \ \langle \text{MemCpy} \rangle &
\end{array} \\
\langle \text{Expression} \rangle & \models & \begin{array}{ll}
\underline{\langle \text{BitRange} \rangle} & | \ \underline{\langle \text{CondExpr} \rangle} \\
| \ \underline{\langle \text{MemLoc} \rangle} & | \ \underline{\langle \text{Nondet} \rangle} \\
| \ \underline{\langle \text{Number} \rangle} & | \ \underline{\langle \text{Operation} \rangle} \\
| \ \langle \text{SpecialExpr} \rangle & | \ \underline{\langle \text{Variable} \rangle}
\end{array} \\
\langle \text{Operation} \rangle & \models & \begin{array}{l}
\langle \text{Expression} \rangle == \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle < \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle \leq \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle <_u \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle \leq_u \langle \text{Expression} \rangle \\
| \ !\langle \text{Expression} \rangle \\
| \ -\langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle \&\& \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle || \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle \hat{} \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle + \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle * \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle / \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle \% \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle >>> \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle >> \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle << \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle \texttt{rol} \langle \text{Expression} \rangle \\
| \ \langle \text{Expression} \rangle \texttt{ror} \langle \text{Expression} \rangle
\end{array} \\
\langle \text{VarAssignment} \rangle & \models & \langle \text{Variable} \rangle := \langle \text{Expression} \rangle \\
\langle \text{MemAssignment} \rangle & \models & \langle \text{MemLoc} \rangle := \langle \text{Expression} \rangle \\
\langle \text{Assume} \rangle & \models & \text{assume}(\langle \text{Expression} \rangle) \\
\langle \text{Variable} \rangle & \models & \%[\text{a-c}]+ \\
\langle \text{Number} \rangle & \models & \%[\text{0-9}]+ \\
\langle \text{MemLoc} \rangle & \models & \mathbb{M}(\langle \text{Expression} \rangle) \\
\langle \text{BitRange} \rangle & \models & \text{bitRange}(\langle \text{Expression} \rangle, \langle \text{Expression} \rangle, \langle \text{Expression} \rangle) \\
\langle \text{CondExpr} \rangle & \models & \langle \text{Expression} \rangle ? \langle \text{Expression} \rangle : \langle \text{Expression} \rangle \\
\langle \text{Nondet} \rangle & \models & \text{nondet}
\end{array}
$$

Figure 5.4.: Fragment of Jakstab's IL relevant for Value Analysis

VarAssignment	Updates the given variable to the value of the given expression
MemAssignment	Updates the value stored at the given address to the value of the given expression
Assume	Prevents execution to continue if given expression evaluates to false
BitRange	Slices a given integer on the bitlevel
CondExpr	Evaluates to the second expression if the first is true and to the third otherwise
MemLoc	A heap location described by the given address
Nondet	An arbitrary value
Operation	Boolean, arithmetic, or bitwise operation
Number	A number
Variable	A variable
==	True exactly when both operands are equal
$<$	True exactly when first operand is strictly smaller than second in signed interpretation
$\leq$	True exactly when first operand is smaller or equal to second in signed interpretation
$<_u$	True exactly when first operand is strictly smaller than second in unsigned interpretation
$\leq_u$	True exactly when first operand is smaller or equal to second in unsigned interpretation
!	Bitwise or Boolean negation
$-$	Arithmetic negation
&	Bitwise or Boolean `and`
||	Bitwise or Boolean `or`
^	Bitwise or Boolean `xor`
$+$	Addition
$*$	Multiplication
/	Division
%	Modulo
>>>	Bitwise shift right
>>	Arithmetic shift right
<<	Shift left
`rol`	Rotate left
`ror`	Rotate right
`nondet`	Non-deterministic value

Table 5.1.: Description of Expressions, Statements, and Operations relevant for Value Analysis

6. BDDStab Jakstab Adapter

In the previous chapter, we introduced the Jakstab framework for binary analysis and its requirements for value analysis CPAs in the form of plug-ins. In this chapter, we use the BDDStab library (Chapter 4) in the implementation of such a plug-in. Besides the merge and stop operators, we define the transfer and abstract evaluation functions (Sections 6.1, and 6.2 respectively) on the intermediate language SSL, which provide the heavy lifting of the plug-in. The most intricate transfer function is the `assume` statement, which makes our analysis path-sensitive by producing precise restrictions for the true and false successors of conditional branches. To improve precision of this transfer function, we define an overapproximating constraint solver (Section 6.3) and an equivalence class analysis (Section 6.4) that enables the restriction of related storage locations that are not directly mentioned in the condition. Lastly, we briefly discuss widening (Section 6.5) and tie the presented analysis parts together to a CPA instance in Section 6.6.

6.1. Statements

In this section, we discuss the transfer functions for the statements of the intermediate language as presented in Figure 5.4. In the definitions, we assume an abstract evaluation function eval, which evaluates a given expression e in a given abstract state $\sigma^\#$ ($\text{eval}[\![e]\!](\sigma^\#)$). We describe the eval function in Section 6.2.

The transfer functions for the statements are as follows, where the subscript in the brackets denotes the statement type:

$$f^{\text{BDD}}_{[\alpha:=e]}(\sigma^\#) = \sigma^\#\texttt{.r}[\alpha] \coloneqq \text{eval}[\![e]\!](\sigma^\#)$$

$$f^{\text{BDD}}_{[\mathbb{M}(e_1):=e]}(\sigma^\#) = \begin{cases} \sigma^\#\texttt{.h}[m] \coloneqq \text{eval}[\![e]\!](\sigma^\#) & \text{if } \{m\} = \text{eval}[\![e_1]\!](\sigma^\#) \\ \bigsqcup\{\sigma^\#\texttt{.h}[m] \coloneqq \sigma^\#\texttt{.h}[m] & \\ \quad \sqcup \text{eval}[\![e]\!](\sigma^\#) \mid m \in \text{eval}[\![e_1]\!](\sigma^\#)\} & \text{otherwise} \end{cases}$$

$$f^{\text{BDD}}_{[\text{assume}(e)]}(\sigma^\#) = \sigma^\# \sqcap \text{solve}_o[\![e]\!](\sigma^\#)$$

The transfer function for the variable assignment updates the register table at the position described by α to the result of the evaluation function on e in $\sigma^\#$. In principle, the transfer function for variable assignment is similar to the transfer function for heap assignment. There is, however, a noticable difference: The evaluation function for the assignment to the heap only updates, and thereby deletes, the old value if its address can be resolved without ambiguity, i.e., can be resolved to a singleton set. If the heap address cannot be resolved to a singleton, then the transfer function does not perform a destructive update and joins the new with the old abstract values instead (weak update). The function is limited to weak updates in case the heap address cannot be resolved to a singleton, because the abstract evaluation function, applied to the expression providing the address, computes an overapproximated result, meaning that not each of the elements from the concretization of the set may be assumed to be feasible. Therefore, it cannot be

assumed that an update at a specific address occurred or not. The situation is different if the set of heap addresses is a singleton: Even though this singleton value could be the result of an overapproximation, and therefore have no feasible concretization, strong update is sound, since control could not reach the heap assignment in the concrete. We delay the discussion of the `assume` statement until Section 6.3.

6.2. Abstract Evaluation Function

In this section, we will use the algorithms provided by the BDDStab library to define the abstract evaluation function $\text{eval}[\![e]\!](\sigma^\#)$. We assume that constants are denoted by c, variables by v, and memory locations by $\mathbb{M}(e)$, and split the definition of the abstract evaluation function into 5 categories:

1. Constants, registers, memory locations, and Boolean and arithmetic negation (Figure 6.1)
2. Relational operations (Figure 6.2)
3. Binary bitwise and arithmetic operations (Figure 6.3)
4. Shifts and rotations (Figure 6.4)
5. Bit-level slicing, conditional expression, and non-determinism (Figure 6.5)

$$\text{eval}[\![c]\!](\sigma^\#) = \texttt{fromNumber(}c\texttt{, }\boxed{0}\texttt{, }\boxed{0}\texttt{, }\boxed{1}\texttt{)}$$
$$\text{eval}[\![v]\!](\sigma^\#) = \sigma^\#\texttt{.r}[v]$$
$$\text{eval}[\![\mathbb{M}(e)]\!](\sigma^\#) = \biguplus\{\sigma^\#\texttt{.r}[c] \mid c \in \text{eval}[\![e]\!](\sigma^\#)\}$$
$$\text{eval}[\![!e]\!](\sigma^\#) = \texttt{not(eval}[\![e]\!](\sigma^\#)\texttt{)}$$
$$\text{eval}[\![-e]\!](\sigma^\#) = \texttt{negate(eval}[\![e]\!](\sigma^\#)\texttt{)}$$

Figure 6.1.: Abstract Evaluation for Constants, Registers, Memory Locations, and Boolean and Arithmetic Negation

The abstract evaluation function for relational operators (Figure 6.2) returns $\boxed{0}$, i.e., $\bot$ for BDD-based integer sets, when the evaluation of either of the operands evaluates to $\boxed{0}$. Returning $\boxed{0}$ in this case is correct, because in the concrete, evaluation of a boolean expression must yield either true or false, and if overapproximated evaluation yields $\boxed{0}$, then execution cannot reach this program point. For $==$, the abstract evaluation function returns $(\boxed{0}\boxed{\circ}\boxed{1})$, i.e., the abstraction of false, when there does not exist a value that is in the evaluation of both expressions. Similarly, it returns $(\boxed{1}\boxed{\circ}\boxed{0})$, i.e., the abstraction of true, when the evaluation of both operands is the same singleton, as then, there does not exist any combination of elements in the concrete that are unequal. In all other cases, i.e., the intersection of the operand evalutions is neither empty nor singleton, the evaluation function returns $\boxed{1}$, i.e., $\top$ for BDD-based integer sets. The remaining

relational operators from Figure 6.2 use the minimum and maximum functions to check whether there exist elements in the evaluations of the operands that fulfill either the relation or its negation, and add ($\boxed{1}\odot\boxed{0}$), i.e., the abstraction of true, and ($\boxed{0}\odot\boxed{1}$), i.e., the abstraction of false, correspondingly.

$$\text{eval}[\![e_1 == e_2]\!](\sigma^\#) = \begin{cases} \boxed{0} & \text{if eval}[\![e_1]\!](\sigma^\#) = \boxed{0} \vee \text{eval}[\![e_2]\!](\sigma^\#) = \boxed{0} \\ (\boxed{0}\odot\boxed{1}) & \text{if eval}[\![e_1]\!](\sigma^\#) \sqcap \text{eval}[\![e_2]\!](\sigma^\#) = \boxed{0} \\ (\boxed{1}\odot\boxed{0}) & \text{if } \{v\} = \text{eval}[\![e_1]\!](\sigma^\#) = \text{eval}[\![e_2]\!](\sigma^\#) \\ \boxed{1} & \text{otherwise} \end{cases}$$

$$\text{eval}[\![e_1 < e_2]\!](\sigma^\#) = \begin{cases} \boxed{0} & \text{if eval}[\![e_1]\!](\sigma^\#) = \boxed{0} \\ & \vee\text{eval}[\![e_2]\!](\sigma^\#) = \boxed{0} \\ \biguplus(\{(\boxed{1}\odot\boxed{0}) \mid \texttt{min(eval}[\![e_1]\!](\sigma^\#)\texttt{)} & \\ \qquad < \texttt{max(eval}[\![e_2]\!](\sigma^\#)\texttt{)}\} & \text{otherwise} \\ \cup\{(\boxed{0}\odot\boxed{1}) \mid \texttt{min(eval}[\![e_2]\!](\sigma^\#)\texttt{)} & \\ \qquad \leq \texttt{max(eval}[\![e_1]\!](\sigma^\#)\texttt{)}\}) & \end{cases}$$

$$\text{eval}[\![e_1 \leq e_2]\!](\sigma^\#) = \begin{cases} \boxed{0} & \text{if eval}[\![e_1]\!](\sigma^\#) = \boxed{0} \\ & \vee\text{eval}[\![e_2]\!](\sigma^\#) = \boxed{0} \\ \biguplus(\{(\boxed{1}\odot\boxed{0}) \mid \texttt{min(eval}[\![e_1]\!](\sigma^\#)\texttt{)} & \\ \qquad \leq \texttt{max(eval}[\![e_2]\!](\sigma^\#)\texttt{)}\} & \text{otherwise} \\ \cup\{(\boxed{0}\odot\boxed{1}) \mid \texttt{min(eval}[\![e_2]\!](\sigma^\#)\texttt{)} & \\ \qquad < \texttt{max(eval}[\![e_1]\!](\sigma^\#)\texttt{)}\}) & \end{cases}$$

$$\text{eval}[\![e_1 <_u e_2]\!](\sigma^\#) = \begin{cases} \boxed{0} & \text{if eval}[\![e_1]\!](\sigma^\#) = \boxed{0} \\ & \vee\text{eval}[\![e_2]\!](\sigma^\#) = \boxed{0} \\ \biguplus(\{(\boxed{1}\odot\boxed{0}) \mid \texttt{minu(eval}[\![e_1]\!](\sigma^\#)\texttt{)} & \\ \qquad < \texttt{maxu(eval}[\![e_2]\!](\sigma^\#)\texttt{)}\} & \text{otherwise} \\ \cup\{(\boxed{0}\odot\boxed{1}) \mid \texttt{minu(eval}[\![e_2]\!](\sigma^\#)\texttt{)} & \\ \qquad \leq \texttt{maxu(eval}[\![e_1]\!](\sigma^\#)\texttt{)}\}) & \end{cases}$$

$$\text{eval}[\![e_1 \leq_u e_2]\!](\sigma^\#) = \begin{cases} \boxed{0} & \text{if eval}[\![e_1]\!](\sigma^\#) = \boxed{0} \\ & \vee\text{eval}[\![e_2]\!](\sigma^\#) = \boxed{0} \\ \biguplus(\{(\boxed{1}\odot\boxed{0}) \mid \texttt{minu(eval}[\![e_1]\!](\sigma^\#)\texttt{)} & \\ \qquad \leq \texttt{maxu(eval}[\![e_2]\!](\sigma^\#)\texttt{)}\} & \text{otherwise} \\ \cup\{(\boxed{0}\odot\boxed{1}) \mid \texttt{minu(eval}[\![e_2]\!](\sigma^\#)\texttt{)} & \\ \qquad < \texttt{maxu(eval}[\![e_1]\!](\sigma^\#)\texttt{)}\}) & \end{cases}$$

