Master Thesis

Taxonomy and applications of alias analysis

Author(s):
Lenherr, Thomas

Publication Date:
2008

Permanent Link:
https://doi.org/10.3929/ethz-a-005697567

Rights / License:
In Copyright - Non-Commercial Use Permitted
Taxonomy and Applications of Alias Analysis

Master’s Thesis

by Thomas Lenherr
Department of Computer Science
ETH Zürich

Adviser: Dr. Thomas Wahl
Institute: Computer Systems
Group: Formal Verification

September 25, 2008
Abstract

Alias Analysis is a standard component of today’s optimizing compilers and automated program analysis tools. Due to the great variety of requirements regarding precision and speed, many different algorithms have been developed for this problem. This places a considerable burden on the engineer to decide which one is most appropriate in the application at hand. The main objective of this Master’s thesis is to assist this effort by presenting the design and implementation of a language-independent, and extensible alias analysis library, called AAL. The second objective is to use this library effectively to improve the precision of the Calysto static analyzer.
Contents

1 Introduction 3

2 Background 5
  2.1 Flow- and path-sensitivity 8
  2.2 Interprocedural analysis and context-sensitivity 9
  2.3 Type-based alias analysis 12
  2.4 Representations 14
  2.5 Selection of well-known algorithms 16

3 Alias Analysis Library 19
  3.1 Motivation and Design Decisions 19
  3.2 Intermediate representation: AALSSA 20
  3.3 Implementation overview 25
    3.3.1 Part 1 - Intermediate representation 25
    3.3.2 Part 2 - Analyzer framework 26
  3.4 Alias analysis in practice 27
    3.4.1 Specifics 27
    3.4.2 Problems 30
  3.5 Experimental results 31
    3.5.1 Comparison to LLVM’s Andersen’s analysis 32
    3.5.2 Precision of AAL 33
    3.5.3 Requirements 33
    3.5.4 Origin of $\top$-references 35
    3.5.5 Influential input characteristics 35
  3.6 Conclusion 39

4 Application 42
  4.1 Bit Vector Logic 42
    4.1.1 AAProver 43
  4.2 Calysto 44
  4.3 AACalysto 45
    4.3.1 Reduction to bit vector logic 45
  4.4 Comparison 46
  4.5 Conclusion 48

5 Conclusion 49
  5.1 Summary 49
  5.2 Results 49
  5.3 Future work 50

References 51
1 Introduction

Most imperative languages feature pointers or references, allowing a piece of code to modify a memory location that it does not directly refer to. For optimizing passes within a compiler, such as register allocation or loop-invariant code motion, this means that a conservative approach must be chosen, unless this indirect modification can be captured and taken into account. Similarly, when trying to prove certain conditions over some program, using different kinds of model checking techniques, such indirect modifications cannot be ignored. Alias Analysis names a set of methods that approximate what memory location a pointer might point to and thereby allows these passes and verifiers to make more informed decisions. For example, a compiler without alias analysis must assume that any two pointers might point to the same memory location, and that memory accesses performed using these pointers might interact with each other. If, however, alias analysis is applied, the result might contain information about whether two given pointers always or never point to the same memory location, thereby allowing the compiler to optimize the code accordingly.

As the accuracy of many different methods and optimizations may be increased using alias analysis, many different solutions have been developed, ranging from fast but imprecise to slow and precise. Optimizing compilers are typically more interested in fast and comparably imprecise analyses, while model checkers often value precision over time. Due to the importance of this type of analysis there has been done much research in the area, giving a user many choices of how to design and implement his analysis. The major objective of this thesis has been the design and development of a language-independent and easily extensible library to perform this task. AAL (alias analysis library), as it will be developed and presented, may not only be used to analyze most imperative languages, but will also do this with a competitive performance while requiring only a basic level of understanding from the user. As for developers of new analyses, AAL will allow them to concentrate on the actual analysis by keeping the API minimal. Furthermore, the introduction of function instances will completely disengage the problem of context-sensitivity from the actual analysis, which may be implemented completely context-insensitive and then combined (at runtime) with different kinds of context-sensitivities. Besides the development of AAL, I will also present a simple approach to determine the important input characteristics using a simple, least-square-based approach. This will be used to substantiate some conclusions drawn from the performance results of AAL.

Finally, a possible application of the developed library will be shown by implementing a tool for static checking and bug-hunting, similar to Calysto[1], but eliminating some of its unsound assumptions concerning aliases. The result, AA-Calysto, may then be used for checking of arbitrary assertions at a bit-precise, path-sensitive, and context-sensitive level, similar to Calysto.
Structure of this thesis  Section 2 will introduce the reader to the basic concepts of alias analysis and explain the mechanisms and logic behind it. In section 3 I will present AAL, the general library for alias analysis I developed, combined with a frontend for analyzing arbitrary ANSI C code. In section 4 I will use this library to refine a reimplementation of Calysto[1]. Furthermore, subsection 4.1 provides a short overview on bit vector logic and the used implementation, AAProver. Finally, Section 5 will wrap up the overall results.
2 Background

Assume a fragment of C code as shown in Figure 1. If the compiler can determine at compile time that \( x \) and \( y \) never point to the same memory location it may safely replace the argument \( *x \) on line 9 with the constant 42, since the assignment on line 7 cannot affect its value, otherwise the compiler would have had to load this value \( *x \) from memory again.

On the other hand, if the compiler is able to assess that \( x \) and \( y \) always point to the same location, it may again do a few optimizations, which could result in actual code as shown in Figure 2. This optimization eliminated again a load from memory and reduced the amount of stores from 1-2 (depending on execution) to exactly 1. Note that the second call to \texttt{initialize()} cannot be removed unless further information about its side-effects are known.

Similar problems occur if one tries to prove certain facts for some program, such as a base pointer being unequal to null or any kind of assertions involving pointers as the two examples in Figure 3 demonstrate.

\textit{Alias analysis} is the problem of determining for two different access paths whether they (may) refer to the same memory location. From the examples given above, it is obvious to see that the conservative assumption for a compiler or prover not (or only partially) performing alias analysis is that any two pointers \textit{might} point to the same memory location.

\textbf{Terminology.} When speaking about different kinds of alias analyses one usually differs between \textit{MAY} and \textit{MUST} analyses, where a \textit{MAY} analysis is able to tell whether two pointers \textit{might} point to the same memory objects in some of all possible executions, while a \textit{MUST} analysis determines whether two pointers \textit{must} point to the same memory object in all possible executions. A solution of the alias analysis problem is considered \textit{sound} if all memory locations that \textit{MAY} alias each other are determined as such. A solution is considered \textit{imprecise} if the

\begin{verbatim}
1 int *x = initialize();
2 int *y = initialize();
3 *x = 42;
4 if (someCondition)
5   *y = 0;
6 printf("%d\n", *x);
\end{verbatim}

Figure 1: Basic piece of C code
2 BACKGROUND

```c
int *x = initialize();
initialize();
int tmp = 42;
if (someCondition)
tmp = 0;
*x = tmp;
printf("%d\n", tmp);
```

Figure 2: Possible optimization of code from Figure 1 using alias analysis. More optimizations would be possible.

```c
struct S *s = initialize();
s->someField = 42;
```

Figure 3: Examples of assertions needing alias analysis to be provable

solution believes more memory locations to alias with an other memory location than it might actually alias with at runtime.

**Example.** This part gives an example on how points-to sets may be calculated for some specific piece of code. Points-to sets, further explained in section 2.4 are sets containing all possible targets a pointer might point to at runtime. While more elaborate approaches use worklists or extract a system of constraints, I’ll stick to a simple fixpoint iteration for the sake of simplicity. The left side of Figure 4 shows the input program we process in this example and the right side shows all the steps taken. Initially, the algorithm assumes that none of the pointers point anywhere, which translates to empty points-to sets (step 1). Then all instructions are processed, theoretically in arbitrary order. I chose the order as given in the listing.

Whenever the address of a variable is assigned to a pointer (e.g. `ptr = &var;`) this variable becomes an element of the pointer’s (e.g. `ptr`) points-to set (→ step 1, step 2). Upon an assignment (e.g. `ptr1 = ptr2;`) all elements of the points-to set of the right-hand-side become a member of the points-to set of the left-hand-side, more formally, the points-to set of the right-hand-side becomes a subset of
\begin{verbatim}
1 int a, b;
2 int *p1, *p2, *p3;
3 int **dp;
4 p1 = &a;
5 dp = &p1;
6 while (...) {
7    *dp = &b;
8 }
9 *dp = &b;
10 p3 = *dp;
\end{verbatim}

- 15.: 3rd iteration to detect fixpoint

Figure 4: Input and steps for fixpoint iteration

the points-to set of the left-hand-side (e.g. \texttt{pointsTo(ptr1) \supseteq pointsTo(ptr2)}) (\rightarrow \text{step 3}). Finally stores (e.g. \texttt{*ptr = var}) and loads (e.g. \texttt{var = *ptr}) are dealt with as if they were assignment where the dereferencing expression is replaced by all members of the base pointers points-to set (\rightarrow \text{step 4}, step 5).

The formal rule for a store of the form \texttt{*ptr = var} is \(\forall dst \in \text{pointsTo(ptr)} : \text{pointsTo}(dst) \supseteq \text{pointsTo}(var)\), and similarly for a load of the form \texttt{var = *ptr} it is \(\forall src \in \text{pointsTo(ptr)} : \text{pointsTo}(var) \supseteq \text{pointsTo}(src)\).

Once all instructions have been processed they all get reprocessed since there were changes of the points-to sets during the first iteration. During this iteration the only change happens in step 8 when the assignment \(p2 = p1\); on line 9 gets reprocessed. After the second iteration a third iteration is performed to detect that a fixpoint has been reached effectively terminating the algorithm, returning the calculated points-to sets as result.

Undecidability. The problem of alias analysis, MAY or MUST, in general cannot be decided but merely approximated. This can be proven by reducing the halting problem to alias analysis (see Figure 5): Assume \texttt{main} is the entry point of the program for which the halting problem should be decided. We setup two pointers so they do not alias, then we call \texttt{main}, and right after it returns we set one of the pointers (\texttt{ptrA} in this case) equals the other (\texttt{ptrB}). If the general MAY alias analysis returns that \texttt{ptrA} and \texttt{ptrB} never alias we know that \texttt{main} never returns. On the other hand, if the result is that they may alias we
2.1 Flow- and path-sensitivity

The algorithms I described so far ignored any information about control flow and processed the instructions in arbitrary order, they were flow-insensitive. When looking at Figure 5, the usual flow-insensitive algorithm would need three steps to determine the final points-to set of \( p \) being \( \{t1, t2, t3\} \), which is imprecise during any point of the execution. A flow-sensitive algorithm on the other hand (such as [4]) approximates different points-to sets for \( p \) at all program points. Let’s take another look at the same code as before in Figure 6, this time the different points-to sets as a flow-sensitive algorithm would calculate them are inserted as comments. It is obvious that these kinds of algorithms deliver higher precision, however they do that at cost of memory and time.

Note that depending on the exact situation the removal of \( t1 \) at Point 2 and 3 cannot always be done without violating soundness. In general MUST-alias information must be available to calculate correct kill information.

\[ ^{1} \text{This proof is easily adjusted for a MUST analysis.} \]
2.2 Interprocedural analysis and context-sensitivity

So far we’ve looked only at intraprocedural examples, but when one tries to adapt the presented methods for an interprocedural analysis new problems arise. How should we pass information along when a method is called? How do we deal with globals? What if one of the parameters points to a stack variable in the caller?

Typically the passing of this kind of information is handled similar to assignments. Imagine a program without any recursive functions. We could easily transform all formal arguments of a function into globals and use them to supply the function with our actual arguments. The same could be done for the return value, just in the other direction. Once we’ve done this transformation there is no need to handle a call in a special way. While a flow-insensitive algorithm could just process all instructions from the whole program in arbitrary order, a

Path-sensitivity describes the next step after flow-sensitivity. While a flow-sensitive alias analysis algorithm takes into account where an instruction resides in a control flow graph, it does not take into account under what conditions it is actually reached. There are different levels of path-sensitivity, commonly referred to as limited or full path-sensitivity. Figure 8 shows the code of two functions, where the assertion in test1() may typically be proven using limited path-sensitivity while the assertion in test2() might require full path-sensitivity.

