A Hybrid Approach for Parametric Memory Dependence Analysis

Tina Jung

Bachelor's Thesis

Compiler Design Lab
Faculty of Natural Sciences and Technology I
Department of Computer Science
Saarland University

Supervisor:
Prof. Dr. Sebastian Hack

Advisors:
Johannes Doerfert
Kevin Streit

Reviewers:
Prof. Dr. Sebastian Hack
Prof. Dr. Dr. h.c. Reinhard Wilhelm

Submitted: December 09th, 2015

Declaration of Authorship

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Datum/Date: ______________________________

Unterschrift/Signature: ______________________________

SAARLAND UNIVERSITY

Abstract

Compiler Design Lab
Department of Computer Science

Bachelor of Science

by Tina Jung

The detection of memory dependencies is challenging for many aggressive optimizations, such as automatic parallelization. Two problems automatic parallelization faces are alias resolution and loop carried memory dependence detection. A relational analysis is necessary to statically analyze aliasing. Different iterations of the same loop need to be compared to prove the absence of loop carried memory dependencies. Even most flow sensitive analyses look at statements, but not statement instances. Thus, loop iterations are indistinguishable and different states of the loop are not taken into account. We precisely describe statement instances by utilizing the polyhedral model. This model is commonly used to analyze and optimize affine loop nests. Our approach extends this usage to whole program regions, overapproximating non-affine expressions. Overall, we present a parametric context and flow sensitive inter-procedural analysis, which gives tight descriptions of memory accesses for arbitrary granularities of program regions.

Acknowledgments

I would like to thank Professor Sebastian Hack for catching my interest in this area of computer science and giving me the opportunity to write this thesis. I am also grateful that Professor Reinhard Wilhelm reviews this work.

Furthermore, my special thanks go to Johannes Doerfert and Kevin Streit for supporting me with all theoretical and technical difficulties of the thesis. Various discussions helped me to gain a deeper understanding of the theoretical foundations of this work. The critical feedback on my writing was very helpful to improve on it. I also want to thank Maximilian Fickert for proofreading the thesis.

Finally, my thanks go to my family and friends for supporting me all along my studies. They had a good sense to distract me or be patient, whichever was appropriate at that time.

Contents

Declaration of Authorship iii

Abstract v

Acknowledgments vii

1 Introduction 1

1.1 Desired Properties of Memory Access Descriptions 1

1.2 Overview 3

2 Background 5

2.1 Polytopes 5

2.2 LLVM 6

2.3 Scalar Evolution 8

2.4 Integer Set Library 9

3 Related Work 11

4 Analysis 13

4.1 Parametric Access and Value Description 14

4.2 Language Definition 15

4.3 Intra-procedural Analysis 16

4.3.1 Value Analysis 16

4.3.2 Local Access Descriptions 20

4.3.3 Propagate Access Descriptions 21

4.3.4 Overapproximation and Intra-procedural Widening 25

4.4 Inter-procedural Analysis 26

4.4.1 Application of Inter-procedural Analysis 27

4.4.2 Fixpoint Iteration 29

4.4.3 Widening 30

4.5 Memory Dependence Analysis 32
4.5.1 Loop Carried Dependencies 32

5 Limitations **37**
5.1 Non-affine Expressions 37

6 Evaluation **39**
6.1 Enabling Parallelization 39
6.1.1 Aliasing 39
6.1.2 Recursion 41
6.2 Benchmarks 42

7 Technical Details **45**
7.1 Condition Propagation on the CFG 45
7.2 Further Remarks 47

8 Conclusion and Future Work **49**
8.1 Future Work 49
8.2 Conclusion 50

Bibliography **53**

Chapter 1

Introduction

Memory dependence analysis is used for a wide range of applications such as alias resolution or vectorization. Many of the available analyses and optimizations are combined to automatically parallelize programs. Parallel program execution is necessary to effectively utilize multi-core systems, but many existing applications are not designed for multi-core execution and can therefore not make use of the given hardware. Writing parallel programs is generally harder than writing sequential ones, which makes it desirable to have the compiler taking care of parallelization.

This work focuses on generating descriptions for memory accesses of arbitrary program regions, which can be used to support an automatic parallelizer. The parallelizer faces several challenges regarding memory dependencies such as loop carried memory dependencies and aliasing. We will discuss the challenges and present solutions to them in terms of a static analysis with ideas for run-time extensions.

1.1 Desired Properties of Memory Access Descriptions

In order to obtain memory access descriptions fitting the needs of a parallelizer, we first look at the challenges caused by memory dependencies in automatic parallelization.

```
1 for (i := 0; i < 10; i := i + 2) {
2     A[i] := A[i + 1];
3 }
```

Figure 1.1: Loop with Static Bound

Each iteration of the loop shown in Figure 1.1 can be executed in parallel and additionally, all the information is already statically available. The write accesses are made to odd positions in the array while the read accesses are at even offsets. The result of a simple interval analysis for this example could look like this:

$$interval(i) = [0,8]$$
$$interval(i+1) = [1,9]$$

In order to check whether the accesses $A[i]$ and $A[i+1]$ overlap, the values for the index positions are intersected.

$$interval(i) \cap interval(i+1) = [0,8] \cap [1,9] = [1,8]$$

Since the resulting interval is not empty, we have to assume that there may be dependencies.

This example already shows problems with the granularity of a simple interval analysis. The results for i are typically widened to have a sound description for all possible values of i. No difference is made between the effect of one particular iteration and the accumulated effect of the entire loop. The values of i and $i+1$ cannot interfere within the same iteration as they refer to an integer and its successor.