Figure 6.2.: Abstract Evaluation for Relational Operators

For multiplication in Figure 6.3, the evaluation function is defined for multiplication by constants using `mulSingl`. The general case still uses `mulSingl` if evaluation of one

operand yields a singleton set. Otherwise, we re-use the abstract multiplication function from interval arithmetic ($*^{\#[]}$) and the function `bdd2ival`, which overapproximates a BDD-based integer set by a set of covering intervals, and the function `ival2bdd` to convert the resulting intervals back to BDD-based integer sets. The parameter `p` defines the precision of the overapproximation and must be set appropriately. We use the same method to evaluate division, i.e., conversion from and to intervals and application of the interval transfer function for division ($/^{\#[]}$).

$$\text{eval}[\![e_1||e_2]\!](\sigma^\#) = \texttt{algo(eval}[\![e_1]\!](\sigma^\#)\texttt{, eval}[\![e_2]\!](\sigma^\#)\texttt{, } \vee\texttt{)}$$
$$\text{eval}[\![e_1\&\&e_2]\!](\sigma^\#) = \texttt{algo(eval}[\![e_1]\!](\sigma^\#)\texttt{, eval}[\![e_2]\!](\sigma^\#)\texttt{, } \wedge\texttt{)}$$
$$\text{eval}[\![e_1 + e_2]\!](\sigma^\#) = \texttt{add(eval}[\![e_1]\!](\sigma^\#)\texttt{, eval}[\![e_2]\!](\sigma^\#)\texttt{)}$$
$$\text{eval}[\![e * c]\!](\sigma^\#) = \texttt{mulSingl(eval}[\![e]\!](\sigma^\#)\texttt{, eval}[\![c]\!](\sigma^\#)\texttt{)}$$
$$\text{eval}[\![c * e]\!](\sigma^\#) = \texttt{mulSingl(eval}[\![e]\!](\sigma^\#)\texttt{, eval}[\![c]\!](\sigma^\#)\texttt{)}$$
$$\text{eval}[\![e_1 * e_2]\!](\sigma^\#) = \begin{cases} \texttt{mulSingl(eval}[\![e_2]\!](\sigma^\#)\texttt{, } v_1\texttt{)} & \text{if } \{v_1\} = \text{eval}[\![e_1]\!](\sigma^\#) \\ \texttt{mulSingl(eval}[\![e_1]\!](\sigma^\#)\texttt{, } v_2\texttt{)} & \text{if } \{v_2\} = \text{eval}[\![e_2]\!](\sigma^\#) \\ \biguplus\{\texttt{ival2bdd(}i_1 *^{\#[]} i_2\texttt{)} \mid & \\ i_1 \in \texttt{bdd2ival(eval}[\![e_1]\!](\sigma^\#)\texttt{, p)} & \text{otherwise} \\ i_2 \in \texttt{bdd2ival(eval}[\![e_1]\!](\sigma^\#)\texttt{, p)}\} & \end{cases}$$
$$\text{eval}[\![e_1/e_2]\!](\sigma^\#) = \begin{array}{l} \biguplus\{\texttt{ival2bdd(}i_1/^{\#[]}i_2\texttt{)} \mid i_1 \in \texttt{bdd2ival(eval}[\![e_1]\!](\sigma^\#)\texttt{, p)} \\ i_2 \in \texttt{bdd2ival(eval}[\![e_1]\!](\sigma^\#)\texttt{, p)}\} \end{array}$$

Figure 6.3.: Abstract Evaluation for Binary Bitwise and Arithmetic Operations

Abstract evaluation for shifts and rotations (Figure 6.4) is only supported if the evaluation of the second operand yields a singleton. Otherwise, the abstract evaluation function returns $\boxed{1}$, i.e., $\top$ for BDD-based integer sets.

$$\text{eval}[\![e_1 >>> e_2]\!](\sigma^\#) = \begin{cases} \texttt{shiftr(eval}[\![e_1]\!](\sigma^\#)\texttt{, } v_2\texttt{)} & \text{if } \{v_2\} = \text{eval}[\![e_2]\!](\sigma^\#) \\ \boxed{1} & \text{otherwise} \end{cases}$$
$$\text{eval}[\![e_1 >> e_2]\!](\sigma^\#) = \begin{cases} \texttt{shiftrarith(eval}[\![e_1]\!](\sigma^\#)\texttt{, } v_2\texttt{)} & \text{if } \{v_2\} = \text{eval}[\![e_2]\!](\sigma^\#) \\ \boxed{1} & \text{otherwise} \end{cases}$$
$$\text{eval}[\![e_1 << e_2]\!](\sigma^\#) = \begin{cases} \texttt{shiftl(eval}[\![e_1]\!](\sigma^\#)\texttt{, } v_2\texttt{)} & \text{if } \{v_2\} = \text{eval}[\![e_2]\!](\sigma^\#) \\ \boxed{1} & \text{otherwise} \end{cases}$$
$$\text{eval}[\![e_1 \texttt{ rol } e_2]\!](\sigma^\#) = \begin{cases} \texttt{ror(eval}[\![e_1]\!](\sigma^\#)\texttt{, } v_2\texttt{)} & \text{if } \{v_2\} = \text{eval}[\![e_2]\!](\sigma^\#) \\ \boxed{1} & \text{otherwise} \end{cases}$$
$$\text{eval}[\![e_1 \texttt{ ror } e_2]\!](\sigma^\#) = \begin{cases} \texttt{rol(eval}[\![e_1]\!](\sigma^\#)\texttt{, } v_2\texttt{)} & \text{if } \{v_2\} = \text{eval}[\![e_2]\!](\sigma^\#) \\ \boxed{1} & \text{otherwise} \end{cases}$$

Figure 6.4.: Abstract Evaluation for Shifts and Rotations

The evaluation function for the `bitRange` expression (Figure 6.5) returns $\boxed{1}$, i.e., $\top$ for

BDDs, if the lower and upper bit position cannot be resolved to a singleton. Otherwise, it calls the `bitExtract` algorithm with the computed bit positions. The evaluation of the ternary operator must include the evaluation of e_2, if the evaluation of e_1 includes 1, and the evaluation of e_3 if the evaluation of e_1 includes 0. The corresponding evaluation function implements this behavior by unioning a set of at most two BDDs, one for the evaluation of e_1 and one for the evaluation of e_3.

$$\text{eval}[\![\texttt{bitRange}(e_1, e_2, e_3)]\!](\sigma^\#) = \begin{cases} \texttt{bitExtract}(V_3) & \text{if } \begin{array}{l} \{v_1\} = \text{eval}[\![e_1]\!](\sigma^\#) \\ \wedge \{v_2\} = \text{eval}[\![e_2]\!](\sigma^\#) \\ V_3 = \text{eval}[\![e_3]\!](\sigma^\#) \end{array} \\ \boxed{1} & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{eval}[\![e_1 \ ? \ e_2 : e_3]\!](\sigma^\#) = \ & \biguplus(\{\text{eval}[\![e_2]\!](\sigma^\#) \mid \\ & \qquad \text{eval}[\![e_1]\!](\sigma^\#) = \boxed{1} \vee \text{eval}[\![e_1]\!](\sigma^\#) = (\boxed{1} \boxdot \boxed{0})\} \\ & \cup \{\text{eval}[\![e_3]\!](\sigma^\#) \mid \\ & \qquad \text{eval}[\![e_1]\!](\sigma^\#) = \boxed{1} \vee \text{eval}[\![e_1]\!](\sigma^\#) = (\boxed{0} \boxdot \boxed{1})\}) \end{aligned}$$

$$\text{eval}[\![\texttt{nondet}]\!](\sigma^\#) = \boxed{1}$$

Figure 6.5.: Abstract Evaluation for Bit-level Slicing, Conditional Expressions and Nondeterminism

6.3. Assume Statement

In this section, we discuss the handling of the `assume` statement. Jakstab generates this statement from conditional branches. As an example, consider the assembly program on the left-hand side of Figure 6.6. This program first loads into %ebx the contents stored in the heap under an address given by %eax, and then compares it to 9. If the comparison determines that the contents of %ebx is greater than 9, then the code jumps to $\$g$ using jg (jump greater). Otherwise the code jumps to $\$f$ using jle (jump less or equal). In the abstract, a perfect transfer function would not only determine which transitions can be traversed in the given abstract state, but would also restrict the storage locations affected by the condition accordingly. Considering the example code in Figure 6.6 and assuming that the variation domain of %ebx is an interval from 0 to 20, the jump conditions of both jumps are determined as possibly true, whereas if the VD is 0 to 9, then only the transition to $\$f$ should be determined as possible. Assuming that the address stored in %ebx can exactly be determined (its VD is singleton), the analysis should derive that both %ebx and the abstract value stored under the address in %ebx should be restricted to 0 to 9 for the transition to $\$f$, and 10 to 20 for $\$g$.

Since the abstract domains operate on Jakstab's intermediate language SSL, it is necessary to look at the corresponding SSL code. The right-hand side of Figure 6.6 shows the SSL code from the assembly on the left. This SSL code makes the computation of the registers explicit, which is hidden in the assembly code. The compare instruction gets compiled to a subtraction, the result of which is used in the computation of the carry

```
mov (%eax) %ebx
cmpl $9 %ebx
jg g
jle f
```

```
ebx := mem32[(eax)]
tmp1 := (ebx + -9)
CF := (ebx u< 9)
OF := ((0 < tmp1) & (ebx < 0))
SF := (tmp1 < 0)
ZF := (ebx == 9)
{
assume ((SF == OF) & (!ZF))
call g
} or {
assume (! ((SF == OF) & (!ZF)))
call f
}
```

Figure 6.6.: Conditional Branching in Assembly and SSL

```
ebx := mem32[(eax)] := 9
tmp1 := (9 + -9) := 0
CF := (9 u< 9) := 0
OF := ((0 < 0) & (9 < 0)) := 0
SF := (9 < 0) := 0
ZF := (9 == 9) := 1
{
assume ((0 == 0) & (!1))
call g
} or {
assume (! ((0 == 0) & (!1)))
call f
}
```

```
ebx := mem32[(eax)] := 10
tmp1 := (10 + -9) := 1
CF := (10 u< 9) := 0
OF := ((0 < 1) & (10 < 0)) := 0
SF := (1 < 0) := 0
ZF := (10 == 9) := 0
{
assume ((0 == 0) & (!0))
call g
} or {
assume (! ((0 == 0) & (!0)))
call f
}
```

Figure 6.7.: Conditional Branching in SSL for %ebx = 9 (left) and 10 (right)

flag (CF), the overflow flag (OF), the sign flag (SF), and the zero flag (ZF). Afterwards, the `assume` statements select whether control can reach their successors. As an example, the left side of Figure 6.7 lists the evaluation of the SSL code for %ebx= 9, where all flags are assigned to 0, with the exception of the zero flag. Therefore, the expression of the `assume` clause that guards the execution of $\$g$ is evaluated to 0, and, conversely, the `assumes` clause that guards $\$f$ can be traversed. Additionally, the right side of Figure 6.7 lists the evaluation of the SSL code for %ebx= 10, where all flags are assigned to 0. In this case, the expression of the `assume` clause that guards the execution of $\$g$ is evaluated to 1, and therefore only the transition to $\$g$ is possible.

It is challenging to restrict %ebx and mem32[(eax)] for the successor of an `assume` statement, because the corresponding condition does not refer to these memory locations explicitly. To rewrite the `assume` statements to formulas over the original entities, Jakstab includes the forward substitution domain, which keeps the value of the flags and

```
%ebx := mem32[(%eax)]
tmp1 := (%ebx + -9)
CF := (%ebx u< 9)
OF := ((0 < tmp1) & (%ebx < 0))
SF := (tmp1 < 0)
ZF := (%ebx == 9)
{
assume ((((%ebx + -9) < 0) == ((0 < (%ebx + -9)) & (%ebx < 0)))
       & (!(%ebx == 9)))
call g
} or {
assume (! ((((%ebx + -9) < 0) == ((0 < (%ebx + -9)) & (%ebx < 0)))
       & (!(%ebx == 9))))
call f
}
```

Figure 6.8.: Conditional Branching in SSL with Forward Substitution

temporary variables symbolic. This domain therefore keeps a set of valid associations between flags, temporary variables, and storage locations per location. The associations become invalid when any of the included storage locations or flags gets assigned a new value. Figure 6.8 shows the corresponding assembly and SSL with enabled forward substitution. Even though the original condition was essentially %ebx < 10, where it is obvious how to restrict the VD of %ebx, it is not obvious how to do that on forward substituted assume clauses.