In both examples the key is to observe under which conditions the given assertion may be reached. In test1() a simple removal of NULL from the points-to set of ptr during the branch does the trick. In test2() the prover must be able to determine that the assertion may only be reached if the store on line 7 has been reached as well.

```c
int t1, t2, t3;

int *p = &t1;
// Point 1: p → {t1}

if (...) {
  p = &t2;
  // Point 2: p → {t2}
}
else {
  p = &t3;
  // Point 3: p → {t3}
}
// Point 4: p → {t2, t3}
```

Figure 7: Example
2.2 Interprocedural analysis and context-sensitivity

```c
void test1() {
    int *ptr = initialize();
    if (ptr != NULL) {
        assert(ptr);
        *ptr = 42;
    }
}

void test2() {
    int target;
    int *ptr = NULL;
    bool someCondition = . . ;
    if (someCondition) {
        ptr = &target;
    }
    if (someCondition) {
        assert(ptr);
        *ptr = 42;
    }
}
```

Figure 8: Example code for path-sensitivity

A flow-sensitive algorithm could process the called function as soon as it reaches its call (function pointers will be discussed in section 3.4.1).

So, when looking at arbitrary input programs, possibly containing recursive or mutually recursive functions, one just needs to merge all points-to information available at the call site of the function, over all call sites for the same function. This includes any information that might possibly be reached from the function, through arguments, globals or any indirect way rooting in one of those. This is demonstrated in Figure 9.

Unfortunately, this merging of points-to sets reaching and leaving a function introduces a new problem due to sparse points-to information, as can be seen in Figure 10. Because of this simple way of merging points-to information from different call sites, the algorithm must assume that `returnGivenPointer(&x)` might return `NULL`. This is obviously wrong and giving a prover trying to prove the assertion on line 6 a hard time. Even applying a flow-sensitive algorithm would not solve this problem as flow-sensitivity only concerns intraprocedural handling.

This problem concerns context-sensitivity, which is similar to flow-sensitivity, but on an interprocedural level. When functions and calls are handled as described above and demonstrated in Figure 9, one speaks of a context-insensitive interprocedural analysis because the contexts from where the calls occur are ignored. Opposed to that stand many different kinds of context-sensitive ways to handle this situation. The general idea is to differentiate between calls of the same function from different call sites, as demonstrated in Figure 11. Depending

---

2 Locals, arguments, and returns use a special yet obvious naming scheme.

3 Which may, by the way, be handled as a global variable statically initialized to point to a special null-object.
background

2.2 Interprocedural analysis and context-sensitivity

```c
void test() {
  int x, y;

  store42(&x);
  store42(returnGivenPointer(&y));
  recursive(&x);
}

void store42(int *ptr) {
  *ptr = 42;
}

int *returnGivenPointer(int *ptr) {
  return ptr;
}

void recursive(int *ptr) {
  int local = 0;

  if (*ptr == 42) {
    recursive(&local);
  }
}
```

Flow-insensitive results:

```
returnGivenPointer :: ptr = {test :: y}
returnGivenPointer : returns = {test :: y}
store42 :: ptr = {test :: x, test :: y}
recursive :: ptr = {test :: x, recursive :: y}
```

Figure 9: Example for interprocedural analysis
2.3 Type-based alias analysis

Type-based alias analysis is a collection of possible optimizations when the analyzed language is type safe (e.g., Java). First and foremost, any two instances of unrelated type (neither type is a supertype of the other type) cannot alias. This follows immediately from the fact that two such instances cannot share the same memory area in a type safe environment. And second, a member field may only alias with itself (as part of the same type or any subtype thereof) since these

```c
void test() {
    int x;
    int *ptr;
    ptr = returnGivenPointer(&x);
    assert(ptr);
    *ptr = 42;
    returnGivenPointer(NULL);
}

int *returnGivenPointer(int *ptr) {
    return ptr;
}
```

Flow-insensitive results:

```
returnGivenPointer :: ptr = {test :: x, NULL}
returnGivenPointer : returns = {test :: x, NULL}
test :: ptr = {test :: x, NULL}
```

Figure 10: Problems due to merging of points-to sets

on the level of precision requested this may also concern indirect calls, as shown in Figure 12 where `g()` is called twice from the same call site but within different contexts.

Flow-sensitivity and context-sensitivity are two orthogonal properties of an alias analysis algorithm. This is can be shown using the code in Figure 13. When trying to prove statically that both pointer dereferences never have `NULL` as base, the algorithm used must be both flow- and context-sensitive. While the first store can only be proven safe using a flow-sensitive analysis the second can only be proven using a context-sensitive analysis. This observation will influence design decisions presented in section 3.1.
2 BACKGROUND

2.3 Type-based alias analysis

```c
void test() {
    int x;
    int *ptr;

    ptr = returnGivenPointer1(&x);
    assert(ptr);
    *ptr = 42;

    returnGivenPointer2(NULL);
}
```

Flow-insensitive results:

```
returnGivenPointer1 :: ptr = {test :: x}
returnGivenPointer1 : returns = {test :: x}
returnGivenPointer2 :: ptr = {NULL}
returnGivenPointer2 : returns = {NULL}
test :: ptr = {test :: x}
```

Figure 11: Context-sensitive handling of function calls

```c
void test() {
    f();
    f();
}

void f() {
    g();
}

void g() {
}
```

Figure 12: Context-sensitive callgraph with indirect calls
2.4 Representations

When performing alias analysis on some input one needs an efficient data structure to represent these alias relations. Over time, three different kinds of data structures emerged.

**Equivalence classes.** This model puts all memory locations that might alias each other into the same equivalence class, and any two locations MAY alias iff they are in the same equivalence class. By doing so this model assumes that the alias relation is transitive. This is a sound yet imprecise assumption, as can be seen in the listing in Figure 14. The most accurate equivalence classes would be:

\[
\{ptrA\}, \{ptrB\}, \{ptrC\}, \{*ptrA, *ptrB, *ptrC, target1, target2\}.
\]

In the given input \(*ptrA\) and \(*ptrB\) may never alias one another, but both of them MAY alias with \(*ptrC\). Because of that all three of them must be part of the same equivalence class which means that \(*ptrA\) and \(*ptrB\) might alias as well, at least according to this model.

**Alias pairs.** While equivalence classes have the problem that the model is already imprecise, alias pairs (introduced in [5]) allow arbitrary alias relations to
be represented. In this representation the user has a set of alias pairs, where a single alias pair consists of two variables with some level of dereferencing. Two memory locations MAY alias each other iff such a pair is in the set. Considering again the code from Figure 14 the set of alias pairs would be:

\{(\ast ptrA, target1), (\ast ptrB, target2), (\ast ptrC, \ast ptrA), (\ast ptrC, target1), (\ast ptrC, \ast ptrB), (\ast ptrC, target2)\}.

As can be seen the problem with this representation is its redundancy, needing not only more memory than necessary but also allowing the representation of states that don’t make much sense.\(^4\)

There have been approaches (see [6] and [7]) where this problem has been addressed by limiting the amount of dereferences on each side of the pair and calculating any higher level information step-by-step when needed. These modifications basically resulted in similar performance as Points-to sets described next. Be advised that these compact representations are usually incomparable with the here presented explicit representation, meaning that both can be more accurate in some cases (see [7], and [8]).

Points-to sets. In [4] a new structure was introduced that does not longer capture whether two memory locations alias but defines a points-to relation between memory locations.\(^5\) Considering again the same piece of code from Figure 14 the final points-to set would be \{ptrA \rightarrow \{target1\}, ptrB \rightarrow \{target2\}, ptrC \rightarrow \{target1, target2\}\}. The example that was given in section 2 used this representation, and unless specified otherwise this is the representation used throughout this thesis.

---

\(^4\)E.g. \{(\ast \ast doublePtr, target), (\ast doublePtr, singlePtr), (\ast singlePtr, someOtherTarget)\}: This would mean that \ast \ast doublePtr and \ast singlePtr point to different memory location though \ast \ast doublePtr aliases with singlePtr.

\(^5\)Compared to alias pairs this would mean that only pairs of the form (\ast a, b) are present.
2.5 Selection of well-known algorithms

This subsection will provide a very short overview of a small selection of well-known algorithms used to do alias analysis. Let the reader be reminded that the trivial yet sound solution for \textit{MAY} and \textit{MUST} analyses is to answer all queries with \textit{MAY}.

\textbf{Address-taken.} The simplest non-trivial algorithm to do alias analysis puts all dereferencing expressions, and all variables whose address is taken in a single large \textit{equivalence class}. This means that any targets of pointers \textit{MAY} alias with each other or with any stack allocations or globals whose address are ever taken. Figure 15 displays a piece of code where the big equivalence class would be \{\texttt{\*ptrA}, \texttt{\*ptrB}, \texttt{x}\}.

\textbf{Andersen’s algorithm.} In [9] a flow-insensitive, context-insensitive, and field-insensitive algorithm using set constraints is described. It is based on the points-to representation and extracts subset constraints by processing every instruction exactly once, then this system of constraints is solved. The table in Figure 16 lists all important rules for generation of constraints and Figure 17 shows a small example, along with the constraints generated by it and their minimal solution.

\textbf{Steensgaard’s algorithm.} [10] presents one of the fastest known non-trivial algorithms (flow-, context-, and field-insensitive) to do alias analysis. According to [11] the precision is already substantially higher than the results produced by

---

\begin{verbatim}
1 int x, y;
2 int *ptrA, *ptrB;
3 ptrA = &x;
4 ptrB = ptrA;
5 ptrA = &x;
6 ptrB = ptrA;

Figure 15: Example code for address-taken algorithm

\begin{tabular}{ll}
\textbf{C instruction} & \textbf{Set constraint} \\
ptr = &target; & \{target\} \subseteq PointsTo(ptr) \\
ptrA = ptrB; & PointsTo(ptrB) \subseteq PointsTo(ptrA) \\
ptrA = *ptrB; & \forall b \in PointsTo(ptrB) : PointsTo(b) \subseteq PointsTo(ptrA); \\
*ptrA = ptrB; & \forall a \in PointsTo(ptrA) : PointsTo(ptrB) \subseteq PointsTo(a);
\end{tabular}

Figure 16: Basic rules for Andersen’s algorithm

\end{verbatim}
the address-taken algorithm\footnote{Size of points-to sets is in average about 8 times smaller}, although this method runs in (almost) linear time as well. This algorithm is based on a mix of equivalence classes and points-to sets: All memory locations, that might be pointed to from the same pointer, are in the same equivalence class. Using an efficient union-find structure and rules similar to the ones shown for Andersen’s algorithm these equivalence classes can be calculated using a single pass.

This is best explained using an example, please take a look at Figure \ref{fig:andersen}. Initially, before the scan, every variable represents its own equivalence class, pointing nowhere (step 1). Then the first instruction on line 6 is processed, now the equivalence class of $dPtr$ points to the equivalence class of $ptrA$ (step 2). After processing of the instruction on line 9 $dPtr$ needs to point to $ptrB$ as well, this is done by uniting the old pointed to equivalence class ($\{ptrA\}$) and the newly pointed to class ($\{ptrB\}$), leading to the graph shown as step 3. Similarly, after processing line 11 and 12, $a$ and $b$ end up in the same equivalence class.

\begin{figure}[h]
\centering
\begin{tabular}{ll}
\hline
\texttt{1 int a, b;} & \textbf{Constraints:} \\
\texttt{2 int *ptrA, *ptrB;} & Line 6: \\
\texttt{3 int **dPtr;} & $\{ptrA\} \subseteq PointsTo(dPtr)$ \\
\texttt{4 \hspace{1cm}} & \\
\texttt{5 if (\ldots) \{} & Line 9: \\
\texttt{6 \hspace{1cm} dPtr = &ptrA;} & $\{ptrB\} \subseteq PointsTo(dPtr)$ \\
\texttt{7 \hspace{1cm}}} & \\
\texttt{8 \hspace{1cm} \}} & Line 11: \\
\texttt{9 \hspace{1cm} dPtr = &ptrB;} & $\{a\} \subseteq PointsTo(ptrB)$ \\
\texttt{10 \}}} & \\
\texttt{11 ptrB = &a;} & Line 12: \\
\texttt{12 *dPtr = &b;} & $\forall x \in PointsTo(dPtr): \{b\} \subseteq PointsTo(x)$ \\
\hline
\end{tabular}
\caption{Example code for Andersen’s algorithm, generated constraints, and their solution}
\end{figure}

\begin{itemize}
\item $PointsTo(ptrA) = \{b\}$
\item $PointsTo(ptrB) = \{a, b\}$
\item $PointsTo(dPtr) = \{ptrA, ptrB\}$
\end{itemize}
2.5 Selection of well-known algorithms

```c
int a, b;
int *ptrA, *ptrB;
int **dPtr;

if (...) {
    dPtr = &ptrA;
} else {
    dPtr = &ptrB;
}
ptrB = &a;
*dPtr = &b;
```

Figure 18: Example for Steensgaard’s algorithm
3 Alias Analysis Library

3.1 Motivation and Design Decisions

As demonstrated by the previous chapter, reimplementing alias analysis for some specific task asks a lot from a developer, as there is a lot of literature around, and more to come. Considering the different possibilities concerning runtime requirements and precision, it might also be desirable to implement multiple algorithms as to compare their performance on a specific class of inputs. These circumstances ask for an abstraction, allowing a developer to use alias analysis in his project with minimal effort and only a basic understanding of its inner workings. AAL (Alias Analysis Library) constitutes a framework of simple components allowing a user to perform alias analysis on most common programming languages with only rudimentary knowledge of the topic. Similarly, the SSA-based (Static Single Assignment) intermediate representation is boiled down such as to contain only information of possible importance to alias analysis, allowing a developer intending to extend the library to focus on the problem at hand.