In order do handle complex conditions that result from forward substitution, we use two techniques:

1. Simplification/backtranslation
2. Overapproximating constraint solving

Specifically, we use Jakstab's simplifier to produce a simpler constraint and afterwards use BDDStab's solver that takes a constraint and computes an overapproximated abstract state that represents at least all concrete states that fulfill the constraint. The transfer functions for `assume` statements compute the meet of this abstract state with the incoming abstract state to produce a restriction, i.e., exclude concrete states that do not fulfill the constraint. One advantage of computing such an overapproximated abstract state first and then use the meet on incoming abstract states is that the condition does not change during the analysis, and therefore the restricting abstract state can be cached.

Technique 1: Simplification/Backtranslation Whenever a complex formula in an `assume` statement is the result of a translation and forward substitution from a simple constraint such as $a < b$, it is possible to use simplification rules to translate back to

that simple form. We have changed Jakstab's simplification rules towards exact backtranslation from complex constraint formulas to simple relational conditions. The rules we use are derived from the code that GCC generates for conditions that use the basic relational operators. Instead of matching a complete formula, we provide simplification rules that, together, perform complete backtranslation.

Whenever the value of the flags has not been computed using test or comparison statements, complete backtranslation is not feasible. In this case, not enough simplification rules are applicable and the best we can hope for is a less complex, though not yet simple constraint. We therefore introduce a simple overapproximating constraint solver that takes a constraint formula, and computes an overapproximated VD for each included variable that contains at least all values that are feasible according to the formula. BDDStab uses this solver to compute restrictions on all `assume` statements after simplification. We now describe this solver formally.

Technique 2: Overapproximating Solver We formulate our solver in terms of two functions, namely solve_o and solve_u, given in Equation 6.2. These functions take a constraint and an abstract state and produce an abstract state that represents all concrete states that fulfill this constraint. Note that we use $\top$ and $\bot$ as the state that assigns all memory locations to $\boxed{1}$ and $\boxed{0}$ respectively. Hence, $\top[a] \coloneqq 1$ is a new abstract state that has a set to 1, and all other locations set to $\boxed{1}$, which is $\top$ in the lattice of BDD-based integer sets. Consequently, $(\top[a] \coloneqq 1)[b] \coloneqq 2$ is an abstract state that has a set to 1, b set to 2, and all other locations set to $\boxed{1}$. In the definitions we assume that a and b are either memory locations with a singleton VD, or registers. In all cases where a memory location with a non-singleton VD or a constant is to be assigned, this assignment shall have no effect. The solve_o function produces the required overapproximation, while solve_u produces an underapproximation that, when complemented, produces a valid overapproximation. To produce an overapproximation of a negated condition, we use the underapproximating solve_u function. Negated constraints may either appear naturally, or from rewriting of disjunctions via De Morgan's law. The reason why an underapproximation is needed lies in Equation 6.1. Assuming X to be an exact set, and $X \cup O$ an overapproximated set with spurious elements $O \setminus X$, complementation of $X \cup O$ yields $\overline{X} \setminus O$, which is clearly an underapproximation. Similarly, complementing the underapproximated set $X \setminus O$ yields $\overline{X} \cup O$, itself clearly an overapproximation. Hence, whenever an overapproximation is needed for a negated condition, then we must complement the result of solve_u, and, conversely, when an underapproximation is needed for a negated condition, then we must complement the result of solve_o. Therefore, the recursive solver must track which kind of approximation is needed, and call its appropriate variant.

$$\begin{array}{rclcl} \overline{X \cup O} & = & \overline{X} \cap \overline{O} & = & \overline{X} \setminus O \\ \overline{X \setminus O} & = & X \cap \overline{O} & = & \overline{X} \cup O \end{array} \tag{6.1}$$

$$
\begin{array}{lcl}
\text{solve}_o[\![a == b]\!](\sigma^\#) & = & (\top[a] \coloneqq \text{eval}[\![b]\!](\sigma^\#))[b] \coloneqq \text{eval}[\![a]\!](\sigma^\#) \\
\text{solve}_u[\![a == b]\!](\sigma^\#) & = & \bot \\
\text{solve}_o[\![a < b]\!](\sigma^\#) & = & (\top[a] \coloneqq \{x \mid x < \texttt{max(eval}[\![b]\!](\sigma^\#)\texttt{)}\})[b] \\
 & \coloneqq & \{x \mid \texttt{min(eval}[\![a]\!](\sigma^\#)\texttt{)} < x\} \\
\text{solve}_u[\![a < b]\!](\sigma^\#) & = & (\bot[a] \coloneqq \{x \mid x < \texttt{min(eval}[\![b]\!](\sigma^\#)\texttt{)}\})[b] \\
 & \coloneqq & \{x \mid \texttt{max(eval}[\![a]\!](\sigma^\#)\texttt{)} < x\} \\
\text{solve}_o[\![a <_u b]\!](\sigma^\#) & = & (\top[a] \coloneqq \{x \mid x <_u \texttt{maxu(eval}[\![b]\!](\sigma^\#)\texttt{)}\})[b] \\
 & \coloneqq & \{x \mid \texttt{minu(eval}[\![a]\!](\sigma^\#)\texttt{)} < x\} \\
\text{solve}_u[\![a <_u b]\!](\sigma^\#) & = & (\bot[a] \coloneqq \{x \mid x <_u \texttt{minu(eval}[\![b]\!](\sigma^\#)\texttt{)}\})[b] \\
 & \coloneqq & \{x \mid \texttt{maxu(eval}[\![a]\!](\sigma^\#)\texttt{)} < x\} \\
\forall x : \text{solve}_x[\![a \leq b]\!](\sigma^\#) & = & \text{solve}_x[\![!(b < a)]\!](\sigma^\#) \\
\forall x : \text{solve}_x[\![a \leq_u b]\!](\sigma^\#) & = & \text{solve}_x[\![!(b <_u a)]\!](\sigma^\#) \\
\forall x : \text{solve}_x[\![A \;\&\; B]\!](\sigma^\#) & = & \text{solve}_x[\![A]\!](\sigma^\#) \sqcap \text{solve}_x[\![B]\!](\sigma^\#) \\
\forall x : \text{solve}_x[\![A \;||\; B]\!](\sigma^\#) & = & \text{solve}_x[\![!(!A \;\&\; !B)]\!](\sigma^\#) \\
\text{solve}_o[\![!A]\!](\sigma^\#) & = & \overline{\text{solve}_u[\![A]\!](\sigma^\#)} \\
\text{solve}_u[\![!A]\!](\sigma^\#) & = & \overline{\text{solve}_o[\![A]\!](\sigma^\#)}
\end{array}
\tag{6.2}
$$

As an example, the solve_o function for constraints of the form $a < b$ creates a state restricting only the two included variables, i.e., a and b. It modifies the value stored for these variables in the $\top$ state, setting a to all values smaller than the maximum value in b's variation domain, because all these values will satisfy the constraint for all values of b. Similarly, b is set to all values greater than the minimum of a's variation domain. Note that the BDD-based integer set of the abstract value for a and b is constructed efficiently using the interval to BDD conversion algorithm (Algorithm 4).

As a further example, let us compute a restriction for our example from Figure 6.6, assuming that backtranslation was able to translate back to %ebx ≤ 9 and !(%ebx ≤ 9). For the example, we will assume that the abstract state before the restriction is $\sigma = \top[\%ebx] := [0\,..\,20]$. The solver proceeds as follows:

$$
\begin{array}{l}
\text{solve}_o[\![\%\text{ebx} \leq \$9]\!](\sigma) \\
\quad = \text{solve}_o[\![!(\$9 < \%\text{ebx})]\!](\sigma) \\
\quad = \overline{\text{solve}_u[\![\$9 < \%\text{ebx}]\!](\sigma)} \\
\quad = \overline{\bot[\%\text{ebx}] \coloneqq \{x \mid \max(\text{eval}[\![\$9]\!](\sigma)) < x\}} \\
\quad = \overline{\bot[\%\text{ebx}] \coloneqq \{x \mid \$9 < x\}} \\
\quad = \overline{\bot[\%\text{ebx}] \coloneqq [\$10\,..\,]} \\
\quad = \top[\%\text{ebx}] \coloneqq [\,..\,\$9]
\end{array}
$$

Note that the solve functions, when operating on $a < b$, would compute a restriction for a and b, but in this example, a is a constant that cannot be assigned. As described before, we assume that assignments to constants have no effect, hence we may omit this assignment without changing the result.

$$
\begin{aligned}
&\text{solve}_o[\![!(\%\text{ebx} \leq \$9)]\!](\sigma) \\
&\quad = \overline{\text{solve}_u[\![\%\text{ebx} \leq \$9]\!](\sigma^\#)} \\
&\quad = \overline{\text{solve}_u[\![!(\$9 < \%\text{ebx})]\!](\sigma^\#)} \\
&\quad = \text{solve}_o[\![\$9 < \%\text{ebx}]\!](\sigma^\#) \\
&\quad = \top[\%ebx] \coloneqq \{x \mid \texttt{min(eval}[\![\$9]\!](\sigma^\#)\texttt{)} < x\} \\
&\quad = \top[\%ebx] \coloneqq \{x \mid \$9 < x\} \\
&\quad = \top[\%ebx] \coloneqq [\$10\,..\,]
\end{aligned}
$$

The resulting abstract state from our example computation contains $\top$ for all elements except %ebx. Therefore, applying the meet operation with another state will only restrict %ebx since $X \sqcap \top = X$. The transfer function for `assume` statements computes the meet of the incoming abstract state and the one computed by the solver to produce the outgoing abstract state. In our example, the abstract state after applying the meet is $\sigma^{true} = \top[\%ebx] := [0\,..\,9]$ for the true successor, and $\sigma^{false} = \top[\%ebx] := [10\,..\,20]$ for the false successor. This result matches our expectation except that if %eax in Figure 6.6 was a singleton, then we would also expect a restriction of the value stored in the heap under this singleton address. Before we define the transfer function for `assume` statements in Section 6.4, we introduce an equivalence class analysis that allows the restriction of storage locations that can be proved to be equivalent to a storage location for which a restriction applies.

6.4. Equivalence Class Analysis

In Section 6.3 we discussed how conditional branches are translated to `assume` statements and how restrictions for these statements can be computed using an overapproximating constraint solver [76]. The presented method will only compute restrictions for those registers or memory locations that directly appear in the given constraint. It will not restrict the abstract values of any related values, including those that have an equivalence relation to a restricted storage location, i.e., must have the same value at runtime as that storage location. The behavior of the transfer functions for `assume` statements is therefore the same as for non-relational value analyses on structured programs. In structured programs, equivalence relations are rare. When a structured program assigns a temporary variable to another left value, and subsequently uses this temporary variable in a condition, it is likely that this temporary variable has no other use than being another name for the left value and is therefore never reassigned. In this scenario, an analyzer could simply rewrite the program to use the left value directly and the relation between the temporary variable and the left value would not be lost during analysis. This rewriting technique is not applicable to low-level programs such as machine code because there exists only a limited number of variables. These variables, namely registers, are commonly reused, thereby invalidating the equivalence relation, and hence the validity of the rewrite. Therefore, we introduce an equivalence class analysis that computes for each program location a set of equivalence relations between registers and memory locations that must hold. This equivalence class analysis is formulated as a data flow analysis and, since it requires an abstract evaluation function to determine possible addresses of

memory locations and the transfer function of `assume` statements will make use of its results, runs in parallel to the value analysis. Hence we extend our abstract state with an equivalence class field, available via subscript e ($\sigma_e^\#$). As usual, we use $:=$ to create a new abstract state that differs only in the subscript e field, which is set to the right-hand side of $:=$ ($\sigma_e^\# := x$). The field $\sigma_e^\#$ contains a value from the lattice L as defined with the corresponding analysis in Figure 6.9. In this definition, we use V as the set of storage locations, i.e., the set of all registers and memory locations.

Every element l in the lattice L is a set of sets of storage locations, such that each element contains each storage location exactly once. Essentially, the sets in l represent equivalence classes, i.e., sets of storage locations that must have equal values at run time. Every storage location must belong to exactly one, possible singleton equivalence class. Assuming $V = \{r_1, r_2, r_3, r_4\}$, the lattice element $\{\{r_1, r_2, r_3\}, \{r_4\}\}$ means that r_1, r_2, and r_3 must have equal values at run time, but r_4 may or may not. In other words, the abstract state represents the equivalence relations $r_1 == r_2$, $r_1 == r_3$, and $r_2 == r_3$.

In an implementation it is not feasible to store those storage locations that are not part of any equivalence relation, e.g., r_4 in the previous example, since the set V of storage locations contains all valid heap addresses. We therefore choose an equivalent representation that only stores storage locations that are part of represented equivalence relations.