Design Decisions

- **Language-independent.** It should be possible to use AAL to analyze the major set of commonly used programming languages, either low-level languages, such as C, or high-level languages, such as Java. Also, the possibility to analyze low-level languages should not imply a penalty for analyzing high-level languages.

- **Minimal intermediate representation.** To achieve language-independence, an intermediate representation should be used which captures all aspects of relevance to alias analysis but nothing more, thereby keeping it tidy and reducing the required effort to implement new or alternative types of analyses. As path-sensitive analyses basically require the complete program to be able to increase precision, this type of analysis will not be supported by AAL.

- **SSA form.** By keeping the intermediate representation in a form similar to SSA (see [12], [13]) use-def chains are made explicit, which should in turn increase precision of flow-insensitive analyses. Furthermore, the fact that every variable is defined at exactly one point usually increases simplicity of analyses. Since most compilers already use SSA internally this requirement seems adequate.

- **Decoupling of context-sensitivity.** Section 2.2 has shown the basic concept of context-sensitivity by handling the same function as if it were

---

8Exceptions are currently not yet supported and must be approximated using other measures.
multiple functions, chosen based on the calling context. Based on this insight AAL should completely decouple the handling of context-sensitivity from the actual analysis, allowing arbitrary combinations of different levels of context-sensitive handlers and actual analyzers.

- **Precise approximation of dynamic calls.** By handling function pointers the same as any other pointers the library should determine targets of dynamic call sites while the analysis is running. By not just assuming a call to any function with matching arguments from a given dynamic call site, the spreading of coarse points-to information is reduced and precision enhanced.

- **Simple client-side interface.** The client interface for the user should be minimal yet simple to use, minimizing the potential for errors.

- **Extensible.** The framework should allow easy extension with new or alternative analyses, and a collection of pre-implemented data structures and algorithms often used by alias analyses should be supplied.

To be able to reach a competitive level of speed this library is implemented in C++. C would also be a reasonable choice, but the modularity provided by classes is expected to result in higher quality interfaces and overall structure of the system.

### 3.2 Intermediate representation: AALSSA

The intermediate representation, called AALSSA, reflects functions as a CFG (Control Flow Graph) of basic blocks. A basic block may have any number of successor basic blocks, each of which must be part of the same function. Exactly one basic block per function is marked as entry block and must not be the successor of any block. A basic block consists of a sequence of instructions, which will be explained below. Path-sensitive information is not represented, therefore transition from a block to any of the successor blocks is nondeterministic. AALSSA knows only a single type: references. These point to a certain offset of a logical memory block, which can be thought of as a sequence of fields (not bytes). A memory block is either a part of memory allocated using the special `alloc`-instruction, or represents the body of a function, thereby allowing to create references to functions, or function pointers. AALSSA supports global variables by allowing `alloc`-instructions at the global scope. A defined function is very similar to other global variables, i.e. the function itself is a global variable referencing the memory block used to represent this function. This allows to immediately use this global variable as call target when performing a static call. Functions either return nothing or a single reference. Finally, there is a single special function called ”`rootFunction`”, which should be used to represent the behavior of
### 3.2 Intermediate representation: AALSSA

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dst = src + 1;</code></td>
<td>Offset-instruction</td>
</tr>
<tr>
<td><code>dst = src[1];</code></td>
<td>Index-instruction</td>
</tr>
<tr>
<td><code>dst = {src1, src2};</code></td>
<td>Phi-instruction</td>
</tr>
<tr>
<td><code>dst = *srcPtr;</code></td>
<td>Load-instruction</td>
</tr>
<tr>
<td><code>*dstPtr = src;</code></td>
<td>Store-instruction</td>
</tr>
<tr>
<td><code>dst = new Object(2);</code></td>
<td>Alloc-instruction</td>
</tr>
<tr>
<td><code>fp(arg1, arg2);</code></td>
<td>Call-instruction</td>
</tr>
<tr>
<td><code>dst = fp(arg1, arg2);</code></td>
<td>Call-instruction, variant 2</td>
</tr>
<tr>
<td><code>return src;</code></td>
<td>Return-instruction</td>
</tr>
</tbody>
</table>

Figure 19: Example of AALSSA instructions

The instructions of AALSSA are all in SSA form, so no variable can be defined at multiple locations within the source. It is important to remember that AALSSA is used to represent an abstraction of the original program. Figure 19 shows examples of the eight different types of known instructions, each of which is explained here:

- **offset**: Applies a constant offset to the reference given as operand, so the new reference points the same memory block, but a different possibly higher field index. This instruction can also be used to create a reference pointing to $\top$ (points to any memory location, represented as ‘?’ in AALSSA).

- **index**: Takes a reference and an index, returns a reference to the specified element. The index can be positive, negative (in case it is used as pointer arithmetic), or unspecified. An analysis approximating the whole array with a single memory location (see [3.4.1]) can usually just ignore these and set $dst = src$. This is similar to the ”logic memory model” as it is used in [13].

- **phi**: This is very similar to a $\phi$-instruction in normal SSA form, it assigns one of the right-hand-sides to the left-hand-side. The difference is, this instruction ignores control flow (originating basic block), what turns this instruction into a simple nondeterministic assignment and it may also be used as such. The only requirement is that a phi-instruction is dominated by at least one of the given right-hand-sides.

- **load**: Loads a reference from the field specified by the operand. This is used as an abstraction for an actual load from memory.

---

For all other instructions all operands must dominate the instruction itself.
3.2 Intermediate representation: AALSSA

- **store**: Stores the specified reference to the specified field. This is used as an abstraction for an actual store to memory. Both operands may be \( \top \), thereby storing \( \top \) to the specified field or storing the specified reference to \( \top \) (a write to anywhere). Most common analyses ignore the src-operand if \( dst\text{Ptr} \) is \( \top \) and handle this case as \( *\top = \top \).

- **alloc**: Used for allocation, allocates an object with the specified amount of fields\(^{10}\) and returns a reference. Furthermore, an initializer block can be attached to setup the contents. This is a basic block which is only allowed to contain offset-, index-, and phi-instructions, as well as store-instructions which only access the currently initializing block.

- **call**: Used for either dynamic or static calls; a single reference is specified as call target. While all arguments can be either a reference or \( \top \), the call target is not allowed to be an explicit \( \top \).\(^{11}\) There is a second variant of the call-instruction, internally called ssacall-instruction, used to call functions when a return value is expected. Usage of this type of call will exclude any functions without a return value as possible call target.

- **return**: The return-instruction in AALSSA is not only used to return a reference to the callers, but has a bit a different semantic from what one might expect: Most functions, even those without a return value, should terminate every path returning to the caller with a return-instruction. If a possible execution path of a function is not terminated by a return-instruction it is assumed that such a path never returns to the caller and the points-to sets from this path won’t be merged back. Typical functions where it is reasonable to waive a return-instruction are exit() and abort().

All of this is best shown using an example: the first part of Figure 20 is a piece of C code, and the remaining parts are its translation to AALSSA.

**Globals.** Lines 1-6 define globals, which may be recognized throughout AALSSA by a prepended @-character. Since globals require memory, this is done using an alloc-instruction, the only instruction allowed at the global scope. In C, the amount of fields might not always be known at the allocation site, therefore a safe approximation is used: structS is the biggest aggregate in the input source, so it is assumed that every allocation site might allocate this type of object. One could argue that this is an unnecessary approximation for globals in C, but for the sake of simplicity the current frontend handles all allocations the same. AAL/AALSSA has no special support for NULL, instead a user creates a global

---

\(^{10}\)Depending on the language this might not be known at the site of allocation. In this case a safe approximation must used (see \(5.4.2\)).

\(^{11}\)If required, an implicit \( \top \)-reference can be created using an offset-instruction which is then used as call target. This will cause the library to approximate the list of possible call targets.
variable which she then uses instead (line 1). Lines 2-4 show the allocation of
@globalPtr, which can be thought of as having type int ** and pointing to the
global variable as it is written in the input code, not being the global variable
itself. This global has an initializer block, initializing the contents to NULL
(as usual in C), using a store-instruction. @globalTarget on the other hand
can be thought of as having type int* and is not initialized because its content
(*@globalTarget) has type int, which is of no relevance for pointer analysis.

Root function. Lines 7-9 display the _rootFunction, simulating the environ-
ment’s behavior. 0x8493de0 is an internal ID for the only basic block contained
in this function. For better precision the environment first constructs an array
of (fake) arguments instead of just calling main() with T (lines 9-11). Finally, it
calls main() and returns immediately after that. Note the prepended @-character
to the call target on line 12: since functions are like global variables, they also
have an @-character prepended.

Small functions. Lines 16-26 show the abstractions for the two functions on
lines 9-15 in the original source. The first function contains a simple store-
instruction (line 18), while the second function contains a load from a global
variable (line 24) and then returns the loaded value (line 25).

Main function. Lines 28-48 display the abstraction for the main() function.
As this function contains more than a single basic block, all blocks are annotated
with their predecessors and successors using their IDs. Lines 31 and 32 feature
again an allocation, as we have seen before, this time without any initializers as
ANSI C doesn’t initialize locals. Lines 33 and 34 show two (static) calls. The
name ptr.0 is a normal variable name and comes from value numbering performed
upon conversion to SSA form. The second basic block represents the loop body
and as ptr changes its value during the loop a phi-instruction has to be used (line
39). Line 40 calculates the reference to the second field of s (see line 30 of input
code). Finally, line 45 produces the reference to the second element of the array
arr and line 46 returns. Note that all stores of non-pointertype values in the
original source have been ignored (line 29 and 34).

Allocations. It might be considered uncommon that AALSSA makes no dif-
fERENCE between stack allocations and dynamic allocations. This does not harm
the cause in any way, as all stack allocations in a given program could be replaced
with dynamic allocations which are never freed, thereby leaking all locals once
per call. For a machine with infinite memory these two situations would behave
the same, and as memory is not modelled as a limited resource in alias analysis,
there is no difference in the results. An alternative would be to distinguish be-
tween stack allocations which are automatically freed upon return and dynamic
3.2 Intermediate representation: AALSSA

```c
int globalTarget;
int *globalPtr;

struct S {
  int f1;
  int f2;
};

void setGlobalPtr() {
  globalPtr = &globalTarget;
}

int *getGlobalPtr() {
  return globalPtr;
}

int main(int argc,
         char **argv) {
  int *ptr;
  int i;
  struct S s;
  int arr[2];

  setGlobalPtr();
  ptr = getGlobalPtr();

  for (i = 0; i < 2; ++i) {
    *ptr = 42;
    ptr = &s.f2;
  }

  ptr = &(arr[1]);
  *ptr = 42;

  return 0;
}
```

Figure 20: Input C code and corresponding AALSSA code
allocations which are freed using a special \textit{free}-instruction. This would also give the possibility to search for leaks or memory blocks freed multiple times.

### 3.3 Implementation overview

AAL can roughly be divided into two parts:

- The intermediate representation along with tools for storing, loading, dumping, and generation of a few simple source statistics.

- A framework for analyzers, along with implementations of often used data structures, especially points-to sets.

#### 3.3.1 Part 1 - Intermediate representation

The class structure for the intermediate representation is pretty straight-forward and can be seen in Figure 21. Though C++ has been used to implement AAL a distinction has been made between \textit{classes} and \textit{interfaces}, which can be recognized by the prefix "I". As AALSSA is a form of SSA all variables are assigned exactly once. The class structure exploits this: All instructions returning a reference are \textit{ISSAVariables} themselves, so they may immediately be used as operand for further instructions, no need to define a variable separately. \textit{ITargetable} is the interface implemented by all elements that can be targets of references (memory blocks), these are allocations and functions. \textit{FunctionSignatureVariables}, representing formal arguments of functions, and \textit{Functions} themselves are the only possible types of references which aren’t created by an instruction.

Furthermore, not shown in this Figure are tools to load and store an intermediate representation, dump it, generate statistics for some source, or extract and abstract callgraphs for a given input in AALSSA form.
3.3 Implementation overview

3.3.2 Part 2 - Analyzer framework

One of the design goals was to decouple context-sensitivity from the actual analysis, allowing arbitrary combinations of context-sensitivity-handlers and context-insensitive analyses, thereby turning them into context-sensitive analyses at zero cost. Many different such context-sensitivity-handlers, or internally called call-graph-constructors (CGC), can be implemented, giving fine-grained control over what level of context-sensitivity should be used, without having to adjust the actual analysis.

This is done by creating the notion of function instances: Analyses do not work over functions, but instead over function instances, and whenever a call site within such a function instance (called call site instance, short CSI) is reached, the analysis provides the CGC with the raw list of call targets, according to the analysis. The CGC then first determines which functions might be called from this CSI and then decides which function instances the actual targets for this call should be, thereby constructing a precise approximation of the PCG (Procedure callgraph) at the same time. Then the analysis continues, using the specified list of function instances as possible call targets. Currently, the following CGCs are already implemented:

- **ContextInsensitiveCGC**: This handler creates at most one function instance per function. Then, whenever a function $f$ is determined to be a possible call target for some CSI, the according function instance is added as call target for this CSI.

- **InlineSmallCGC**: This handler simulates inlining of all calls which don’t contain any calls themselves. First, all functions in the input are scanned for calls and a function instance is created for all functions containing at least one call. Then, whenever a function $f$ is determined to be a possible call target for some CSI and does contain at least one call, the according function instance is added as call target for this CSI, so this behaves the same as the context-insensitive CGC. However if this function $f$ does not contain at least one call, a new function instance of $f$ is created specifically for this CSI and added as possible call target.

- **ContextSensitive**: This handler always creates a new function instance for every CSI, except if there is a recursive or mutually recursive call chain, in which case this would lead to an infinite amount of function instances. Whenever a function $f$ is determined to be the call target for some CSI, and another function instance of $f$ is already on the callstack when this CSI is reached, the very same function instance that is already on the callstack is added as a possible call target instead of a new function instance, resulting in an approximation of the present recursion. This CGC may obviously result in an exponential amount of function instances compared to the amount of original functions.
Currently implemented are a flow-insensitive and a flow-sensitive analyzer, approximating only MAY alias informations.

3.4 Alias analysis in practice

When trying to apply the techniques described in section 2 to a language such as C different kinds of problems arise because of calls to external functions or function pointers, usage of structures or arrays, pointer arithmetics or casts, and so on. This section points out these problems and gives approaches and pointers to solve them.

3.4.1 Specifics

Structures or classes define a list of member fields and can be handled in different ways, the typical being field-insensitive, field-based, and field-sensitive. Field-insensitive handling means all operations performed on any of the fields are merged and approximated as a single memory location (per instance). For field-based handling every member of a structure is approximated using a single memory location over all instances of this structure. Finally field-sensitive handling approximates all fields of every instance as its own memory location. Figure 22 gives a short example and the according results.

When dealing with C the field-based approach has the disadvantage that under ISO C99[15] a basic kind of ”inheritance” is specifically supported: C guarantees that between any two structs the ”prefix” of mutually shared field declarations are located at the same location in memory, allowing the programmer (for example) to have an integer called type as the very first member of many different structs and cast instances according to this type. This means for the field-based approach to be sound all these overlapping fields must be approximated using a single memory location.

[16] presents a simple approach to implement a field-sensitive (flow- and context-insensitive) analysis for C and similar languages and is actually the basis for the current alias analysis algorithm used in GCC.

Arrays may often be of dynamic size or are accessed using a dynamically calculated index, so in general a static analysis must be able to deal with arrays of unknown size and unknown access indexes. Common approaches include approximating the whole array using a single memory location (as it is done with structures in the the field-insensitive approach) or a fixed amount of memory locations (e.g. 2, one for the first element and one for the rest, allowing the analysis to check whether a pointer points to the head of an array). When the size and all indexed accesses can be statically determined it is of course also a possibility to model all elements as a memory location of its own.
3.4 Alias analysis in practice

```c
struct S {
    int *field1;
    int *field2;
} instance1, instance2;

int target1, target2, target3;

instance1.field1 = &target1;
instance1.field2 = &target2;
instance2.field1 = &target3;
```

Field-insensitive results:

\[
\text{instance1} = \{\text{target1, target2}\} \\
\text{instance2} = \{\text{target3}\}
\]

Field-based results:

\[
S_{\text{field1}} = \{\text{target1, target3}\} \\
S_{\text{field2}} = \{\text{target2}\}
\]

Field-sensitive results:

\[
\text{instance1}_{\text{field1}} = \{\text{target1}\} \\
\text{instance1}_{\text{field2}} = \{\text{target2}\} \\
\text{instance2}_{\text{field1}} = \{\text{target3}\}
\]

Figure 22: Different ways of handling fields of a structure
void test() {
    int target1, target2;
    int **ptr1 = allocIntPtr();
    int **ptr2 = allocIntPtr();
    *ptr1 = &target1;
    *ptr2 = &target2;
}

int **allocIntPtr() {
    return (int**) malloc(sizeof(int*));
}

Flow-insensitive results:

\[
\begin{align*}
\text{test} :: \text{ptr1} &= \{\text{allocIntPtr} :: \text{malloc1}\} \\
\text{test} :: \text{ptr2} &= \{\text{allocIntPtr} :: \text{malloc1}\} \\
\text{allocIntPtr} :: \text{malloc1} &= \{\text{test} :: \text{target1}, \text{test} :: \text{target2}\} \\
\text{allocIntPtr} :: \text{return} &= \{\text{allocIntPtr} :: \text{malloc1}\}
\end{align*}
\]

Figure 23: Handling of dynamically allocated data structures

**Heap objects** or dynamically allocated objects raise a similar problem as arrays: Generally, there is no way for a static analysis to determine the amount of dynamically allocated objects, so multiple dynamically allocated objects must be approximated using a static amount of memory locations. An approach often used is to distinguish dynamically allocated objects based on their allocation site in the code (“named objects”, first appeared in [17]). Figure 23 displays a small example for this method. Note that the precision of this method can again be adjusted by taking the callstack into account, similar to context-sensitivity explained in section 2.2.

**Function pointers** can usually be handled without any major effort by modelling every defined function as a normal global variable. When processing a dynamic call site the algorithm needs to be able to name all memory locations the function pointer used might refer to, which may then be used to determine a subset of functions that might be called from the given call site. Since alias analysis algorithms are an approximation, functions incompatible with this specific call site or even non-function memory locations might be returned as a possible call target. Common approach is to filter these (i.e. ignore them as possible
call targets) and print a warning since this might actually happen in a language like C. The handling of arguments and return values is highly implementation-dependent, but a typically working approach is to handle them the same way structures are dealt with. If the amount of arguments is variable, a single special argument which is handled like an array can be used.

3.4.2 Problems

External functions. In practice a program often calls external functions for which no implementation is available during analysis. While for standardized functions an approximation may be supplied the analysis must make worst-case assumptions for any other external functions. Depending on the possibilities the used language offers this might mean that some or all of the points-to sets are not known anymore after such a call returns. These pointers might point to anything, what translates to points-to sets containing the whole universe of targets or $\top$ (top), when looking at the problem as a lattice. It is usually desirable to support points-to sets being $\top$ specifically since stores and loads from such a pointer may have a big impact.

Casts and pointer manipulation. C allows the programmer to do direct modifications on pointers (e.g. by casting them to an integer). ANSI C declares any of these operations as implementation defined and mainly low-level code uses such operations (e.g. malloc() implementations typically store information in the lower bits of aligned pointers). These cases are usually handled by setting the points-to sets of these pointers to $\top$.

Pointer arithmetic. While direct pointer manipulation is implementation defined pointer arithmetic is well-defined by to ANSI C for navigation within arrays of variable or constant size (see [15] section 6.5.6 concerning additive operators). Using pointer arithmetics for anything else than navigation within an array is not only implementation defined but rendered undefined\footnote{Pointers to an object that is not an element of an array are defined to behave the same as a pointer to the first element of an array with size 1.}. This allows an analysis approximating arrays as a single memory location (as described in section 3.4.1) to just ignore these expressions.

Type unknown at allocation site. Dynamic allocation in C is usually done by calling malloc() with the amount of needed bytes, leaving the analysis with the task to figure out what type of object is actually allocated. For smaller projects its often not too hard to determine the actual type since the returned pointer is casted immediately, but as the projects become bigger encapsulations choosing a specific version of malloc() become more common, making it harder for the
analysis to determine the type of the allocated object. A common approach is to assume that a union of all types appearing in the input is allocated.

**Unions and overlapping fields.** When defining a union such as `unionU{int * ptr; float f;};` the two fields `ptr` and `f` alias (at least partially). When analyzing ANSI C the analysis may safely ignore these cases because ANSI C defines that only objects of "similar type" may alias each other. For the definition of "similar types" see [15] section 6.5 point 7.

⊤ (top) function pointers. With everything described so far it might happen that a function pointer used in a call is ⊤. In this case the common approach is to go through the list of defined functions and adding all functions compatible with the given arguments as a possible target. Checking which of these functions are used for anything else than a direct call allows further filtering.

### 3.5 Experimental results

To evaluate the performance of AAL, a selection of 15 often used open source programs will be used as benchmarks: *bchunk*, a small tool for conversion of cd images, *JHead*, an exif jpeg parser, the SAT solver *Limmat*, the arbitrary precision calculator *bc*, the *gzip* compressor software, the file manager of Linus Torvald’s *git* version control system, the lexical analyzer *flex*, the parser generator *bison*, GNU’s version of *make*, the download tool *Wget*, the *Fetchmail* mail retriever and forwarder, ISC’s *inn* usenet daemon, the *ntp* network time server, the famous *bash*, and *cvs*, the concurrent versioning system.

**Frontend.** The frontend to convert these C programs to AALSSA is implemented as an LLVM[15] pass. LLVM (Low Level Virtual Machine) is "[.] a compiler framework designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations [.]." [15]. It defines a simple low-level representation in SSA (Static Single Assignment) form, has a simple type-system, and is well documented. Furthermore, an adaptation of GCC is available, allowing parsing and compilation of most available C projects, thereby designating it as a good basis for the frontend pass. The pass itself contains safe and precise approximations for most standardized C functions, for all other external functions (such as syscalls or calls to libraries) the frontend generates a safe (and very imprecise) approximation (usually a store of ⊤ to ⊤: ∗⊤ = ⊤).