As demanded by the correctness relation R from abstract interpretation, the ordering on the lattice is constructed such that $\top$ is the least precise solution ($\forall v : vR\top$). A lattice element is greater than another lattice element when the smaller element describes all equivalence relations described by the greater element. Since $\top$ is the greatest element, it must be greater than all other elements and therefore all other elements must represent all equivalence relations from $\top$. It is therefore natural that $\top$ represents no equivalence relations, which corresponds to the case where each storage location is equivalent only to itself. It follows that $\bot$ represents all equivalence relations, which corresponds to the case where all storage locations are equivalent to all other storage locations.

The `rem` function takes an abstract state and a set of storage locations A, and removes the storage locations in A from their equivalence class in the abstract state and puts it into a singleton class. The function constructs a state that has each storage location in A in a singleton equivalence class and all other variables in one class, and joins this state with the original one. As a result, this join operation forces the variables in the singleton classes to remain in singleton classes in the output. Only equivalence classes that are subclasses of the $V \setminus A$ class may remain non-singleton. The `rem` function is used whenever the contents of a storage location gets updated. Since the value of the updated storage location may change, equivalence relations on that storage location that were valid before the update may not be valid after the update. Hence `rem` is used to remove the affected storage locations from their equivalence classes.

Conversely, the `estr` function takes two storage locations and creates a lattice element that, if joined with another lattice element, forces the two given storage locations to become a member of one equivalence class, thereby possibly combining existing classes. The function is used in the transfer functions for the `assume` statement, as well as in the

`est` function, described next.

The `est` function takes an abstract state and two storage locations, removes the first storage location from all equivalence classes using `rem`, and then forces both given storage locations to be in one equivalence class. This function is used for assignments, where the contents of one storage location gets updated to the value of another.

The corresponding transfer functions call `rem` for all possibly affected storage locations. To compute this set of possibly affected storage locations, they use the abstract evaluation function from the value analysis. Only if this evaluation function is able to resolve all heap locations to singletons, do the transfer functions establish corresponding equivalence relations. In Figure 6.9, as before, α denotes register names, and e denotes expressions. To distinguish the transfer functions for the equivalence class analysis from other domains, such as our BDD-based integer domain, we name them by using EQ in superscript position.

As an example, let us assume the incoming data flow to an assignment is $\sigma^{\#} = \{\{\texttt{\%ecx}, m\}, \{\texttt{\%eax}, \texttt{\%ebx}\}\}$ and the assignment itself assigns the contents of `%eax` to a memory location given by the expression e ($[\mathbb{M}(e) := \texttt{\%eax}]$). Let us further assume that the value analysis can resolve the VD of e to a singleton set $\{m\}$, and that the set V of storage locations is set to $\{\texttt{\%eax}, \texttt{\%ebx}, \texttt{\%ecx}, m\}$. As result, we intuitively expect that m, `%eax`, and `%ebx` are in an equivalence relation, and that `%ecx` is not. The analysis calls the corresponding transfer function as follows:

$$
\begin{aligned}
&\texttt{rem}(\sigma^{\#}, \{m\}) \\
&= \sigma^{\#} \sqcup \{\{m\}, \{\texttt{\%eax}, \texttt{\%ebx}, \texttt{\%ecx}\}\} \\
&= \{\{\texttt{\%ecx}, m\}, \{\texttt{\%eax}, \texttt{\%ebx}\}\} \sqcup \{\{m\}, \{\texttt{\%eax}, \texttt{\%ebx}, \texttt{\%ecx}\}\} \\
&= \{\{m\}, \{\texttt{\%ecx}\}, \{\texttt{\%eax}, \texttt{\%ebx}\}\}
\end{aligned}
$$

$$
\begin{aligned}
&f^{\mathrm{EQ}}_{[\mathbb{M}(e):=\texttt{\%eax}]}(\sigma^{\#}) \\
&= \texttt{est}(\sigma^{\#}, m, \texttt{\%eax}) \\
&= \texttt{rem}(\sigma^{\#}, \{m\}) \sqcap \{\{m, \texttt{\%eax}\}, \{\texttt{\%ebx}\}, \{\texttt{\%ecx}\}\} \\
&= \quad \{\{m\}, \{\texttt{\%ecx}\}, \{\texttt{\%eax}, \texttt{\%ebx}\}\} \\
&\quad \sqcap \{\{m, \texttt{\%eax}\}, \{\texttt{\%ebx}\}, \{\texttt{\%ecx}\}\} \\
&= \{\{m, \texttt{\%eax}, \texttt{\%ebx}\}, \{\texttt{\%ecx}\}\}
\end{aligned}
$$

This result is exactly as we expected.

Figure 6.9 also defines transfer functions for the `assume` statement, as they can also establish equivalence relations, if their constraint uses the equality operator. In difference to assignments, `assume` statements do not update any storage location's contents, hence the `rem` function is not called since no equivalence relation must be invalidated.

Using Equivalence Relations

In Section 6.3, we formulated our expectations on a good transfer function for the `assume` statement in Figure 6.6. Such a transfer function should not only restrict directly affected storage locations, but also those that are transitively equivalent. The equivalence

$$L = \{X \mid X \subseteq \mathcal{P}(V), \bigcup X = V, \\ \forall x_\mathrm{i}, x_\mathrm{j} \in X : (x_\mathrm{i} = x_\mathrm{j}) \vee (x_\mathrm{i} \cap x_\mathrm{j} = \emptyset)\}$$
$$X \sqsubseteq_L Y \iff (\forall x \in X, y \in Y : x \cap y \neq \emptyset \implies y \subseteq x)$$

$$\mathtt{rem}(\sigma^\#, A) = \sigma_e^\# \coloneqq \sigma_e^\# \sqcup (\{\{v\} \mid v \in A\} \cup \{V \backslash A\})$$

$$\mathtt{estr}(a, b) = \{\{a, b\}\} \cup \{\{v\} | v \in V \backslash \{a, b\}\}$$

$$\mathtt{est}(\sigma^\#, a, b) = \sigma_e^\# \coloneqq \mathtt{rem}(\sigma^\#, \{a\}) \sqcap \mathtt{estr}(a, b)$$

$$f^{\mathrm{EQ}}_{[\alpha_1 := \alpha_2]}(\sigma^\#) = \mathtt{est}(\sigma^\#, \alpha_1, \alpha_2)$$

$$f^{\mathrm{EQ}}_{[\mathbb{M}(e) := \alpha]}(\sigma^\#) = \begin{cases} \mathtt{est}(\sigma^\#, m, \alpha) & \text{if } \{m\} = \mathrm{eval}[\![e]\!](\sigma^\#) \\ \mathtt{rem}(\sigma^\#, \mathrm{eval}[\![e]\!](\sigma^\#)) & \text{otherwise} \end{cases}$$

$$f^{\mathrm{EQ}}_{[\alpha := \mathbb{M}(e)]}(\sigma^\#) = \begin{cases} \mathtt{est}(\sigma^\#, \alpha, m) & \text{if } \{m\} = \mathrm{eval}[\![e]\!](\sigma^\#) \\ \mathtt{rem}(\sigma^\#, \{\alpha\}) & \text{otherwise} \end{cases}$$

$$f^{\mathrm{EQ}}_{[\mathbb{M}(e_1) := \mathbb{M}(e_2)]}(\sigma^\#) = \begin{cases} \mathtt{est}(\sigma^\#, m_1, m_2) & \text{if m} \\ \mathtt{rem}(\sigma^\#, \mathrm{eval}[\![e_1]\!](\sigma^\#) \cup \mathrm{eval}[\![e_2]\!](\sigma^\#)) & \text{otherwise} \end{cases}$$
where m $\iff \{m_1\} = \mathrm{eval}[\![e_1]\!](\sigma^\#) \wedge \{m_2\} = \mathrm{eval}[\![e_2]\!](\sigma^\#)$

$$f^{\mathrm{EQ}}_{[\alpha := \_]}(\sigma^\#) = \mathtt{rem}(\sigma^\#, \{\alpha\})$$

$$f^{\mathrm{EQ}}_{[\mathbb{M}(e) := \_]}(\sigma^\#) = \mathtt{rem}(\sigma^\#, \mathrm{eval}[\![e]\!](\sigma^\#))$$

$$f^{\mathrm{EQ}}_{[\mathrm{assume}(\alpha_1 == \alpha_2)]}(\sigma^\#) = \sigma_e^\# \coloneqq \sigma_e^\# \sqcap \mathtt{estr}(\alpha_1, \alpha_2)$$

$$f^{\mathrm{EQ}}_{[\mathrm{assume}(\mathbb{M}(e) == \alpha)]}(\sigma^\#) = \begin{cases} \sigma_e^\# \coloneqq \sigma_e^\# \sqcap \mathtt{estr}(m, \alpha) & \text{if } \{m\} = \mathrm{eval}[\![e]\!](\sigma^\#) \\ \sigma^\# & \text{otherwise} \end{cases}$$

$$f^{\mathrm{EQ}}_{[\mathrm{assume}(\alpha == \mathbb{M}(e))]}(\sigma^\#) = \begin{cases} \sigma_e^\# \coloneqq \sigma_e^\# \sqcap \mathtt{estr}(\alpha, m) & \text{if } \{m\} = \mathrm{eval}[\![e]\!](\sigma^\#) \\ \sigma^\# & \text{otherwise} \end{cases}$$

$$f^{\mathrm{EQ}}_{[\mathrm{assume}(\mathbb{M}(e_1) == \mathbb{M}(e_2))]}(\sigma^\#) = \begin{cases} \sigma_e^\# \coloneqq \sigma_e^\# \sqcap \mathtt{estr}(m_1, m_2) & \text{if } m \\ \sigma^\# & \text{otherwise} \end{cases}$$
where m $\iff \{m_1\} = \mathrm{eval}[\![e_1]\!](\sigma^\#) \wedge \{m_2\} = \mathrm{eval}[\![e_2]\!](\sigma^\#)$

$$f^{\mathrm{EQ}}_{[\_]}(\sigma^\#) = \sigma^\#$$

Figure 6.9.: Equivalence Class Analysis

analysis from Section 6.4 determines those relations. Whenever the solver computed a restriction for one storage location, we can now select the equivalence class of that location and restrict all members of it the same way.

We are now ready to define a transfer function for the `assume` statement that is more precise than that from Section 6.1. The following transfer function first creates a new restricting abstract state by using the constraint solver from Section 6.3. To apply the computed restrictions not only to the storage locations directly appearing in the expression e, we compute the meet of all restrictions for the variables that are within one equivalence class. The function then uses the result to restrict the incoming abstract state $\sigma^\#$.

$$f_{[\text{assume}(e)]}(\sigma^\#) = \sigma^\# \sqcap s$$
$$\text{where } s = \bigsqcap\{\top[v_1] \coloneqq \text{solve}_o[\![e]\!](\sigma^\#)[v_2] \mid v_1 \in c, v_2 \in c, c \in \sigma^\#_e\}$$

As an example, let us assume that the solver has computed the restricting state $\top[\texttt{\%eax}] \coloneqq [0\,..\,5]$ and that the equivalence analysis determined $\{\{\texttt{\%eax}, \texttt{\%ebx}\}, \{\texttt{\%ecx}\}\}$. In this case, the transfer function should restrict the abstract state $(\top[\texttt{\%eax}] \coloneqq [0\,..\,5])[\texttt{\%ebx}] \coloneqq [0\,..\,5]$, i.e., a state that restricts both `%eax` and `%ebx` equivalently. Computing s in this case proceeds as follows:

$$\bigsqcap\{\top[\texttt{\%eax}] \coloneqq [0\,..\,5], \top[\texttt{\%eax}] \coloneqq \top, \top[\texttt{\%ebx}] \coloneqq [0\,..\,5], \top[\texttt{\%ebx}] \coloneqq \top, \top[\texttt{\%ecx}] \coloneqq \top\} =$$
$$(\top[\texttt{\%eax}] \coloneqq [0\,..\,5])[\texttt{\%ebx}] \coloneqq [0\,..\,5]$$

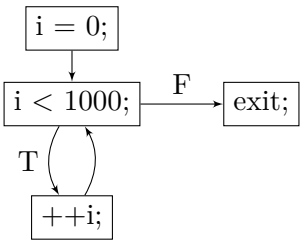
Equivalence relations can also be used to enhance the precision of transfer functions other than that for the `assume` statement. If their operands are equivalent in the concrete, then all binary transfer functions could change their behavior from computing their result via the Cartesian product to computation using element-by-element traversal. Additionally, whenever two storage locations are equivalent in the concrete, it is admissible to intersect their abstract values, since any represented value is infeasible that is not in both abstract values.

6.5. Widening

To ensure termination, it is common in abstract interpretation-based analyses to use a widening operator. Without this widening operator, termination can either not be guaranteed at all or may take unacceptably long. Formally, a widening operator (∇) is an upper bound operator ($a \nabla b \sqsupseteq a \wedge a \nabla b \sqsupseteq b$) that ensures that each ascending chain eventually stabilizes. Although not required by Jakstab, BDDStab therefore also defines a widening operator.

On traditional data flow analyses, the widening operator is commonly applied at confluence points during the analysis. The reason for applying widening at confluence points is to avoid iterating on loops in the CFG of the analyzed program. Consider the example from Figure 6.10, which shows a CFG of a structured program, and the progression of the variation domain of the variable i after the conditional expression.

Without widening, we can see that the analysis of the loop will stop after 1000 iterations, which are as many iterations as the program would execute in the concrete. Hence, widening is needed to speed up the termination.



#	VD(i) after condition, w/o widening	VD(i) after condition, w/ widening
1	[0 .. 0]	[0 .. 0]
2	[0 .. 1]	[0 .. 1]
3	[0 .. 2]	[0 .. 10]
4	[0 .. 3]	[0 .. 11]
5	[0 .. 4]	[0 .. 100]
	...	...
8	[0 .. 7]	[0 .. 1000]

Figure 6.10.: CFG with Loop

The most trivial widening immediately returns $\top$, which is greater or equal to any other value, therefore correctly describes any other value, and hence is a valid result for a widening operator. However, it is obvious that an operator that immediately returns $\top$ will lead to unacceptable precision loss. A less destructive variation on the naive widening operator will return $\top$ only after n iterations, which may give the analysis enough iterations to terminate on its own. The disadvantage with this approach is that if n iterations are not enough to terminate the analysis, then precision is destroyed completely. It is therefore desirable to gradually decrease precision, compromising between runtime and precision of the analysis.

When precision is to be decreased gradually by adding elements to the variation domains, a heuristics is required that determines which elements are added. The most common heuristics compare the abstract states from two consecutive iterations, and add elements where changes happened. How many elements are added commonly grows with the number of widenings that have been executed already, yielding a progressively aggressive widening strategy.

The widening in Figure 6.10 adds elements progressively at iteration 3, 5, and 8, and can therefore terminate the loop after fewer than 10 iterations. In this case, widening computed a set containing exactly those elements that fulfill the loop condition. That is, however, not necessarily the case. Had the loop condition been $i < 900$, and widening had been executed the same way, then the variation domain of i at the false exit of the loop would have been determined as $i \in [900 .. 1000]$, which is an overapproximation by 100 elements.

From this example, we can derive an alternative widening scheme at conditional loops [12], which generates a widening from the loop condition by widening to a state representing all concrete states that fulfill the condition. In the previous example, optimal widening would widen i to $[0 .. 899]$. Within BDDStab, such widening could be implemented easily using the solver that we described in Section 6.3.

Instead, the widening strategy of the BDDStab adapter uses bisection to find changes between abstract states from two consecutive iterations. BDDs support bisection efficiently, as they support $O(1)$ equivalence checking and their structure is derived from the Shannon expansion, itself a form of applied bisection. The widening operator recurses into the changed sub-BDDs to a depth determined by the number of previous widenings, and replaces the found sub-BDD by $\boxed{1}$, thereby adding more elements and reducing the BDD size at the same time.

The implementation of the widening operator of the BDDStab adapter was done as part of a bachelor thesis, which contains a precise description of the operator [77].

6.6. CPA Configuration for BDDStab

After defining the transfer functions for Jakstab's SSL, we are now ready to define a CPA instance for BDDStab. As discussed previously, the CPA formulation that Jakstab uses allows the combination of analyses, facilitated by the required merge and join operators. By default, BDDStab uses the $\text{merge}^{\text{join}}$ and $\text{stop}^{\text{join}}$ operators, i.e., a configuration corresponding to classic DFA. Additionally, Jakstab, by default, combines any analysis with a location analysis, suitable for the analysis of executables. In this configuration, BDDStab constitutes a classic DFA, with the difference that the CFG is built within the location analysis and therefore is not static. Figure 6.11 contains the CPA instance for the BDDStab adapter, where $\mathcal{E}_{\text{BDD}}$ is the described composition of an abstract variable table and a heap model, both using BDD-based integer sets for abstract values. As usual, each abstract BDD state is paired with an abstract location, which we treat as equivalent to the disassembled instruction at this position. The transfer relation $\rightsquigarrow_{\text{BDD}}$ is defined between combinations of location and BDD state, exactly when the right-hand side location is reachable from the left-hand side, and the transfer function of the BDDStab adapter maps the left-hand BDD state to the right-hand side BDD state. The merge operator merges only states with identical locations and the stop operator is true exactly when the join of all states for a location covers the new state. Note that as before, we do not provide the domain of concrete states C and the concretization function $[\![\cdot]\!]$, as they are not needed for the execution of the analysis. Furthermore, we assume the location analysis as implemented in Jakstab, i.e., we assume that it implicitly uses our abstract evaluation function to resolve jump targets.

$$\mathcal{E}_{\text{BDD}} = (\mathcal{E}_{\mathbb{L}} \times \mathcal{E}_{\text{BDDSet}})$$

$$(l_1, \sigma_1^{\#\text{BDD}}) \rightsquigarrow_{\text{BDD}} (l_2, \sigma_2^{\#\text{BDD}}) \iff l_1 \rightsquigarrow_{\mathbb{L}} l_2 \wedge f_{l_1}^{\text{BDD}}(f_{l_1}^{\text{EQ}}(\sigma_1^{\#\text{BDD}})) = \sigma_2^{\#\text{BDD}}$$

$$\text{merge}_{\text{BDD}}((l_1, \sigma_1^{\#\text{BDD}}), (l_2, \sigma_2^{\#\text{BDD}})) = \begin{cases} (l_1, \text{merge}^{\text{join}}(\sigma_1^{\#\text{BDD}}, \sigma_2^{\#\text{BDD}})) & \text{if } l_1 = l_2 \\ (l_2, \sigma_2^{\#\text{BDD}}) & \text{otherwise} \end{cases}$$

$$\text{stop}_{\text{BDD}}((l_1, \sigma^{\#\text{BDD}}), R) = \text{stop}^{\text{join}}(\sigma^{\#\text{BDD}}, \{\sigma_2^{\#\text{BDD}} \mid (l_2, \sigma_2^{\#\text{BDD}}) \in R, l_2 = l_1\})$$

$$\mathbb{D}_{\text{BDD}} = ((C, \mathcal{E}_{\text{BDD}}, [\![\cdot]\!]), \rightsquigarrow_{\text{BDD}}, \text{merge}_{\text{BDD}}, \text{stop}_{\text{BDD}})$$

$$\iota_{\text{BDD}} = (l, \emptyset) \text{ where } l \text{ is an entry node of the analyzed program}$$

Figure 6.11.: BDDStab CPA

6.7. Implementation of BDDStab Jakstab Adapter

The implementation of the Jakstab adapter consists of 6 Java classes. The main entry point to the analysis is the BDDTracking class that contains the CPA instance. This CPA instance uses abstract environments, implemented in the BDDState class, which contains the transfer function and abstract evaluation functions, and therefore contains the adapter's main functionality. The BDDSet class, which implements abstract values, wraps a BDD-based integer set from the BDDStab library, and re-exports most of its functionality. Since Jakstab's output contains string representations of abstract values, the BDDSet class must provide a pretty printer. However, since abstract values in BDDStab are capable of representing many concrete elements, and the output as BDD is not helpful, we only explicitly show sets with up to 100 elements. Should the set contain more elements, then its cardinality is displayed instead.

Usage

The analysis is activated using "z" as a parameter to Jakstab's "–cpa" option. To activate the restriction of flow data at conditional branches (Assume statement), the analysis must be combined with Jakstab's forward substitution analysis, which is done by additionally passing "f" to "–cpa". The analysis additionally supports a threshold parameter ("–bdd-threshold"), that determines when widening should take place, and a concretization threshold ("–bdd-concthreshold") that defines up to how many elements concretizations may take place. The concretization threshold is important, as it determines up to how many addresses may be considered during a write to the heap, and up to how many targets may be considered when resolving jumps.

To analyze the binary "demo.exe" using BDDStab, one could call Jakstab as follows:

```
./jakstab -m demo.exe --cpa fz --bdd-threshold 3 --bdd-concthreshold 1000
```

6.8. Evaluation

Because of the unique challenges connected to sound CFR on binaries, it is unlikely that integer value analysis, even if precise, is enough to produce sound CFGs for larger real-world programs. With this evaluation, we provide an insight into the performance of our domain when applied to real-world programs. Hence, we chose the CPU2006 [78] as benchmark suite, which contains such programs. For the evaluation, we use a configuration of Jakstab that makes the least number of assumptions about the analyzed programs. Specifically, no assumptions about the stack discipline of included functions are made.

6.8.1. The CPU2006 Benchmark Suite

The CPU2006 benchmark suite was designed to measure the performance of compilers as well as the integer and floating point performance of computers and is now widely used for this purpose [79]. The suite contains 17 floating point benchmarks and 13 integer benchmarks from various domains, including fluid dynamics, biochemistry, linear programming, video compression, and path finding algorithms. The benchmark programs are written in C, C++, and Fortran, and perform an expensive calculation depending on input that is included in the benchmark suite.

We chose the CPU benchmark suite because it is well-known [79] and contains adaptations of real-world programs each of which compiles to a single executable. These executables do not load code at runtime, which would complicate the analysis with Jakstab.

However, the benchmark was not designed for benchmarking binary analyzers. More than half of the programs perform mostly floating point operations, which are not supported by our domain. Consequently, these operations are interpreted as nondeterministic in the analysis. Furthermore, the programs perform computations depending on input that the CPU2006 suite provides. It is unclear whether the programs assume correctly formatted input and exhibit undefined runtime behavior if presented with invalid input. If so, then it is possible that the programs contain unbounded control flow and the analysis using a sound analyzer will fail because not all jump targets can be resolved.

6.8.2. Measuring Procedure

The benchmarks are compiled using GCC version 4.8.4, provided by Ubuntu 14.04.3. We use the flag “no-stack-protector” as well was the option “-m32” to suppress stack protection and generate 32-bit code, as Jakstab lacks support for the 64-bit instruction set. Furthermore, we create and measure each binary for the optimization levels 0 to 3, and s. For the measuring procedure, we apply Jakstab with the options “–cpa fz”, “–bdd-threshold 10”, and “–bdd-concthreshold 1000”, i.e., we activate forward substitution and BDDStab, use widening after 10 iterations and concretize sets with up to 1000 elements. This configuration means that Jakstab will use its pessimistic resolve operator, which

means that if a dynamic jump cannot be resolved by BDDStab, then the analysis is stopped. We use the pessimistic resolve operator as it makes no assumptions about the executable and is therefore sound for any executable. If the analysis took more than 10 hours, we terminated it. After termination of Jakstab, we parse its output to generate our measurements. If the analysis was terminated prematurely, and this has lead to specific measurements to not be made, then we denote this in our data using the $\perp$ symbol.

6.8.3. Data Description

In this section, we describe the individual columns of our CPU2006 evaluation data.

Opt The optimization level passed to GCC during for compilation. O0 means no optimization, whereas O3 means maximum optimization. O1 and O2 enable optimizations that should not incur any speed-space tradeoffs, such as loop unrolling, which may increase the executable's size. In difference to O1, O2 includes expensive optimizations such as instruction scheduling, which is a technique to improve pipeline efficiency. O3 enables optimizations that try to improve execution speed but may increase the generated executable such as inlining of functions. Os instructs the compiler to generate small executables, e.g., to run on size constrained systems such as microcontrollers.

Program The name of the analyzed program from the CPU 2006 benchmark suite.

Instrs The number of disassembled assembly instructions. This measurement gives an insight into the size of executables that we can expect Jakstab with BDDStab to analyze soundly.

IL Instrs The number of analyzed intermediate language instructions. During analysis, each disassembled assembly instruction gets lifted to Jakstab's intermediate language. Since each instruction at assembly level may produce several instructions at the level of the intermediate language, the number of intermediate language instructions is generally greater than that on assembly level. This measurement gives an insight into how many IL instructions on average are needed to represent assembly instructions, and how many our analysis actually analyzed.

IBran The number of disassembled indirect branches at assembly level. A branch instruction in X86 is considered indirect if its target is not described by a constant offset to the program counter or by a constant. Furthermore, the target may not lie in the static data region of the binary.

UBran The number of unresolved targets on intermediate language level. A target of a jump instruction in the intermediate language is considered unresolved, if the analysis could not produce a set of target addresses. Within BDDStab, if the variation domain

of the target expression contains more elements than are allowed to be concretized, then the corresponding jump is counted as unresolved. In our evaluation, we configured the concretization threshold to 1000 elements. Together with the number of indirect branches, this measurement gives an insight into how efficient our analysis is at resolving indirect branches.

AvEquivs The average number of equivalence relations per CFG node, as computed by our equivalence class analysis (Section 6.4). Each node in the CFG corresponds to one IL statement. Note that we count equivalence relations between two variables only. Hence, an equivalence class with n elements is counted as $(n * (n - 1))/2$ individual equivalence relations.