**Hardware.** All benchmarks have been performed on an Intel Xeon "Harpertown" (X5472), an x86-64-architecture, at 3.0 GHz and with 15.7 GB of memory (though the software was limited to 13 GB).
3.5 Experimental results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SLOCs</th>
<th>Derefs</th>
<th>LL VM’s Andersen not NULL time [s]</th>
<th>AAL: FI, CI not NULL time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>bchunk-1.2.0</td>
<td>379</td>
<td>151</td>
<td>69 (46%)</td>
<td>107 (71%)</td>
</tr>
<tr>
<td>jhead-2.6</td>
<td>3197</td>
<td>697</td>
<td>576 (83%)</td>
<td>631 (91%)</td>
</tr>
<tr>
<td>limmat-1.3</td>
<td>6279</td>
<td>1112</td>
<td>361 (32%)</td>
<td>394 (35%)</td>
</tr>
<tr>
<td>bc-1.06: bc</td>
<td>6535</td>
<td>2142</td>
<td>1394 (65%)</td>
<td>1157 (54%)</td>
</tr>
<tr>
<td>gzip-1.3.9</td>
<td>8606</td>
<td>2258</td>
<td>2102 (93%)</td>
<td>2124 (94%)</td>
</tr>
<tr>
<td>flex-2.5.35</td>
<td>14042</td>
<td>7676</td>
<td>5527 (72%)</td>
<td>5678 (74%)</td>
</tr>
<tr>
<td>bison-2.3</td>
<td>22386</td>
<td>11539</td>
<td>6993 (61%)</td>
<td>7401 (64%)</td>
</tr>
<tr>
<td>make-3.81</td>
<td>22904</td>
<td>5530</td>
<td>3032 (55%)</td>
<td>2049 (37%)</td>
</tr>
<tr>
<td>git-4.9.4: gitfm</td>
<td>23305</td>
<td>5088</td>
<td>2574 (51%)</td>
<td>2243 (44%)</td>
</tr>
<tr>
<td>wget-1.10.2</td>
<td>25924</td>
<td>5223</td>
<td>2577 (49%)</td>
<td>3051 (58%)</td>
</tr>
<tr>
<td>fetchmail-6.3.8</td>
<td>28972</td>
<td>6120</td>
<td>3344 (55%)</td>
<td>3495 (57%)</td>
</tr>
<tr>
<td>inn-2.4.5</td>
<td>42061</td>
<td>9504</td>
<td>4089 (43%)</td>
<td>3724 (39%)</td>
</tr>
<tr>
<td>ntp-4.2.4p5: ntp</td>
<td>68641</td>
<td>17841</td>
<td>8266 (46%)</td>
<td>8757 (49%)</td>
</tr>
<tr>
<td>bash-3.2</td>
<td>72863</td>
<td>24225</td>
<td>13967 (58%)</td>
<td>13608 (56%)</td>
</tr>
<tr>
<td>cvs-1.12.2</td>
<td>75877</td>
<td>20601</td>
<td>9226 (45%)</td>
<td>9143 (44%)</td>
</tr>
<tr>
<td><strong>Averages</strong></td>
<td></td>
<td></td>
<td>4273 (57%)</td>
<td>4237 (58%)</td>
</tr>
</tbody>
</table>

Figure 24: Comparison of Andersen’s algorithm with the flow-insensitive, context-insensitive AAL analysis. "Derefs" names the amount of loads and stores which cannot trivially be proven unequal to NULL (see 3.5.1). "not NULL" names the amount of loads and stores whose base pointers could be proven unequal to NULL using the corresponding analysis, excluding the same cases as for "Derefs".

3.5.1 Comparison to LLVM’s Andersen’s analysis

In a first step I’d like to evaluate the precision and practicality of AAL by doing a basic comparison with the implementation of a field-sensitive variant of Andersen’s algorithm shipped along with LLVM. Assuming no program ever wants to dereference a NULL-pointer, the comparison looks at all loads (\(dst = *srcPtr\)) and stores (\(*dstPtr = src\)) in the original code (not AALSSA) and checks for a possible alias of the base pointer with NULL, then the amount of dereferences proven to have a base pointer unequal to NULL are counted\(^{13}\). To sustain a certain degree of comparability with other implementations, all loads and stores where the pointer operand is the immediate result of LLVM’s \texttt{alloca}-instruction (stack allocation) are ignored from this count, because these can trivially be proven unequal to NULL and other implementations usually don’t consider accesses to stack variables as dereferencing.

\(^{13}\)Slight adjustments had to be made to LLVM’s implementation to allow this type of comparison.
Figure 24 shows the results of this comparison. While AAL needs a little bit more time in all cases (in average almost 90%, including the time required for translating the input to AALSSA), the precision of both algorithms is more or less the same. LLVM’s analysis scores slightly lower than AAL for smaller programs but is able to compensate that when analyzing bigger programs, most prominently on the benchmark ”make”.

3.5.2 Precision of AAL

Comparison of the precision of different types of AAL analyses is done in a way similar to [19] and [7], by looking at the size of points-to sets. Once the analysis is completed, the original program (not the AALSSA abstraction) is scanned again for loads and stores and the average number of possible targets is determined.\footnote{Similar to before, this scan ignores all loads and stores where the base pointer is a stack allocation and the number of targets therefore trivially 1.}

At high precision this average should be close to 1. Should the base pointer be a $\top$-reference it is excluded from this average and counted separately. Figure 25 gives the numerical results for the described measurements.

For all analyses listed the overall precision slightly decreases for bigger programs. Though somewhat unexpected this can be explained by reusage of code: It must be assumed that bigger programs have more shared code. As an example, a small program might use a linked list for a single, specific purpose, thereby allowing precise approximation of this list. On the other hand, a more elaborate piece of code could have a general implementation for linked lists and reuse this for managing different lists, thereby decreasing the precision of the analysis.

In general, the increase of precision when switching to higher-quality approximations, though present, is slim, but the costs are sometimes exceptionally higher, as we will see later. It is my belief that this situation has to be looked at from another perspective: Even the precision of the most inaccurate analysis (FI/CI) shows already a pretty astonishing results. It is probable that this precision follows directly from the design decision to use a sort of SSA-alike form, thereby turning flow-insensitive analyses into flow-sensitive analyses for everything but memory accesses (stores and loads, see 3.1). The substantial amount of $\top$-references will be investigated further in section 3.5.4.

3.5.3 Requirements

Figure 26 shows the required processing time and memory\footnote{Both time and memory usage are measured over the whole process of translating the input to AALSSA, approximating a result, and measuring its precision. Experiments have shown these effects to be negligible.} for all benchmarks and analyses. It is obvious to see that the FI/CI-analyses has by far the lowest requirements with at most 2.98 seconds processing time and 127 MB used memory.
### Experimental results

#### 3.5.1 ALIAS ANALYSIS LIBRARY

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SLOCs</th>
<th>FI / CI (avg.)</th>
<th>FS / CI (avg.)</th>
<th>FI / IS (avg.)</th>
<th>FS / IS (avg.)</th>
<th>FI / CS (avg.)</th>
<th>FS / CS (avg.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bchunk-1.2.0</td>
<td>379</td>
<td>1.17</td>
<td>1.17</td>
<td>1.17</td>
<td>1.17</td>
<td>1.17</td>
<td>1.17</td>
</tr>
<tr>
<td>jhead-2.6</td>
<td>3197</td>
<td>1.06 (7)</td>
<td>1.06 (7)</td>
<td>1.06 (7)</td>
<td>1.06 (7)</td>
<td>1.03 (7)</td>
<td>1.03 (7)</td>
</tr>
<tr>
<td>limmat-1.3</td>
<td>6279</td>
<td>1.17 (64)</td>
<td>1.17 (64)</td>
<td>1.13 (64)</td>
<td>1.12 (61)</td>
<td>1.00 (64)</td>
<td>1.00 (61)</td>
</tr>
<tr>
<td>bc-1.06: bc</td>
<td>6535</td>
<td>1.16 (42)</td>
<td>1.16 (41)</td>
<td>1.15 (42)</td>
<td>1.15 (41)</td>
<td>1.14 (42)</td>
<td>timeout</td>
</tr>
<tr>
<td>gzip-1.3.9</td>
<td>8606</td>
<td>1.04 (5)</td>
<td>1.06 (5)</td>
<td>1.05 (5)</td>
<td>1.01 (5)</td>
<td>1.01 (4)</td>
<td>1.01 (4)</td>
</tr>
<tr>
<td>flex-2.5.35</td>
<td>14042</td>
<td>1.06 (23)</td>
<td>1.06 (23)</td>
<td>1.05 (23)</td>
<td>1.03 (23)</td>
<td>1.03 (23)</td>
<td>1.03 (23)</td>
</tr>
<tr>
<td>bison-2.3</td>
<td>22386</td>
<td>1.11 (30)</td>
<td>1.11 (30)</td>
<td>1.10 (30)</td>
<td>1.10 (30)</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td>make-3.81</td>
<td>22904</td>
<td>1.15 (58)</td>
<td>1.15 (58)</td>
<td>1.15 (58)</td>
<td>1.15 (58)</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td>git-4.9.4: g corruption</td>
<td>23305</td>
<td>1.13 (44)</td>
<td>1.13 (44)</td>
<td>1.13 (44)</td>
<td>1.13 (44)</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td>wget-1.10.2</td>
<td>25924</td>
<td>1.52 (36)</td>
<td>1.52 (36)</td>
<td>1.51 (35)</td>
<td>1.51 (35)</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td>fetchmail-6.3.8</td>
<td>28972</td>
<td>1.15 (36)</td>
<td>1.15 (36)</td>
<td>1.14 (36)</td>
<td>1.14 (36)</td>
<td>1.05 (35)</td>
<td>OOM</td>
</tr>
<tr>
<td>inn-2.4.5</td>
<td>42061</td>
<td>1.30 (26)</td>
<td>1.30 (26)</td>
<td>1.29 (26)</td>
<td>1.29 (26)</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td>ntp-4.2.4p5: ntpd</td>
<td>68641</td>
<td>1.23 (44)</td>
<td>1.23 (44)</td>
<td>1.23 (44)</td>
<td>1.23 (44)</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td>bash-3.2</td>
<td>72863</td>
<td>1.11 (40)</td>
<td>1.11 (40)</td>
<td>1.11 (40)</td>
<td>1.11 (40)</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td>cvs-1.1.22</td>
<td>75877</td>
<td>1.43 (37)</td>
<td>1.43 (37)</td>
<td>1.66 (37)</td>
<td>1.66 (37)</td>
<td>OOM</td>
<td>OOM</td>
</tr>
</tbody>
</table>

**Averages**

<table>
<thead>
<tr>
<th></th>
<th>FI / CI (avg.)</th>
<th>FS / CI (avg.)</th>
<th>FI / IS (avg.)</th>
<th>FS / IS (avg.)</th>
<th>FI / CS (avg.)</th>
<th>FS / CS (avg.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.19 (35)</td>
<td>1.19 (35)</td>
<td>1.20 (34)</td>
<td>1.20 (34)</td>
<td>1.06 (29)</td>
<td>1.05 (24)</td>
</tr>
</tbody>
</table>

Precision statistics of AAL’s flow-insensitive (FI) and flow-sensitive (FS) analyses combined with different context-sensitivities (see §3.3.2): context-insensitive (CI), inline-small (IS), and context-sensitive (CS). “OOM” stands for “out of memory”, with a limit of 13GB. A timeout occurred after 1 hour of processing.
In general for these benchmarks, the following ordering always holds for memory usage and almost always for processing time: $FI/CI < FI/IS < FS/CI < FS/IS$, and, where results are available: $FI/CS < FS/CS$. Particularly interesting is the benchmark "bc-1.06: bc": While the fully context-sensitive analyses (CS) have a spike in both processing time and memory usage, the only partially context-sensitive (IS) analyses decrease to the same level as context-insensitive (CI) analyses, yet there is almost no difference in precision. A probable cause for this behavior is a highly recursive structure, which can be expected in a calculator software.

### 3.5.4 Origin of $\top$-references

As seen in Figure 25 the relative amount of $\top$-references ranges from 4% to 64% (average of 33%, standard deviation of 16%) for the given benchmarks. As this portion can be considered substantial I tried to determine the main causes by following a random selection to their point of origin. Initially, two main causes could be identified, first and foremost calls to external functions which had to be replaced with safe approximations (see 3.5). The second main cause was identified to be $\text{main}(\text{argc}, \text{argv}, \text{envv})$ being called with $\text{argv}$ and $\text{envv}$ being $\top$, though the effect of this obviously decreased for bigger programs. Once the frontend had been adjusted to contain more precise approximations for most standard C functions, and to generate a more accurate $\text{rootFunction}()$ calling $\text{main}(\ldots)$ with more precise approximations for $\text{argv}$ and $\text{envv}$ than $\top$, the remaining main cause could be tracked to still unknown external functions, such as syscalls and library functions.

### 3.5.5 Influential input characteristics

A question that often came up during the development of AAL/AALSSA was which characteristics of a given input influence a specific performance value, such as "does the amount of lines of codes in a source program influence the time required for processing?". In this section I’d like to determine these factors using a simple least-squares approach.