Opt	Program	Instrs	IL Instrs	Ibran	UBran	AvEquivs
O0	GemsFDTD	719	2202	21	4	$\perp$
O0	astar	249	746	7	3	1.9
O0	bwaves	214	609	9	2	$\perp$
O0	bzip2	203	584	12	5	2.0
O0	cactusADM	922	2333	8	6	$\perp$
O0	calculix	2294	10335	9	0	$\perp$
O0	gamess	179	615	8	1	$\perp$
O0	gcc	253	901	5	2	8.4
O0	gobmk	276	830	2	1	5.7
O0	gromacs	420	1360	8	9	1.3
O0	h264ref	1121	3994	21	21	4.0
O0	hmmer	741	2009	13	6	2.5
O0	lbm	212	500	6	1	4.8
O0	leslie3d	127	408	8	1	$\perp$
O0	libquantum	290	1079	4	2	$\perp$
O0	mcf	549	1147	12	5	0.7
O0	milc	91	429	3	1	3.3
O0	namd	323	988	7	4	$\perp$
O0	omnetpp	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
O0	povray	431	1343	4	12	3.4
O0	sjeng	314	1062	2	1	$\perp$
O0	soplex	223	784	2	1	7.9
O0	specrand	146	370	4	0	4.8
O0	sphinx_livept	617	2176	10	2	$\perp$
O0	tonto	779	2384	6	4	2.9
O0	wrf	3671	9970	6	1	$\perp$
O0	zeusmp	263	601	8	2	$\perp$

Table 6.1.: CPU2006, Optimization Level 0, Pessimistic Resolver

6.8.4. Results

Tables 6.1, 6.2, 6.3, 6.4, and 6.5 show the results of applying Jakstab with BDDStab on the CPU2006 programs, compiled with varying optimization levels. These tables do not list results for the "dealII", "perlbench", and "xalancbmk" benchmarks, as Jakstab was either not able to parse the corresponding ELF file, or BDDStab was not able to provide any usable result in time. The tables use the $\perp$ symbol, if no result could be obtained for the corresponding measurement. Since the AvEquivs measurement depends on the number of CFG edges on IL level, it requires a CFG to be generated, which does not happen if the analysis terminated abnormally.

Opt	Program	Instrs	IL Instrs	Ibran	UBran	AvEquivs
O1	GemsFDTD	368	1021	20	5	$\perp$
O1	astar	294	981	9	4	1.2
O1	bwaves	849	3523	9	0	$\perp$
O1	bzip2	100	351	4	2	3.7
O1	cactusADM	752	2230	5	7	4.7
O1	calculix	40	484	1	0	3.9
O1	gamess	1971	8116	10	0	$\perp$
O1	gcc	71	595	2	1	7.4
O1	gobmk	217	767	2	1	5.0
O1	gromacs	262	1047	7	4	2.9
O1	h264ref	326	922	14	7	1.7
O1	hmmer	325	1049	5	6	4.9
O1	lbm	166	414	6	1	6.8
O1	leslie3d	783	3655	8	2	$\perp$
O1	libquantum	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
O1	mcf	499	1168	12	8	$\perp$
O1	milc	76	410	3	1	3.4
O1	namd	86	370	5	2	2.9
O1	omnetpp	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
O1	povray	236	1045	3	7	2.6
O1	sjeng	146	573	2	1	3.5
O1	soplex	236	980	3	0	$\perp$
O1	specrand	124	313	4	0	3.2
O1	sphinx_livept	236	833	8	5	$\perp$
O1	tonto	513	1636	5	3	3.5
O1	wrf	1445	3821	4	1	$\perp$
O1	zeusmp	231	545	8	2	2.1

Table 6.2.: CPU2006, Optimization Level 1, Pessimistic Resolver

With BDDStab resolving dynamic jumps, Jakstab was able to disassemble between

40 (“calculix”, O1) and 18070 (“wrf”, O3) instructions. Since the “wrf” executable is 4.3 megabytes in size, it is safe to assume that even in the optimal case, our analysis only covered a small part of the executable before it stopped. Overall, each assembly instruction gets translated to about 3 instructions in Jakstab’s intermediate language.

Opt	Program	Instrs	IL Instrs	Ibran	UBran	AvEquivs
O2	GemsFDTD	2325	9673	22	10	⊥
O2	astar	308	1075	9	3	2.0
O2	bwaves	580	2269	9	1	⊥
O2	bzip2	102	360	4	2	3.4
O2	cactusADM	679	2101	6	1	⊥
O2	calculix	44	497	1	0	3.8
O2	gamess	209	745	7	0	⊥
O2	gcc	66	583	3	1	8.7
O2	gobmk	217	783	2	1	4.9
O2	gromacs	732	2874	12	8	⊥
O2	h264ref	325	962	14	7	⊥
O2	hmmer	298	1038	5	6	4.1
O2	lbm	171	431	6	1	5.6
O2	leslie3d	779	3571	9	1	⊥
O2	libquantum	239	677	6	2	1.5
O2	mcf	483	1123	12	8	⊥
O2	milc	78	422	3	1	3.3
O2	namd	82	364	5	2	2.8
O2	omnetpp	346	1397	16	9	0.9
O2	povray	225	1041	3	7	2.5
O2	sjeng	147	598	2	1	3.3
O2	soplex	170	798	3	0	⊥
O2	specrand	124	335	4	0	3.9
O2	sphinx_livept	585	2185	9	4	⊥
O2	tonto	527	1642	5	3	3.2
O2	wrf	1421	3697	4	1	⊥
O2	zeusmp	242	559	8	2	1.9

Table 6.3.: CPU2006, Optimization Level 2, Pessimistic Resolver

It may be surprising that the “UBran” column can contain greater numbers than the “IBran” column. Unfortunately, the number of successfully resolved indirect branches cannot be computed by subtracting the “UBran” column from the “IBran” column, because the “IBran” column refers to instructions on assembly level, while the “UBran” column refers to instructions on the IL level. Furthermore, “UBran” lists all instances where the analysis was unable to determine a set of successors. Therefore, “UBran” includes cases where the analysis was unable to resolve return statements, which can

happen if the precision of the stack has deteriorated. An example of such a case is the "povray" benchmark in optimization level O1. Therefore, the number of resolved indirect branches can be larger than the difference between the "UBran" and "IBran" columns, suggesting that BDDStab is effective at resolving dynamic jumps, but must be improved in terms of context sensitivity.

Opt	Program	Instrs	IL Instrs	Ibran	UBran	AvEquivs
O3	GemsFDTD	1322	5249	21	7	⊥
O3	astar	287	967	9	3	0.7
O3	bwaves	567	2229	9	0	⊥
O3	bzip2	103	367	4	2	3.4
O3	cactusADM	527	1595	6	1	⊥
O3	calculix	44	497	1	0	3.8
O3	gamess	242	853	7	0	⊥
O3	gcc	66	583	3	1	8.0
O3	gobmk	211	772	2	1	4.5
O3	gromacs	700	2713	12	8	⊥
O3	h264ref	⊥	⊥	⊥	⊥	⊥
O3	hmmer	298	1038	5	6	4.2
O3	lbm	171	431	6	1	5.6
O3	leslie3d	⊥	⊥	⊥	⊥	⊥
O3	libquantum	239	672	6	2	1.6
O3	mcf	⊥	⊥	⊥	⊥	⊥
O3	milc	79	430	3	1	3.3
O3	namd	82	364	5	2	2.8
O3	omnetpp	346	1397	16	9	0.9
O3	povray	212	1019	3	3	3.0
O3	sjeng	147	598	2	1	3.2
O3	soplex	196	845	3	0	⊥
O3	specrand	124	330	4	0	3.6
O3	sphinx_livept	819	3152	10	4	⊥
O3	tonto	507	1586	5	2	3.1
O3	wrf	18070	80923	6	1	⊥
O3	zeusmp	242	577	8	2	1.9

Table 6.4.: CPU2006, Optimization Level 3, Pessimistic Resolver

The "AvEquivs" column shows that there exist up to 8.7 equivalence relations on average per CFG node ("gcc", O2). However, in this benchmark, the analysis only recovered 66 instructions. The fewer instructions are analyzed, the more likely is the number of equivalence relations per node overestimated, because when more parts of the CFG are reconstructed, back edges may lead to additional confluence points. The join of the equivalence class analysis only keeps two variables in the same equivalence

class if they are considered equivalent in all incoming flow data. We therefore believe the 1.5 equivalence relations per CFG node in the "povray" benchmark for optimization level Os to be more realistic. The high number of equivalences for programs with fewer recovered instructions is still potentially contributing to the analysis' precision during analysis.

Opt	Program	Instrs	IL Instrs	Ibran	UBran	AvEquivs
Os	GemsFDTD	108	523	7	1	3.2
Os	astar	369	1122	8	3	2.4
Os	bwaves	⊥	⊥	⊥	⊥	⊥
Os	bzip2	97	379	4	1	3.7
Os	cactusADM	1213	4049	7	1	⊥
Os	calculix	54	514	1	0	4.0
Os	gamess	⊥	⊥	⊥	⊥	⊥
Os	gcc	85	633	2	1	8.4
Os	gobmk	467	1276	1	3	1.6
Os	gromacs	318	1231	7	5	2.1
Os	h264ref	414	1363	15	17	2.0
Os	hmmer	131	690	3	2	5.7
Os	lbm	186	535	6	1	6.8
Os	leslie3d	292	1187	8	0	⊥
Os	libquantum	166	588	4	1	2.3
Os	mcf	⊥	⊥	⊥	⊥	⊥
Os	milc	99	472	3	1	4.2
Os	namd	259	704	8	3	1.0
Os	omnetpp	⊥	⊥	⊥	⊥	⊥
Os	povray	3176	10430	3	9	1.5
Os	sjeng	151	605	2	1	3.7
Os	soplex	107	558	2	1	4.8
Os	specrand	130	383	4	0	4.5
Os	sphinx_livept	857	3296	11	15	⊥
Os	tonto	353	1244	5	3	4.3
Os	wrf	189	965	4	1	2.2
Os	zeusmp	158	490	8	0	⊥

Table 6.5.: CPU2006, Optimization Level s, Pessimistic Resolver

7. Future Work

In this chapter, we discuss future research directions to improve the efficiency and precision of our BDD-based, non-convex abstract domain.

Benchmark Suite

As described in the evaluation of our Jakstab plug-in, one cannot yet hope to perform sound binary analysis of real-world programs such as those in the CPU2006 benchmark suite, which is not surprising as this benchmark suite was not designed for benchmarking binary analyzers. Simply compiling benchmark suites for software analyzers on structured languages is insufficient to produce a useful benchmark suite for binary analyzers, since the challenges in these suites are often to detect undefined runtime behavior such as out of bounds array access. As discussed briefly in Section 2.1, the compiler may produce code with unbounded control flow, i.e., a jump instruction with non-deterministic target, for source code with undefined behavior. Binary analyzers cannot decide whether the source of such a non-deterministic jump is undefined behavior in the source program, or overapproximation during the analysis. A specialized benchmark suite for sound binary analyzers would contain for each instruction of an executable in the suite a trace that proves that this instruction is indeed reachable. This information can then be used as ground truth during the evaluation of an analyzer.

Tree-based Heap Model

Our domain is efficient at working with large VDs. However, in the BDDStab adapter, there are two operations that force concretization, which is not feasible for large VDs:

1. Heap Access
2. Resolve Operator

The resolve operator is used by Jakstab to compute successor abstract locations from an abstract state and an instruction. Heap access forces concretization, because we use the heap model of Jakstab, which is based on a simple map structure from heap addresses to abstract values. Therefore, when reading and writing to a heap address represented by a VD with n elements, n map accesses are performed, which may be unacceptable. An abstract heap based on binary tree structures similar to multi-terminal BDDs (MTBDD) [66], could be used to speed up read and write operations, if the BDD representation of the address VD is efficient. We envision this binary tree structure to have the same interpretation of its edges as our BDD-based integer sets, i.e., the edges correspond to the bits in the heap address. As in our BDD structure, sub-trees would correspond to an interval of heap addresses. All nodes in the heap tree would contain a BDD that represents all abstract values written to exactly this interval of heap addresses.

Therefore, performing a weak update would traverse the address BDD and the heap tree together until a 1 is found in the address BDD. Then, the BDD stored in the corresponding heap tree node is updated to the union of the abstract value that must be stored and previously stored value. Read operations would again traverse the address BDD and the heap tree together, while unioning all BDDs in the heap tree's nodes. If a 1 node is found in the address BDD, then the complete sub-heap-tree must still be traversed to collect also those values written to sub-address ranges. However, it should be possible to cache in each heap tree node the union of all BDDs in the nodes on the path to this node, as well as the union of all BDDs below this node. This caching must be implemented lazily to avoid traversing the whole heap tree in each operation.

Context Sensitivity

The current CPA configuration for our domain is equivalent to classic DFA. Therefore, we currently join the incoming data flow information at each confluence point. In this join, the stack of all abstract states get merged. Because the stack contains the return addresses from function calls, joining the stack destroys context sensitivity. Jakstab contains a CPA that uses call strings [80] to avoid joining different contexts. Yet, using this domain is only sound if the analyzed program uses stack discipline, i.e., does not change the return address explicitly. By changing the CPA merge operator to merge only abstract states with equal stack value, or merging only if there are too many abstract states with the same abstract location, limited context sensitivity could be achieved without assuming stack discipline.

Naive Sets for Small VDs

Our BDD-based domain remains efficient for large integer sets. However, for small sets, it is likely that our transfer functions are slower than those on naive sets. It is therefore possible to speed up our domain using naive sets for VDs with fewer elements than a given threshold and BDD-based integer sets otherwise. To avoid creating naive sets that are larger than the threshold, transfer functions on naive sets must be able to approximate the size of their result. Using a non-relational abstraction, as we do in our domain, the size of the result of most transfer functions can safely be approximated using the multiplication of all operand set sizes. Should this multiplication be above the threshold, then BDD-based sets are used in the computation.

Relational Information in Transfer Functions

By itself, our integer value analysis domain is non-relational, which compromises precision for efficiency. However, we believe that introducing relational information can be used to improve the precision as well as the efficiency of our transfer functions. The computation of this relational information would be left to an extra domain, e.g., the equiv-

alence class domain presented in Section 6.4. The exact abstract version of the binary operator $\circ$ of non-relational value analysis domains computes $\{a \circ b \mid a \in A, b \in B\}$, i.e., it combines every value from the VD A with every value form the VD B. If it is known that the variables whose values these VDs represent must have equal value at runtime, then the result of the transfer function can be simplified to $\{a \circ b \mid a \in A, b \in B, a == b\}$, which is likely to contain fewer elements, corresponding to better precision, but is also potentially easier to compute because fewer combinations must be considered.

Generally, our algorithms for binary transfer functions use four recursive calls in the case where both BDDs are non-terminals. These four recursive calls use all four combinations of the two decision node's sub-BDDs as arguments. With the equality constraint, the number of recursive calls can be reduced to two, namely those on both true successor BDDs and both false successor BDDs, as equal integers have the same bit pattern and the edges encode the bits.

It is also possible to support more complex constraints such as $\pm a \pm b \leq c$, where c is a constant, as computed by the octagon abstract domain [15]. A path of length m to a decision node in the BDD of an unsigned n-bit integer set, described by an m-bit bitvector p, corresponds to a weight of $p_\mathrm{v} = \sum_{i=0}^{m-1} p_{\{i\}} * 2^{i+n-m-1}$ in the integers that include the bitvector as their first m bits. Hence, all these integers are greater or equal than p_v and smaller or equal than $p_\mathrm{v} + 2^{n-m} - 1$. Our algorithms can use these bounds to decide whether the BDDs of a recursive call can fulfill the constraint or not. If the BDDs cannot fulfill the constraint, then the recursive call does not have to be executed.

8. Conclusion

In this dissertation, we presented an abstract domain for integer value analysis. We make the implementation of this abstract domain available in the BDDStab library, and use it to implement an integer value analysis for binaries using the Jakstab binary analysis framework. The domain uses BDD-based integer sets to implement efficient abstract values that neither enforce convexity nor are restricted to a specific type of non-convexity. Hence, this domain does not suffer from any precision loss through joining at confluence points, other than the loss that is caused by its non-relational abstraction. To support BDD-based integer sets, we define a variation of classic BDDs with complement edges that differs in two ways:

1. Labelless Decision Nodes
2. Precomputed Satisfiability Counts

Labelless decision nodes allow the sharing of structurally equivalent BDDs that have different semantics. This extension compromises on the efficiency of BDD reduction, as each decision node's label is given by the length of the path from the BDD's root node to it. Since our BDDs only contain path lengths of up to the bitwidths of the represented integers, which are usually 64, this compromise seems advantageous, especially because improved sharing also improves the memoization of algorithms.

Precomputed satisfiability counts allow us to determine in $O(1)$ the number of different paths from a BDD's root node to a true terminal. In our BDD-based integer sets, satisfiability count corresponds to the set cardinality.

We define algorithms for the most common transfer functions that are defined on the structure of the BDDs instead of the integers they represent. Hence our algorithms are efficient even on very large integer sets. The algorithms for bitwise `and` and `or`, as well as the algorithms for addition and subtraction, are exact in the non-relational case. Other algorithms such as multiplication, shifts, and rotations are only exact when one operand set is a singleton, as exact versions that do not rely on concretization are not known. Efficient conversion from strided intervals and to regular intervals allows us to fall back on known transfer functions when no more precise algorithm is known.

Our BDDStab adapter plug-in for the binary analysis framework Jakstab implements BDD-based value analysis for binaries. The reliance of machine code on dynamically computed jumps, necessitates precise analysis of jump target addresses. Since the distribution of target addresses within an executable is arbitrary, arbitrary sets are needed for a precise representation of the abstract target value.

The treatment of conditional jumps in the analysis of executables is challenging, because the jump condition does not directly depend on registers or memory locations, but instead on formulas over special variables called flags, the value of which in turn depends on registers and memory locations. Since a non-relational abstraction loses the relationship between registers, memory locations, and flags, non-relational analyzers can only compute restrictions for flags at conditional jumps, and not for registers

and memory locations as required. We use a combination of forward substitution and simplification to translate conditions back to registers and memory locations, and then use our overapproximating constraint solver for BDD-based integer sets to create an abstract state that represents all states that fulfill the constraint. To restrict not only those abstract values of registers and memory locations that are directly referenced in the simplified constraint, but also those that must have equal values at runtime, we define and implement a specialized analysis that computes equivalence relations between storage locations.

In difference to most other abstractions used in integer value analyses, such as strided intervals, our BDD-based integer sets require no approximation. However, as the evaluation of our BDDStab library shows, they can still efficiently exploit regularities in the represented sets. The performance evaluation of our BDD-based integer sets for intervals and congruence classes leads us to expect that our sets are efficient at representing set shapes that are similar to the evaluated ones. For example, if a strided interval is joined with a singleton interval that is not in the strided interval's congruence class, then the resulting join will have a stride of 1. Our BDD-based integer sets simply add this singleton without approximation.

The evaluation of the BDDStab library also shows that the performance of the evaluated algorithms does not depend on the size of the operand sets, but rather on the structure of the associated BDDs, which allows our algorithms to remain efficient even for large integer sets. The evaluation using congruence classes, which contain many references to the same sub-BDD in memory, showed that our algorithms avoid recomputing results for previously computed operands.

The evaluation of our BDDStab adapter using CPU2006 benchmarks confirms that it is necessary for a binary analysis framework to resolve indirect branches, and that our integer value analysis is able to do so. Furthermore, the evaluation substantiates our assumption that equivalence relations between storage locations are frequent in the machine code representation of programs. This discovery justifies our implementation of a dedicated equivalence analysis.

The low number of analyzed instructions in the CPU2006 benchmark shows that, without further assumptions about the binary, sound analysis of real-world executables is not yet within the capabilities of integer value analysis, even if it focuses on precision as BDDStab does. Smaller sized programs, such as the example from Section 2.1, can be analyzed soundly using a combination of Jakstab and BDDStab, which suggests that BDDStab will be useful in the analysis of parts of executables, e.g., functions. Hence, we suggest further research in combining BDDStab with unsound analysis methods, such as identifying functions in binaries, or iteratively improving precision only if needed.

The unique challenges of the analysis of binaries cannot be solved using only well-known techniques from the analysis of structured programs. We think that the combination of precise value analyzers with heuristics that guide the analysis is the direction that future research should take.

Bibliography

[1] S. Mattsen, A. Wichmann, and S. Schupp, "BDDStab: BDD-based Value Analysis of Binaries," in *The Fifth Workshop on Tools for Automatic Program Analysis (TAPAS)*, 2014.

[2] S. Mattsen, A. Wichmann, and S. Schupp, "A non-convex abstract domain for the value analysis of binaries," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 271–280, Mar. 2015.

[3] J. Kinder and H. Veith, "Jakstab: A Static Analysis Platform for Binaries," in *Computer Aided Verification* (A. Gupta and S. Malik, eds.), Lecture Notes in Computer Science, pp. 423–427, Springer, July 2008.

[4] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds., *Structured Programming*. London, UK, UK: Academic Press Ltd., 1972.

[5] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 42, pp. 230–265, Jan. 1937.

[6] X. Meng and B. P. Miller, "Binary Code is Not Easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), pp. 24–35, ACM, 2016.

[7] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (Berkeley, CA, USA), pp. 845–860, USENIX Association, 2014.

[8] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of Executable Code Revisited," in *Proceedings of the 9th Working Conference on Reverse Engineering*, WCRE '02, (Washington, DC, USA), pp. 45–54, IEEE Computer Society, 2002.

[9] "Radare: a Portable Reversing Framework." www.radare.org. Last accessed 29 March 2017.

[10] "IDAPro: The Interactive Disassembler." www.hex-rays.com/products/ida. Last accessed 29 March 2017.

[11] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, (New York, NY, USA), pp. 290–299, ACM, 2003.

[12] A. Sepp, B. Mihaila, and A. Simon, "Precise Static Analysis of Binaries by Extracting Relational Information," in *Proceedings of the 18th Working Conference*

on Reverse Engineering, WCRE '11, (Washington, DC, USA), pp. 357–366, IEEE Computer Society, 2011.

[13] H. G. Rice, "Classes of Recursively Enumerable Sets and Their Decision Problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[14] T. Ball and S. K. Rajamani, "The SLAM Toolkit," in *Computer Aided Verification*, Lecture Notes in Computer Science, (London, UK), pp. 260–264, Springer, 2001.

[15] A. Miné, "The Octagon Abstract Domain," *Higher Order Symbolic Computation*, vol. 19, pp. 31–100, Mar. 2006.

[16] J. Kinder and H. Veith, "Precise Static Analysis of Untrusted Driver Binaries," in *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, (Austin, TX), pp. 43–50, FMCAD Inc, 2010.

[17] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A Static Analyzer for Large Safety-critical Software," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, (New York, NY, USA), pp. 196–207, ACM, 2003.

[18] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti, "Experience Report: OCaml for an Industrial-strength Static Analysis Framework," in *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, (New York, NY, USA), pp. 281–286, ACM, 2009.

[19] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the Second International Symposium on Programming*, pp. 106–130, Dunod, Paris, France, 1976.

[20] P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints Among Variables of a Program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, (New York, NY, USA), pp. 84–96, ACM, 1978.

[21] L. Chen, A. Miné, and P. Cousot, "A Sound Floating-Point Polyhedra Abstract Domain," in *Programming Languages and Systems* (G. Ramalingam, ed.), Lecture Notes in Computer Science, pp. 3–18, Springer, Dec. 2008.

[22] K. Ghorbal, E. Goubault, and S. Putot, "The Zonotope Abstract Domain Taylor1+," in *Computer Aided Verification* (A. Bouajjani and O. Maler, eds.), Lecture Notes in Computer Science, pp. 627–633, Springer, June 2009.

[23] G. Balakrishnan and T. Reps, "WYSINWYX: What You See is Not What You eXecute," *ACM Transactions on Programming Languages and Systems*, vol. 32, pp. 23:1–23:84, Aug. 2010.

[24] J. Brauer, A. King, and S. Kowalewski, "Abstract Interpretation of Microcontroller Code: Intervals Meet Congruences," *Science of Computer Programming*, vol. 78, pp. 862–883, July 2013.

[25] P. Granger, "Static Analysis of Linear Congruence Equalities Among Variables of a Program," in *Proceedings of the International Joint Conference on Theory and Practice of Software Development on Colloquium on Trees in Algebra and Programming (CAAP '91): Vol 1*, TAPSOFT '91, (New York, NY, USA), pp. 169–192, Springer, 1991.

[26] E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic," in *Logic of Programs, Workshop*, (London, UK, UK), pp. 52–71, Springer, 1982.

[27] K. L. McMillan, *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993.

[28] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis," in *Computer Aided Verification* (W. Damm and H. Hermanns, eds.), no. 4590 in Lecture Notes in Computer Science, pp. 504–518, Springer, July 2007.

[29] S. Mattsen, P. Cuoq, and S. Schupp, "Driving a sound static software analyzer with branch-and-bound," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 63–68, Sept. 2013.

[30] K. Ghorbal, F. Ivančić, G. Balakrishnan, N. Maeda, and A. Gupta, "Donut Domains: Efficient Non-convex Domains for Abstract Interpretation," in *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, (Berlin, Heidelberg), pp. 235–250, Springer, 2012.

[31] L. Chen, J. Liu, A. Miné, D. Kapur, and J. Wang, "An Abstract Domain to Infer Octagonal Constraints with Absolute Value," in *Static Analysis Symposium* (M. Müller-Olm and H. Seidl, eds.), Lecture Notes in Computer Science, pp. 101–117, Springer, Sept. 2014.

[32] G. A. Kildall, "A Unified Approach to Global Program Optimization," in *Proceedings of the 1th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '73, (New York, NY, USA), pp. 194–206, ACM, 1973.

[33] C. Lattner, *LLVM: An Infrastructure for Multi-Stage Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec. 2002.

[34] U. P. Khedker and D. M. Dhamdhere, "Bidirectional Data Flow Analysis: Myths and Reality," *SIGPLAN Not.*, vol. 34, pp. 47–57, June 1999.

[35] M. Madsen and A. Møller, "Sparse Dataflow Analysis with Pointers and Reachability," in *Static Analysis* (M. Müller-Olm and H. Seidl, eds.), Lecture Notes in Computer Science, pp. 201–218, Springer, Sept. 2014.

[36] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, (New York, NY, USA), pp. 238–252, ACM, 1977.

[37] M. Sintzoff, "Calculating Properties of Programs by Valuations on Specific Models," in *Proceedings of ACM Conference on Proving Assertions About Programs*, (New York, NY, USA), pp. 203–207, ACM, 1972.

[38] N. D. Jones and S. S. Muchnick, "A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, (New York, NY, USA), pp. 66–74, ACM, 1982.

[39] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of Pointers and Structures," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, (New York, NY, USA), pp. 296–310, ACM, 1990.

[40] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric Shape Analysis via 3-valued Logic," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, (New York, NY, USA), pp. 105–118, ACM, 1999.