The basic idea is to try and estimate the performance values (average number of targets for non-$\top$ base pointers, number of $\top$-references, time requirements, and memory requirements) based on the characteristics shown in Figure 27 using least-squares. As least-squares is only able to solve linear problems the squares of all characteristics are added to this matrix as well, allowing some quadratic approximations as well. Furthermore, to make up any constant parts, a constant column of 1’s is added as well. Then the algorithm selects up to 6 columns of input values (the constant input column is always selected) and extracts the optimal weights using 75% of all benchmarks. If this selection of criteria along with their weights minimizes the overall error when trying to extrapolate the results
3.5 Experimental results

Figure 26: Usage of resources for different analyses.
### 3.5 Experimental results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SLOCs</th>
<th>#Ext</th>
<th>AALSSA stats</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>#Fun</td>
</tr>
<tr>
<td>bchunk-1.2.0</td>
<td>379</td>
<td>9</td>
<td>33</td>
</tr>
<tr>
<td>jhead-2.6</td>
<td>3197</td>
<td>58</td>
<td>90</td>
</tr>
<tr>
<td>limmat-1.3</td>
<td>6279</td>
<td>23</td>
<td>195</td>
</tr>
<tr>
<td>bc-1.06: bc</td>
<td>6535</td>
<td>59</td>
<td>161</td>
</tr>
<tr>
<td>gzip-1.3.9</td>
<td>8606</td>
<td>75</td>
<td>179</td>
</tr>
<tr>
<td>flex-2.5.35</td>
<td>14042</td>
<td>408</td>
<td>322</td>
</tr>
<tr>
<td>bison-2.3</td>
<td>22386</td>
<td>718</td>
<td>804</td>
</tr>
<tr>
<td>make-3.81</td>
<td>22904</td>
<td>464</td>
<td>409</td>
</tr>
<tr>
<td>git-4.9.4: gitfm</td>
<td>23305</td>
<td>414</td>
<td>557</td>
</tr>
<tr>
<td>wget-1.10.2</td>
<td>25924</td>
<td>482</td>
<td>595</td>
</tr>
<tr>
<td>fetchmail-6.3.8</td>
<td>28972</td>
<td>797</td>
<td>464</td>
</tr>
<tr>
<td>inn-2.4.5: innd</td>
<td>42061</td>
<td>832</td>
<td>887</td>
</tr>
<tr>
<td>ntp-4.2.4p5: ntpd</td>
<td>68641</td>
<td>1038</td>
<td>1092</td>
</tr>
<tr>
<td>bash-3.2</td>
<td>72863</td>
<td>1830</td>
<td>2344</td>
</tr>
<tr>
<td>cvs-1.11.22</td>
<td>75877</td>
<td>1171</td>
<td>1274</td>
</tr>
</tbody>
</table>

**Legend:**

- **SLOCs**: Source lines of code
- **#Ext**: Number of static calls to external functions
- **#Fun**: Number of functions
- **#BB**: Total number of basic blocks
- **#Inst**: Number of instructions
- **#Lo**: Number of loads
- **#Sto**: Number of stores
- **#Call**: Number of calls

Figure 27: Characteristics for all benchmarks

For the remaining 25% of the benchmarks, the algorithm marks this selection of criteria as optimal for a specific performance value. Figure 28 sketches the used algorithm, which I applied to all analyses with complete result sets (i.e. \{FlowInsensitive, FlowSensitive\} × \{ContextInsensitive, InlineSmall\}).

**Findings.** Figure 29 lists which characteristics are determined most influential ("correlated") for all different types of analyses and types of performance measurements. As sketched in the algorithm in Figure 28, the actual values of the determined weights have been dropped and merely the selection of which characteristics get a weight unequal to zero is shown.

In my opinion one of the most interesting results is the fact that both performance measurements concerning precision name the number of external functions as an influential factor, either in its linear form or squared. This coincides with the findings presented in section 3.5.4 about the origin of T-references. While the very same characteristics are considered influential for the number of T-references, regardless of the type of analyses, the average number of targets for dereferences is influenced by one of two sets, depending on context-sensitivity.
3.5 Experimental results

function [solution] = search(inputs, results)
% allow simple quadratic approximations
inputs = appendSquaredCriteria(inputs);
%
% add column of constant 1s
inputs = prependConstant1Criterion(inputs);

minError = someBigNumber;

for each selection selCrit in "up to 5 criteria + the constant criterion"
    conInput = constrainInputCriteria(inputs, selCrit);
    currentError = 0;
    for each selection selBench in "75% of all benchmarks"
        weights = calcWeightsUsingLeastSquares(conInput, results, selBench);
        approxRes = approxResults(weights, allBenchmarks - selBench);
        currentError = currentError + calcError(results, approxRes);
    end
    if currentError < minError
        minError = currentError;
        solution = selCrit;
    end
end

Figure 28: Pseudocode for search of important input characteristics (criteria)
Concerning the analysis of time performance measurements, the main difference lies, as expected, between the flow-insensitive and flow-sensitive analyses. For both flow-insensitive analyses the amount of lines of code squared matters (among others), which can be explained by the internal fixpoint-iteration. The flow-sensitive analyses both depend on the same choice of characteristics, which is larger than in the flow-insensitive case. This suggests that this partial context-sensitivity has only a minimal impact on time requirements and that the flow-sensitive analysis mostly determines the amount of required processing time. I’d like to point out the choice of the number of basic blocks as influential characteristic: Compared with other types of characteristics (e.g. number of functions or instructions) this is the only relict representing some indication of the complexity of flow-sensitive structuring within the current program\cite{16}, so the selection of this characteristic seems reasonable.

The analysis didn’t give many conclusive results for memory requirements. The amount of functions has been determined influential for all 4 types of analyses, which seems reasonable since at least one characteristic indicating the size of the source being analyzed should be chosen.

3.6 Conclusion

The flow-sensitive $MAY$-analysis costs a lot more resources than the flow-insensitive variant and yet the return in increased precision is marginal at best. The next step would be a flow-sensitive analysis calculating $MAY$ and $MUST$ informations: If the analysis is able to prove that some reference is overridden and pointing to a new target afterwards, it may replace the old list of possible targets with a new list of targets, containing only the new target, thereby increasing precision. In data-flow analyses this is called a kill set or kill information. Still, the current flow-sensitive analysis has some spikes in memory and processing time requirements (e.g. benchmarks ”jhead-2.6” and ”ntp-4.2.4p5: ntpd”), and it must be assumed that these will still be present in the extended implementation, possibly even worse. This kind of behavior might be considered impractical for use in practice, especially in compilers.

On the other hand, the precision of the flow-insensitive analysis is exceptionally high, e.g. compared to \cite{17}, where the average precision for the flow-insensitive analysis is around 2.2 targets per dereferencing. The benchmarks analyzed there are fairly small, the biggest input has about 30k SLOCs and the second biggest less than 8k. The average is around 1k SLOCs. This comparison is somewhat cumbersome as paper \cite{17} doesn’t deal with $T$-references specifically and instead eliminates those by not having any external dependencies\cite{18}, but it gives a rough idea.

\footnote{Assuming the CFG and thereby the amount of basic blocks is optimized, which I did.}

\footnote{Most tested programs were artificial, not real-life software.}
### 3.6 Conclusion

#### ALIAS ANALYSIS LIBRARY

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Const 1</th>
<th>SLOCs</th>
<th>#Ext</th>
<th>#Fun</th>
<th>#BB</th>
<th>#Inst</th>
<th>#Lo</th>
<th>#Call</th>
<th>SLOCs²</th>
<th>#Ext²</th>
<th>#Fun²</th>
<th>#BB²</th>
<th>#Inst²</th>
<th>#Lo²</th>
<th>#Sto²</th>
<th>#Call²</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FI / CI</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>FS / CI</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>FI / IS</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>FS / IS</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

### Legend:

- **SLOCs**: Source lines of code
- **#Ext**: Number of static calls to external functions
- **#Fun**: Number of functions
- **#BB**: Total number of basic blocks
- **#Inst**: Number of instructions
- **#Lo**: Number of loads
- **#Sto**: Number of stores
- **#Call**: Number of calls
- **FI**: flow-insensitive
- **FS**: flow-sensitive
- **CI**: context-insensitive
- **IS**: inline-small (see §3.3.2)
- **Prec:** Number of $T$-references
- **Prec:avg**: Average number of targets
- **Time**: Processing time
- **Memory**: Memory usage

Figure 29: Influential input characteristics for different types of analyses.
Using a simple, least-square-based approach I was able to get a better picture of what properties of a given input program influence the performance of the different analyses, most prominently displaying the major effect of calls to external functions on precision.

With AAL the community gets a simple tool to add support for alias analyses to a project without the need to know all the literature. Especially for smaller projects AAL offers a good solution to get competitive results.

**Related work.** Using an intermediate representation similar to AALSSA, [20] presents a flow-sensitive, context-insensitive analysis by dissolving most parts of the typically known structure of functions and replacing it with control and data dependencies. A major weight was put on optimizing this specific analysis by (e.g.) extracting the loop tree, giving the possibility to calculate the the fixpoint for all contained loops before the containing loop is approximated. It does not describe something similar to the handling of context-sensitivity as described here. However, in [21] a context-sensitive analysis is presented by cloning functions for every calling context, similar to function instances described here, thereby producing context-sensitive results using a context-insensitive algorithm.
4 Application

Calysto\([1]\), shortly introduced in section 4.2, is a static checker, mostly used for bug-hunting, and contains a few unsound assumptions concerning aliases. In section 4.3 a tool similar to Calysto will be presented, AACalysto, where AAL will be used to discharge some of those assumptions. As a bit-precise prover is needed, section 4.1 will give an overview of the inner workings of bit vector logic and the used implementation, AAProver.

4.1 Bit Vector Logic

When trying to prove the assertion

\[
\text{assert}( (x == 5) && (y == x^3) && (y + x == 22 >> 1))
\]

(with x and y being defined as an int) in C normal mathematical rules aren’t adequate as they don’t take representation-dependent properties such as overflows into account and translation of bit operations would be cumbersome to say at least. Provers supporting bit vector logic (sometimes called modular arithmetic) are capable of capturing these properties by modelling integers as what they really are, vectors of bits (booleans). Such a prover can typically be decomposed into 5 steps:

1. Parsing of the given input

2. Translation of all operations to boolean expressions
   This is usually done by replacing every operation by its equivalent as a circuit. For example for a bitwise ”or” ( | ) the resulting bit vector is a vector of \( \lor \)-operations of the corresponding bits of the left-hand-side and right-hand-side, and for an addition the resulting bit vector is constructed using a sequence of full adders. This transformation generates a DAG (directed, acyclic graph) sharing many nodes (since a variable \( x \) may occur multiple times in the formula).

3. Elimination of constants

4. Generation of CNF from formula using Tseitin tautologies
   The formula generated by the previous step may have arbitrary depth of boolean operations. Tseitin tautologies\([22]\) allow translation of the given formula to CNF in linear time without exponential blowup. Following are the Tseitin tautologies used to replace \( \land \) and \( \lor \)-operations on the left side. The right side feature the same formula translated to CNF.

<table>
<thead>
<tr>
<th>Tseitin tautologies</th>
<th>Equivalent CNFs</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a \equiv b \land c))</td>
<td>((a \lor \neg b \lor \neg c) \land (\neg a \lor b) \land (\neg a \lor c))</td>
</tr>
<tr>
<td>((a \equiv b \lor c))</td>
<td>((\neg a \lor b \lor c) \land (a \lor \neg b) \land (a \lor \neg c))</td>
</tr>
</tbody>
</table>
These equivalences may now be used to flatten an arbitrary boolean formula so the result is a single big conjunction of equivalences. Finally, these equivalences can be replaced by the right side of the table above resulting in a pure CNF.

**Example.** Assume a formula \((a \lor b) \land \neg(a \lor b)\) where the children of the \(\land\)-operation are shared (DAG). First the children are replaced by a fresh variable which is restricted by setting it equal to the original operation. This results in \((tmp_0 \land \neg tmp_0) \land (tmp_0 == a \lor b)\). Then this step is repeated for the next operation of the original formula, which leaves the formula like this: \(tmp_1 \land (tmp_1 == tmp_0 \land \neg tmp_0) \land (tmp_0 == a \lor b)\). Note that \(tmp_1\) representing the result of the original formula is added as a clause, otherwise the produced formula is always true as we replaced everything with tautologies. Finally, these equivalences are replaced with the right side of the provided table resulting in a CNF with at most 3 times more clauses than operations were present in the initial formula.

5. **Solving of formula using an a SAT solver**
Finally, the CNF is solved using any SAT solver, such as MiniSAT\cite{23}.

4.1.1 A4Prover

The used bit-precise theorem prover is written in C++ and uses MiniSAT\cite{23} for the actual proving. It supports all integer operators known from C, such as addition, division, arithmetical right shifting, bitwise operators, etc. The internal structure is roughly the way as described in section 4.1. Given an arbitrary boolean expression, possibly containing bit vector expressions, it searches for a satisfying assignment which can then be reported to the user of the library. If the original formula was a DAG (directed, acyclic graph) with shared nodes, this sharing is sustained throughout the whole process of preprocessing this formula for MiniSAT, allowing efficient translation to a CNF using Tseitin tautologies\cite{22} and avoiding exponential blowup.