[41] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "CodeSurfer/x86—A Platform for Analyzing x86 Executables," in *Compiler Construction*, pp. 250–254, Springer, Apr. 2005.

[42] C. Cifuentes and S. Sendall, "Specifying the semantics of machine instructions," in *6th International Workshop on Program Comprehension, 1998. IWPC '98. Proceedings*, pp. 126–133, 1998.

[43] A. Scholl, "A signedness-agnostic interval domain with congruences and an implementation for jakstab," Bachelor thesis, Hamburg University of Technology, May 2016.

[44] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *Computer Aided Verification*, Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 463–469, Springer, 2011.

[45] E. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 Decompilation using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring," *Proceedings of the USENIX Security Symposium*, pp. 353–368, Aug. 2013.

[46] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, (Berlin, Heidelberg), pp. 1–25, Springer, 2008.

[47] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 41–41, USENIX Association, 2005.

[48] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The BINCOA Framework for Binary Code Analysis," in *Computer Aided Verification* (G. Gopalakrishnan and S. Qadeer, eds.), Lecture Notes in Computer Science, pp. 165–170, Springer, July 2011.

[49] S. Bardin and P. Herrmann, "OSMOSE: Automatic Structural Testing of Executables," *Software Testing, Verification and Reliability*, vol. 21, pp. 29–54, Mar. 2011.

[50] E. Fleury, O. Ly, G. Point, and A. Vincent, "Insight: An Open Binary Analysis Framework," in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pp. 218–224, Springer, 2015.

[51] S. Bardin, P. Herrmann, and F. Védrine, "Refinement-based CFG Reconstruction from Unstructured Programs," in *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, (Berlin, Heidelberg), pp. 54–69, Springer, 2011.

[52] C. Y. Lee, "Representation of Switching Circuits by Binary-Decision Programs," *Bell System Technical Journal*, vol. 38, pp. 985–999, July 1959.

[53] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. 27, pp. 509–516, June 1978.

[54] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 677–691, Aug. 1986.

[55] W. J. Masek, *A Fast Algorithm for the String Editing Problem and Decision Graph Complexity.* Master thesis, Massachusetts Institute of Technology, 1976.

[56] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, (New York, NY, USA), pp. 40–45, ACM, 1990.

[57] M. Fujita, Y. Matsunaga, and T. Kakuda, "On variable ordering of binary decision diagrams for the application of multi-level logic synthesis," in *Proceedings of the European Conference on Design Automation.*, EURO-DAC '91, pp. 50–54, Feb. 1991.

[58] H. Fujii, G. Ootomo, and C. Hori, "Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, ICCAD '93, (Los Alamitos, CA, USA), pp. 38–41, IEEE Computer Society Press, Nov. 1993.

[59] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, ICCAD '93, (Los Alamitos, CA, USA), pp. 42–47, IEEE Computer Society Press, Nov. 1993.

[60] B. Bollig and I. Wegener, "Improving the Variable Ordering of OBDDs Is NP-Complete," *IEEE Transactions on Computers*, vol. 45, pp. 993–1002, Sept. 1996.

[61] J. C. Madre and J. P. Billon, "Proving circuit correctness using formal comparison between expected and extracted behaviour," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, DAC '88, pp. 205–210, June 1988.

[62] U. Kebschull, E. Schubert, and W. Rosenstiel, "Multilevel logic synthesis based on functional decision diagrams," in *Proceedings of the European Conference on Design Automation*, EURO-DAC '92, pp. 43–47, Mar. 1992.

[63] R. Drechsler, B. Becker, and M. Theobald, "Fast OFDD Based Minimization of Fixed Polarity Reed-Muller Expressions," in *Proceedings of the European Conference on Design Automation*, EURO-DAC '94, (Los Alamitos, CA, USA), pp. 2–7, IEEE Computer Society Press, 1994.

[64] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski, "Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams," in *Proceedings of the 31st ACM/IEEE Design Automation Conference*, DAC '94, (New York, NY, USA), pp. 415–419, ACM, 1994.

[65] S. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," in *Proceedings of the 30st ACM/IEEE Design Automation Conference*, DAC '93, pp. 272–277, June 1993.

[66] M. Fujita, P. C. McGeer, and J. C.-Y. Yang, "Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation," *Formal Methods in System Design*, vol. 10, pp. 149–169, Apr. 1997.

[67] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic Decision Diagrams and Their Applications," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, ICCAD '93, (Los Alamitos, CA, USA), pp. 188–191, IEEE Computer Society Press, Nov. 1993.

[68] Y. T. Lai and S. Sastry, "Edge-valued binary decision for multi-level hierarchical verification," in *Proceedings of the 29th ACM/IEEE Design Automation Conference*, DAC '92, pp. 608–613, June 1992.

[69] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard, "Fully symbolic model checking of timed systems using difference decision diagrams," in *Workshop on Symbolic Model Checking*, vol. 23, (The IT University of Copenhagen, Denmark), pp. 88–107, June 1999.

[70] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse, "Data-structures for the verification of timed automata," in *Hybrid and Real-Time Systems* (O. Maler, ed.), Lecture Notes in Computer Science, pp. 346–360, Springer, Mar. 1997.

[71] A. Bouraffa, "Fast SAT-count for labelless complementable bdds in bddstab," Bachelor thesis, Hamburg University of Technology, July 2014.

[72] M. Clagett, *Ancient Egyptian Science: Ancient Egyptian mathematics*. American Philosophical Society, 1989.

[73] J. Müller, "Multiplication of BDD-based integer sets for abstract interpretation of executables," Bachelor thesis, Hamburg University of Technology, Mar. 2017.

[74] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Compiler Construction*, CC '02, (London, UK, UK), pp. 213–228, Springer, 2002.

[75] T. Reps, G. Balakrishnan, and J. Lim, "Intermediate-representation Recovery from Low-level Code," in *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, (New York, NY, USA), pp. 100–111, ACM, 2006.

[76] S. Mattsen and S. Schupp, "The challenge of indirection: Treating Flags during Sound Analysis of Machine Code," in *18th Workshop Software-Reengineering and Evolution*, 2016.

[77] F. Lublow, "Change- and precision-sensitive widening for BDD-based integer sets," Bachelor thesis, Hamburg University of Technology, Oct. 2016.

[78] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[79] "SPEC CPU2006 Results." www.spec.org/cpu2006/results. Last accessed 29 March 2017.

[80] M. Sharir and A. Pnueli, *Two Approaches to Interprocedural Data Flow Analysis*. New York: Courant Institute of Mathematical Sciences, New York University, 1978.

A. Auxiliary Algorithms

In the following, we present algorithms to select the minimum and maximum value from a BDD-based integer set. The algorithms make use of the MSB to LSB ordering, meaning that the unsigned variants (`maxu`, `minu`) can make use of the fact that all represented values on the true successor side of the root node of a BDD-based integer set are larger than those on the false successor side. All algorithms assume that the given set is not empty. The signed variants perform a case distinction on the root node (MSB), and afterwards reuse the unsigned variants.

```
   Algorithm min(A_{n}) is
1  |  switch A_{n} do
2  |  |  case 1 do return −2^{n−1}
3  |  |  case (S ⊙ U) do
4  |  |  |  return ¬minu(S) + 1
   |  |  end
   |  end
   end
```

Algorithm 25: Min

```
   Algorithm max(A_{n}) is
1  |  switch A_{n} do
2  |  |  case 1 do return 2^{n−1} − 1
3  |  |  case (S ⊙ U) do
4  |  |  |  return maxu(U)
   |  |  end
   |  end
   end
```

Algorithm 26: Max

```
   Algorithm maxu(A_{n}, h = n) is
1  |  switch A_{n} do
2  |  |  case 1 do return 2^h − 1
3  |  |  case (0 ⊙ U) do return maxu(U, h = h − 1)
4  |  |  case (S ⊙ U) do return maxu(S, h = h − 1) + 2^{h−1}
   |  end
   end
```

Algorithm 27: Unsigned Max

```
   Algorithm minu(A_{n} ,  h = n) is
1  |  switch A_{n}  do
2  |  |  case 1 do return 0
3  |  |  case (S ⊡ 0) do return minu(S ,  h = h − 1) + 2^{h−1}
4  |  |  case (S ⊡ U) do return minu(U ,  h = h − 1)
   |  end
   end
```

Algorithm 28: Unsigned Min

B. BDDStab Library Example Project

In this chapter, we will discuss a self-contained example project in Scala that uses our BDDStab library for computations with large integer sets. We assume that the `git` version control system and the scala build tool `sbt` are installed on the user's machine. Figure B.1 shows the project skeleton, including the directory structure and the contents of the `assembly.sbt` and `build.sbt` files. The `assembly.sbt` file loads a plug-in that generates self-contained Java jar files, and the `build.sbt` file describes our example project. We name this project "bddusage", and use Scala version 2.12.2. The last two lines in the `build.sbt` file introduce a dependency on our BDDStab library, instructing it to be built if the example project is built.

```
/
├── build.sbt
├── Main.scala
└── project
    └── assembly.sbt
```

(a) Directory Structure of Example Program

```
addSbtPlugin("com.eed3si9n"
% "sbt-assembly" % "0.14.4")
```

(b) `assembly.sbt`

```
name := "bddusage"
scalaVersion := "2.12.2"
lazy val root = Project("root", file(".")).dependsOn(bddproject)
lazy val bddproject =
      RootProject(uri("https://github.com/bigmac2k/integerset-bdd.git"))
```

(c) `build.sbt`

Figure B.1.: Example Project Skeleton

Figure B.2 contains the Scala source code of our example program, which computes all 64-bit integers that are even (`evens`), and those that are odd (`odds`). It then uses the constraint solver to compute all positive even and odd numbers (`evensPos` and `oddsPos` respectively), and computes their intersection and union. Additionally, it computes the addition of all 64-bit odd integers to all 64-bit even integers, and compares the resulting set to `odds`.

```scala
import cc.sven.tlike._
import cc.sven.constraint._
import scala.collection.immutable.HashMap
object Main extends App {
//Empty 64-bit integer set
val empty = IntLikeSet.apply[Long, Long](64)
//All 64-bit integers that are not in empty = all 64-bit integers
val top = !empty
//Singleton set containing 0
val zero = IntLikeSet.range[Long, Long](64, 0, 0)
//Singleton set containing 1
val one = IntLikeSet.range[Long, Long](64, 1, 1)
//Bitwise boolean negation of all bits in one
val negtwo = one.bNot
//Bitweise AND of all 64-bit integers with negtwo
//Should give all even integers since LSB is bit masked out
val evens = top.bAnd(negtwo)
//Add one to all even integers without wrap-around, store in odds
val (odds, _) = evens.plusWithCarry(one)
//All positive even integers
val evensPos = {
  //Mapping between variables, named using integers, and their contents
  val m = HashMap(0 -> evens, 1 -> zero)
  //Greater or equal (GTE) of two variables
  val t = GTE(0, 1).solveIntLike[Long, Long](m)
  //Select the constrained value for variable 0
  t(0)
}
//All positive odd integers
val oddsPos = {
  val m = HashMap(0 -> odds, 1 -> zero)
  val t = GTE(0, 1).solveIntLike[Long, Long](m)
  t(0)
}
//Add all odd integers to all even integers, store in odds_
val (odds_, _) = evens.plusWithCarry(odds)
//Output results
println(s"all positive evens:\n${evensPos}\n")
println(s"all positive odds:\n${oddsPos}\n")
println(s"intersection:\n${evensPos intersect oddsPos}\n")
println(s"union:\n${evensPos union oddsPos}\n")
println(s"evens + odds == odds:\n${odds_ == odds}\n")
}
```

Figure B.2.: Example Scala Program (`Main.scala`)

The example program can be compiled to a Java jar file and executed using the commands in Figure B.3. Since the resulting sets are often too large to be displayed, their cardinality is shown in brackets after "`Set`". When the sets do not contain all 64-bit integers and are non-empty, then the `(lowerBound, MANYVAL, upperBound)` pattern is used as an approximative set visualization. Hence, the set of all positive 64-bit integers contains $4611686018427387904 = 2^{62}$ elements between 0 and 9223372036854775806. This result is as expected since there are 2^{63} positive integers and half of them are even. The set of all 64-bit positive integers that are odd is similar, with boundaries increased by 1. As expected, the intersection of the two sets is empty, and their union contains $9223372036854775808 = 2^{63}$ integers. Lastly, the program determines that the addition of all even 64-bit integers to all odd 64-bit integers yields the set of all odd 64 integers. Again, this result is as expected, since 1 is in the set of all odd integers, all even integers plus 1 yields all odd integers, and the addition of an even and an odd integer must be odd. Executing this program on an early 2015 MacBook Pro with a 3.1 Ghz i7 processor and 16 GB of DDR3 ram took less than a second, including the Java runtime start-up.

```
$ sbt assembly
...
$ time java -jar target/scala-2.12/bddusage-assembly-0.1-SNAPSHOT.jar
...
all positive evens:
Set[4611686018427387904](0, MANYVAL, 9223372036854775806)

all positive odds:
Set[4611686018427387904](1, MANYVAL, 9223372036854775807)

intersection:
Set[0]()

union:
Set[9223372036854775808](0, MANYVAL, 9223372036854775807)

evens + odds == odds:
true

java -jar   1,01s user 0,04s system 165% cpu 0,632 total
```

Figure B.3.: Building and Executing the Example Project