Note that boolean \(\neg\)-operations can be translated to CNF at zero cost: When translating a given formula to a CNF using Tseitin, a single variable represents the result for a given boolean subformula. If a \(\neg\)-operation is applied to this subtree, the translation process may tag the \(\neg\)-operation as returning the very same variable, but inverted. Then, when the result of this \(\neg\)-operation is used, say in a \(\land\)-operation, the translation process inverts all uses of this variable in the resulting clauses, thereby generating not a single additional clause or variable for \(\neg\)-operations (see section 4.1 for Tseitin tautologies of \(\land\)-operations).
4.2 Calysto

In [1] a software for bit-precise static checking of assertions called Calysto is presented. Similar to model checkers such as CBMC [24] for C or SPIN [25] for Promela, it allows checking of arbitrary properties for C programs. However, its philosophy is the same as used by lint [26]: Precision and soundness are sacrificed for scalability, the tool “doesn’t promise to find all bugs, nor does it promise that all bugs reported are real bugs”.

The underlying approach, further described in [27], is based on an abstraction-checking-refinement loop terminating when a certain assertion can either be proven or disproven or no more refinements are possible. This is not a novel approach and has been used often in model checking. What’s new is the assumption that programmers don’t set function boundaries at random, but instead use functions as abstractions as well. Based on this assumption, Calysto initially introduces unbound variables, so-called summary nodes, to represent the effects of a function call. Only if an assertion can neither be proven nor disproven, such summary nodes are replaced with a more precise abstraction of the called function’s internals.

Simplifications. To increase scalability Calysto makes a few unsound simplifications:

- Loops are unrolled a fixed amount of times, by default exactly once.
- Recursion is not handled at all, i.e. all edges in the callgraph to functions already on the callstack are simply ignored.
- Pointer arithmetic is not supported.
- Non-constant array indices are unsoundly replaced with constants.
- Pointers passed as function arguments are assumed not to alias.

Problems. Unfortunately, it was not possible to get a hold of the original implementation, as the author intends to base a new tool on the ideas used in Calysto. Furthermore, the papers describing Calysto ([27], [1]) leave several questions unanswered or unclear: First, the examples in these papers show some evidence of alias analysis, but the exact handling is not described. However, the earlier mentioned simplification concerning aliases suggests that aliases aren’t determined using a sound method, as these results could also be used to determine aliasing of function arguments. Second, the used method for replacing non-constant array indices with constants is not described at all. Third, [1] is unclear on how elements of arrays are handled. At one point it says that \( \ast ptr \) and \( \ast (ptr + i) \) are assumed to refer to the same object, suggesting that they are
handled unsound by not approximating all elements using a single element but by approximating arrays as having only one element which might get overridden occasionally, while later an informal section about alternatives suggests a proper handling of arrays ("lumping the rest of the elements together"). Finally, fourth, it is unclear whether recursion is handled sound (e.g. by ignoring the recursive call but leaving its effects unbound), or using some unsound method (e.g. by ignoring the call completely, assuming the called function has no effects).

Especially the handling of aliases has some major implications: If aliases are handled in a sound fashion, besides the mentioned exception, a load from memory could be affected by any stores to memory reaching this load, if the pointer operands MAY alias. This means all stores on all (interprocedural) paths leading to a given load must be considered. If a store’s pointer operand MUST alias with the load’s pointer operand, all further preceding stores on this path cannot reach the load anymore, but are blocked off by this store. Another sound approach would be to leave the results of all loads unbound, which is obviously very imprecise. Finally, since Calysto is unsound already, there may also be unsound solutions to consider, providing more accurate results in the average case.

4.3 AACalysto

AACalysto, my reimplementation of Calysto, uses informations provided by AAL / AALSSA (see section 3) to handle aliases precisely and sound. Recursions are handled by leaving their effects unbound (sound), while loads and stores are handled as described: Upon refinement, all possible paths reaching a load are scanned in reverse and all stores, whose pointer operand might alias with the pointer used for the load, are considered as a possible definition for the result of the load operation. As aliasing of indexed accesses is handled by AAL as well, the only unsound simplification left is the constant unrolling of loops.

4.3.1 Reduction to bit vector logic

In imperative languages sequences of instructions and conditional jumps are used to represent behavior. Mapping such a sequential program to (non-sequential) bit vector logic is usually done using one of two approaches: Either forward, using symbolic execution[28], or backwards, using weakest (liberal) precondition[29]. Calysto uses forward symbolic execution, and so does AACalysto, therefore I’d like to give a small example of how this reduction to bit vector logic is done.

Assume the code fragment shown on the left in Figure 30 and its CFG on the right. This code should be prepared so the assertion on line 7 can be proven. Since Calysto unrolls loops all CFGs are DAGs, making processing quite easy:

\[\text{Remember that all loops are unrolled and recursion is simply ignored.}\]
4.4 Comparison

As Calysto is not available, a direct comparison to AACalysto is impossible, and a comparison via the published stats is rendered meaningless because only a special
void test() {
    int a, b;

    argsDontAlias(&a, &b);
    argsDoAlias(&a, &a);
}

void argsDontAlias(int *a, int *b) {
    *a = 42;
    *b = 0;
    assert(*a == 42);
}

void argsDoAlias(int *a, int *b) {
    *a = 42;
    *b = 0;
    assert(*a == 42);
}

Figure 31: Aliasing of function arguments

Because of these circumstances I present here just two cases where the behavior of Calysto and AACalysto will obviously differ.

Figure 31 defines the two functions argsDontAlias() and argsDoAlias(), both with the same body. As mentioned in section 4.2, Calysto assumes that pointers passed as function arguments never alias, and due to this assumption, Calysto is able to prove the assertion in both functions. Sadly, this is wrong in the case of argsDoAlias() where the store *b = 0 overrides the earlier store *a = 42. AACalysto on the other hand bases its approximations of loads and stores on the results provided by AAL (see section 3) and assumes that in function argsDoAlias() *b MAY alias with *a and is therefore only able to prove the assertion in argsDontAlias().

Another simplification done by Calysto is the replacement of dynamic array indices with constants. In the listing in Figure 32 the user is asked for two indices a and b, rendering these variables statically unknown. If Calysto replaces both a and b with the same constant, it is able to prove the first assertion, although this is obviously wrong. If however both variables are replaced by different constants, Calysto is able to prove the second assertion which is just as wrong as the first

The only type of assertions checked by Calysto was that the base pointers of dereferencings were unequal to NULL, but AACalysto leaves this kind of assertion for proving to AAL, which is not path-sensitive.
4.5 Conclusion

This section presented AACalysto, a tool based on the ideas of Calysto. Similar to Calysto, it allows bit-precise, path- and context-sensitive static checking of arbitrary assertions. However, while Calysto contained multiple unsound simplifications, AACalysto reduced those to just one, by using the results of the alias analysis library presented in section 3. Due to the remaining simplification, it cannot claim completeness or correctness of the given results, reducing it to a bug-hunting tool, but as multiple unsound simplifications of the original Calysto have been discharged, its results are assumed to be more precise.

Figure 32: Replacement of array indices with constants

```c
void test() {
    int a, b;
    int arr[2] = {0, 0};
    scanf("%d", &a);
    arr[a % 2] = 42;
    scanf("%d", &b);
    assert(arr[b % 2] == 42);
    assert(arr[b % 2] == 0);
}
```
case. Similar to before, AACalysto is unable to prove either assertion. This constitutes a more precise result.

4.5 Conclusion

This section presented AACalysto, a tool based on the ideas of Calysto. Similar to Calysto, it allows bit-precise, path- and context-sensitive static checking of arbitrary assertions. However, while Calysto contained multiple unsound simplifications, AACalysto reduced those to just one, by using the results of the alias analysis library presented in section 3. Due to the remaining simplification, it cannot claim completeness or correctness of the given results, reducing it to a bug-hunting tool, but as multiple unsound simplifications of the original Calysto have been discharged, its results are assumed to be more precise.

```c
void test() {
    int a, b;
    int arr[2] = {0, 0};
    scanf("%d", &a);
    arr[a % 2] = 42;
    scanf("%d", &b);
    assert(arr[b % 2] == 42);
    assert(arr[b % 2] == 0);
}
```
5 Conclusion

5.1 Summary

In section 3 the general framework AAL to perform alias analysis has been developed and presented. It uses an intermediate representation called AALSSA, which is an abstraction of the actual program, supporting only instructions of possible importance to alias analysis. This small set of only 8 different types of instructions reduces the possibility for erroneous usage of the library and keeps the burden low when extending the library. The decoupling of context-sensitivity from the actual analysis using function instances allows analyses to be implemented completely context-insensitive and then combined at runtime with an arbitrary callgraph-constructor, thereby producing context-insensitive or context-sensitive results, based on the chosen callgraph-constructor (see section 3.3.2). Furthermore, in section 5.5.5 a least-square-based approach to determine the most influential characteristics of an input (e.g. lines of code or number of functions) on different types of performance measurements (e.g. resulting precision or required processing time) has been presented. This has then been used to substantiate some claims about the experimental results produced by AAL.

In section 4 AAL was applied to a static analyzer based on the ideas of Calysto[1], allowing to discharge multiple unsound simplifications of Calysto. The resulting tool, AACalysto, allows bit-precise, context-, and path-sensitive checking of arbitrary assertions, however, since it still contains a single unsound simplification, it can claim neither completeness nor correctness for the produced results.

5.2 Results

With AAL this thesis presents a new tool, giving especially smaller projects the opportunity to incorporate alias analysis without the need to implement the analyses themselves, but still at a competitive performance, as the experimental results have shown. Especially the flow-insensitive analysis has been shown to perform exceptionally well. It must be assumed that this is a direct consequence from the design decision that the intermediate representation, AALSSA, should be SSA-based, making all use-def chains explicit (see section 3.1). Furthermore, external functions often require safe and highly imprecise approximations (e.g. \(*T = T\) ). As SSA-variables cannot be changed by store-instructions, these approximations do not (directly) affect the resulting approximation for most SSA-variables. Only the results of load-instructions are directly affected and may then in turn affect other SSA-variables. This increases precision for both flow-insensitive and -sensitive analyses compared to non-SSA-based algorithms which might discard all points-to sets when an imprecise approximation of external functions as given above is called. On the other hand, the current implementation of a
flow-sensitive analysis costs a lot more resources but gives only a small increase in precision. A possible cause might be absence of MUST alias information, making it impossible to calculate precise kill information.

This thesis and all code of the work presented will be made publicly available as soon as possible. This concerns AAL, the alias analysis library (section 3), AAProver, the bit vector theorem prover (section 4.1.1), and AACalysto, the bit-precise static analyzer (section 4.3). Also, the possibility of integrating AAL into LLVM is currently in discussion.

5.3 Future work

While working on AAL a simple idea for a special type of flow-sensitive analysis came up: If the analyzed code is in SSA form, such as AALSSA, a flow-insensitive algorithm behaves almost the same as a flow-sensitive algorithm, the only instructions affected by flow-sensitivity are memory accesses. Based on this insight it should be possible to extend Andersen’s algorithm (see section 2.5) to calculate a fully flow-sensitive approximation: The basic rules for Andersen’s algorithm use a ∀-operation whenever a dereferencing is involved. By extending those with a filter based on reachability (or rather one of its approximations, such as simple CFG connectivity), it might be possible to get fully flow-sensitive results.
References


A AAL Manual

This manual explains the usage of AAL in more detail and therefore assumes that the reader has read sections 2 and 3 as well.

The code base of AAL may be divided into three parts:

- **Intermediate representation**
  This part consists of a single header file, `ast.h`, containing all definitions required to construct and query the (abstract) intermediate representation AALSSA (see section 3.2). This part is obviously of interest to both, a user of AAL, as well as an engineer implementing new analyses for AAL.

- **Analyzer framework and implementations**
  This part’s headers reside in the folder ‘an’. They define the base classes for analyzers and callgraph-constructors (see section 3.3.2), the class for points-to sets (see section 2.4), and contain the definition for the already implemented callgraph-constructors as defined in section 3.3.2. A user of AAL is only interested in a very small part of this part.

- **Miscellaneous**
  In the folder ‘util’ some general purpose utilities reside. These include: Tools to load and store an intermediate representation, to pretty-print AALSSSA, to extract and abstract static callgraph information, to generate stats of the given intermediate representation, and to query the usage of resources. This part is of interest to both, a user of AAL, as well as an engineer extending AAL.

Instead of explaining the first two parts on their own, this documentation will show how they are used, once from a user’s perspective, then from the engineer’s perspective. Only the third part, which is of no importance to perform an actual analysis, and is of equal interest to both the user and the engineer, is explained separately.

A.1 A user’s perspective

Usage of the library is very simple and may be broken down into 5 steps:

1. **Produce an intermediate representation**
   The header file `ast.h` contains all definitions required for the intermediate representation AALSSA\(^{20}\). As a first step, the user creates an instance of the class `AliasAnalysis`, which is the root for an intermediate representation. Then, the user may add functions by creating instances of the

\(^{20}\)All code for AAL is in the namespace ‘aal’.
AAL MANUAL

A.1 A user’s perspective

class *Function*, which may again be populated with instances of the class *BasicBlock*. Finally, the instructions described in section 3.2 may be added to these basic blocks by instantiating the corresponding classes. All classes contain member functions with mostly self-explaining names to query the intermediate representation. Once the user defined a function and connected the basic blocks, one of these blocks (without any predecessors) must be set as entry block for this function, otherwise the representation is invalid and a later analysis will fail. Calling `AliasAnalysis::getGlobals()` returns a special basic block, which must only contain `alloc`-instructions and may be used to define globals. Finally, `AliasAnalysis::getRootFunction()` returns the predefined root function (see section 3.2) with an empty body at first. The user must then add a body appropriately modelling the behavior of the environment, such as adding a *call*-instruction calling the *main()* function. AALSSA does not support external functions, instead the user must generate approximations for them.

2. Setup a callgraph-constructor

Once the intermediate representation of the code under consideration has been prepared, the user must instantiate a subclass of *CallgraphConstructor*. Currently, the 3 different types of callgraph-constructors as described in section 3.3.2 are available. In addition, the user may provide an instance of class *CGCFilter* upon instantiation of any callgraph-constructors. These may be used to limit the set of possible call targets for a given call site: Since AALSSA does not conserve type informations it might sometimes consider a function as possible callee for some (dynamic) call site, although the original language does not allow such a call (e.g. Java) or renders it undefined (C/C++) due to incompatible types. However, if the user supplies an instance of *CGCFilter* to the callgraph-constructor, this object receives a callback whenever a function is considered as possible callee for some call site, allowing it to veto this function as a possible callee for this call site. As an example, the frontend used in section 3.5 supplied such a filter, and whenever a function was considered as a possible callee for some dynamic call site, it was vetoed unless the function pointer was ever used as something else besides being the target of static call sites. This is basically the same as the address-taken algorithm presented in section 2.5.

3. Setup an analyzer

For the currently implemented analyses, flow-insensitive or -sensitive, this means simply instantiating the according class, passing it the previously constructed intermediate representation and callgraph-constructor.

4. Perform the analysis

Then, a simple call to `Analyzer::analyze()` triggers the analysis. Through
setupReportUsage() the user may request a log of memory and cpu usage over time.

5. Query results
Once the analysis is completed, the user may query the function instances that have been created using the functions Analyzer :: getNumFunctionInstances() and Analyzer :: getFunctionInstance(). Depending on the type of callgraph-constructor used, multiple function instances for the same function might have been created. Using the function Analyzer :: getTargets() the user can then query the set of possible targets for a given SSA-variable in a given function instance. If NULL is returned the SSA-variable in question is ⊤, otherwise an immutable set of possible targets is returned. The user may then check two such sets for an intersection and thereby a MAY-alias using the function TargetSet :: doesIntersect(). If the set of targets has exactly one element, this element is a MUST-alias. It may even happen that a set of targets has no members at all, this usually happens if the code under consideration contains dead (unreachable) code.

A.2 An engineer’s perspective

Target sets. The class TargetSet is used to represent the right-hand-side of the points-to relation, i.e. the set of targets for some pointer. To reduce the memory footprint it is desirable to be able to share right-hand-sides, especially in the case of flow-sensitive analyses, where usually only a comparably small set of pointers changes from one program point to the next program point. To achieve this goal all instances of TargetSet are modelled as immutable and contain a reference counter to determine when they should be freed. The class TargetSetFactory provides methods to create new TargetSets by (e.g.) merging to TargetSets. As correct usage of the reference counter is somewhat error-prone, the class PointsToSet provides a complete implementation for the points-to relation, providing the engineer with a simple interface for mutations. However, the engineer must not completely forget about the reference counter: When looping over the elements of a TargetSet and performing operations that might possibly decrease the reference counter of this very TargetSet, the engineer must pay attention that the reference counter does not reach zero, thereby freeing the TargetSet. Simply increasing the reference counter before the loop and decreasing it afterwards solves this problem.

Callgraph constructors. Implementing a new callgraph-constructor is pretty simple and usually done with less than 100 lines of code. The new class must be a subclass of CallgraphConstructor and implement the virtual function addTarget().
In this function, given some call site instance and a function determined to be called from this call site instance, the callgraph-constructor must determine the actual function instance being called, possibly creating a new one using `Analyzer::createInstance()`, and add it to the list of possible callees using `CallSiteInstance::addTarget()`. Note that `CallgraphConstructor::addTarget()` is called if and only if the proposed callee hasn’t been vetoed by a `CGCFilter` (see section A.1).

**Analyzers.** To implement a new analysis, the engineer must subclass the class `Analyzer` and, most importantly, implement the virtual method `analyze()`. There, an arbitrary context-insensitive algorithm may be implemented to calculate approximative results. First, the analysis must artificially create a function instance for the `rootFunction`, then start its processing from there. Once a call is reached, e.g. a call to `main()`, `CallgraphConstructor::updateCSI()` must be used to update the list of possible call targets for this call site instance. Handling of all other instructions is completely up to the algorithm to be implemented. To allow querying by the user the virtual method `getTargets()` must be implemented, returning the set of possible targets for some SSA-variable and some function instance, or `NULL` if the set is determined to be \( \top \). Furthermore, the class `Analyzer` provides some protected helper methods for tasks most analyses need to perform.

### A.3 Miscellaneous

#### Serialization and deserialization.

Two simple functions `serialize()` and `deserialize()` may be used to perform the according action on a given intermediate representation (instance of class `AliasAnalysis`). These functions are declared in `util/loadstore.h` and are based on Google’s Protocol Buffers.

#### Pretty-printing.

In `util/printvisitor.h`, class `PrintVisitor` is defined, which may be used to pretty-print most parts of the intermediate representation. Furthermore, by subclassing `PrintVisitor::IAnnotator` certain instructions can be annotated. In `util/printannotated.h` the class `PrintAnnotated` is defined: Once an intermediate representation has been analyzed, this class may be used to pretty-print the code, where all call sites are annotated with a list of possible callees, and all `load`- and `store`-instructions are annotated whether they `MAY` dereference `NULL`.

---

21 Remember section A.3.2: While a call site is a specific call instruction in a function, a call site instance is a specific call instruction in a function instance.

Callgraph extraction and abstraction. In `util/staticcallgraph.h` the class \textit{Callgraph} declares a number of static member functions allowing the extraction, abstraction, and storage of callgraphs. Once the initial static callgraph has been extracted using \textit{extractStaticCG()}, multiple abstractions can be applied, organizing multiple nodes of the initial graph as a group of nodes.

- \textit{extractHierarchy()}: Puts all members of a strongly connected component (SCC) into a single group per SCC, thereby turning the graph into a DAG (directed, acyclic graph).
- \textit{extractTrees()}: Puts all members of a tree into a single group per tree.
- \textit{extractPartialTrees()}: Puts all children, which are the root of a tree, of a parent, which is not a tree, into a single group per parent.
- \textit{simplify()}: All groups containing only one element are eliminated and replaced by the element itself.

Finally, \textit{genDOT()} may be used to dump the generated, possibly hierarchical callgraph as a dot-file.

Statistics. The class \textit{Stats} defined in `util/stats.h` may be used to calculate some simple statistics for an intermediate representation, such as counting the number of functions or instructions. The class \textit{AnalyzerStats} defined in `util/analyzerstats.h` may be used only once some analyzer has been used on an intermediate representation and calculates advanced statistics, such as the average number of callees for dynamic call sites or the percentage of loads and stores whose base pointer was proven unequal to \textit{NULL}.

Resources usage. In `util/resources.h`, the function \textit{memUsage()} is declared, returning the current memory footprint in bytes, as reported by the operating system. Also, the function \textit{cpuUsage()} is declared, returning the amount of used cpu time up until now in microseconds.
B AAProver Manual

This manual explains the usage of the bit-precise theorem prover AAProver in more detail, based on the given overview in section 4.1.

B.1 Representation of formulas

Formulas are represented as a DAG (directed, acyclic graph) of nodes, allowing sharing of nodes between multiple parents. There are two basic types of nodes: Those of type \textit{BoolFormula}, representing a boolean formula, and those of type \textit{BVTerm}, representing a vector of bits. Figure \ref{fig:allNodeTypes} shows the hierarchy of all different node types\footnote{All code for AAProver is in the namespace 'aaprove'.} with their meaning shortly described here:

- \textit{BoolFormula}: This is the base class for all boolean formulas.
- \textit{BoolConst}: This class may be used to represent the boolean constants \top (true) or \bot (false).
- \textit{BoolVariable}: An instance of this class represents a boolean variable.
- \textit{BoolUnOp}: This node type is used to represent an unary boolean operation. Only $\neg$ (not) is supported.
- \textit{BoolBinOP}: This node type is used to represent a binary boolean operation. It supports $\lor$ (or) and $\land$ (and) operations.
- \textit{BoolRelation}: This class connects boolean formulas and bit vector expressions. An instance of this class may be used to compare two bit vectors with each other. Supported operations are: $=$ (eq), $\neq$ (neq), $>$ (gtr), $\geq$ (geq), $<$ (lss), and $\leq$ (leq).
- \textit{BVTerm}: This is the base class for all bit vector expressions. It has a simple type attached, defining the number of bits. Note that the type does not define whether the bit vector is signed or unsigned.
- \textit{BVConst}: This class may be used to represent a constant bit vector.
- \textit{BVVariable}: An instance of this class represents a bit vector variable with a size defined by the type attached to it.
- \textit{BVUnOp}: This node type is used to represent an unary bit vector operation. Supported operations are: $\sim$ (bitwise not), $-$ (negation), and a signed and an unsigned cast. The cast operations may be used to change the size of a bit vector, either truncating it or extending it. If a signed cast is used sign extension is performed if the number of bits is increased.

\footnote{All code for AAProver is in the namespace 'aaprove'.}
B.2 Solving of formulas

Solving of a given formula is fairly simple:

1. **Type checking.** First, the function `typecheck()` must be used to check the types of all bit vector expressions contained in a boolean formula. The check is mandatory as it also annotates all intermediate nodes (operations) with the correct typing information, which is needed by later passes.
2. **Translating bit vector logic to boolean formulas.** Using an instance of the class `GenPL`, all bit vector expressions within a boolean formula may be translated to boolean formulas as well.

3. **Elimination of boolean constants.** The formula produced by the previous step might still contain some boolean constants. These were either defined as boolean constants in the initial formula, or were introduced by the previous step when translating bit vector constants to boolean formulas. The function `eliminateBooleanConstants()` either returns an equivalent formula containing no boolean constants, or a single boolean constant.

4. **Check for trivial result.** The previous step might reduce the given formula to a simple \( \top \), denoting the initial formula to be a tautology, or \( \bot \), denoting the initial formula to be a contradiction. If this is the case, the formula is solved and processing may stop here. Note that not all tautologies and contradictions will be caught by this step.

5. **Solving.** Finally, an instance of `MiniSATSolver` may be used to translate the boolean formula to CNF using Tseitin tautologies and solving it using MiniSAT. If a solution is found, the values for all boolean and bit vector variables can be queried.
Acknowledgements

Special thanks to go my advisor Dr. Thomas Wahl for giving me the opportunity to write this thesis, and for many helpful hints and advice he gave me throughout the practical part as well as while working on the written part. Whenever I got stuck during the implementation phase, it was Christoph M. Wintersteiger who gave me another perspective of the problem at hand during hours of discussions. Furthermore, I would like to thank Prof. Daniel Kröning for the opportunity to write this thesis in his group and the possibility of giving an interim presentation. Finally, I’d like to thank the LLVM development group for such a great development basis.