Not only the memory dependencies within the same loop iteration but also those across different loop iterations are interesting. Therefore the widening of the interval analysis, which cannot take the increment of 2 into account, is too overapproximative to give a useful solution for this purpose. In order to enable the parallel execution of statements inside a loop, overlapping memory accesses within the same iteration and within different iterations must be identified.

```
1 for (i := 0; i < N; ++i) {
2     A[i] := A[i + C];
3 }
```

FIGURE 1.2: Parametric Loop Carried Dependence

The example in Figure 1.2 shows a loop for which it is not statically decidable whether there are memory dependencies. It depends on the values of N and C. The loop can be vectorized with some vector width vw, if $vw \leq C$ holds. The loop can be completely parallelized if $N \leq C$ holds. Both conditions are candidates for run-time checks that can be generated and placed in front of the loop to select either the parallelized version or the original sequential version.

```
1 for (i := 0; i < N; ++i) {
2     for (j := 5; j < M; j := j + 2) {
3         A[i] := A[i] + B[j];
4     }
5 }
```

FIGURE 1.3: Aliasing

Since there can be aliasing in C-like languages, the two arrays A and B in Figure 1.3 may refer to the same memory region. A run-time check can resolve this problem: if $A+i$ and $B+j$ are disjoint memory regions for every i and j, the accesses do not overlap. The same holds if A and B do not alias. In these cases the outer loop can be parallelized.

Not only loops, but arbitrary program regions or instructions are interesting for memory dependence analysis. Consider the function calls $f(A,x)$ and $g(A,x)$ with an array A and a scalar variable x and the following definitions for the functions:

```
1  func f(B[], b) {
2      for (i := 0; i < b; ++i) {
3          B[i] := 0;
4      }
5  }
6
7  func g(C[], c) {
8      C[c] := 1;
9  }
```

FIGURE 1.4: Potentially Parallelizable Functions

The called functions need to be analyzed to determine whether the function calls can access the same memory regions. The function f accesses the given array in the range from 0 to $b-1$, and the function g accesses the array at position c. Whether memory accesses of functions overlap often depends on the parameters of the functions. Summarized access descriptions that are parametric in the function parameters can be used to conclude that $f(A,x)$ and $g(A,x)$ cannot overlap.

Our analysis should be capable of aliasing, have a parametric description of accesses and also be inter-procedural to provide summarized descriptions for functions. In order to support loop parallelization it should also be capable of the concept of loop iterations. To achieve this, we designed an inter-procedural context and flow sensitive analysis, which works with polyhedral descriptions for function accesses.

1.2 Overview

Chapter 2 will present background information on the theoretical concepts and the technologies for the implementation. A comparison to similar approaches is provided in Chapter 3. The memory access descriptions and the analysis to generate these descriptions is presented in Chapter 4. The limitations of this approach are discussed afterwards, in Chapter 5. Experimental results for the analysis are discussed in Chapter 6 together with possible run-time extensions. Chapter 7 describes technical details

of the implementation. Finally, Chapter 8 concludes this thesis and provides ideas for future research on this topic.

Chapter 2

Background

This section gives an overview of the theoretical and technical background of the thesis. We use a polyhedral description of the memory accesses, which are represented by objects of the Integer Set Library (ISL [1]). The analysis works on LLVM-IR and uses the Scalar Evolution Analysis [2] in combination with techniques from Polly [3] to interpret the instructions and generate the polyhedral description.

2.1 Polytopes

To describe accesses, we collect affine constraints on program variables. A system of these constraints can be interpreted as a geometric structure, a convex n-dimensional polytope, where n is the number of variables. Figure 2.1 shows the 2-dimensional polytope for the constraint system $\{x \geq 2,\ i \leq 6,\ x + i \geq 4\}$.

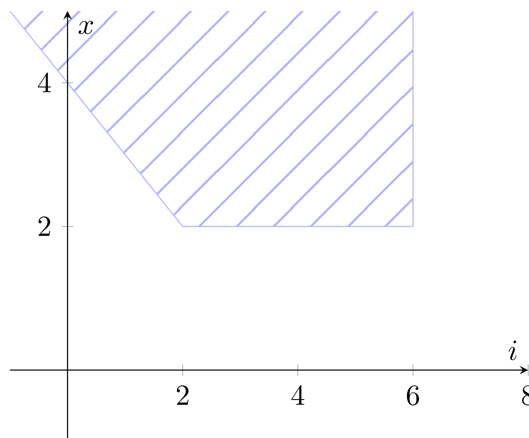


Figure 2.1: 2-Polytope with Dimensions x and i

We chose polytopes for our analysis because they are relational, thus allowing us to describe values of a variable depending on values of another variable. Convex n-polytopes are part of Presburger arithmetic [4]. This arithmetic is a first order logic based on

addition. It is decidable, complete and consistent. The logic does not formulate multiplication or division, but some properties that are related to those operations can still be expressed. Presburger arithmetic includes existential and universal quantification, therefore the property that a variable x is divisible by two can be expressed as $\exists y.\ y + y = x$.

Note that convex polytopes only include constrains based on $\leq$ and $\geq$. We will later only consider integer values as the domain of polytopes and use the notation $x < y$, which can be expressed as $x \leq y - 1$ (similarly $x = y$ for $x \leq y \wedge y \leq x$).

2.2 LLVM

LLVM [5] is a compiler framework, which is built for program analysis and transformation. A wide range of different languages such as C, C++, Objective C, Java and Python can be compiled to the low level LLVM Intermediate Representation (LLVM-IR). This is one of the aspects making LLVM an attractive choice to implement the analysis: We are not tied to a high level language and do not need support for high-level language features, but work on a language independent low level representation. Another important advantage of LLVM-IR is that it is in Static Single Assignment (SSA) form [6], which guarantees that every variable is defined exactly once and that the definition dominates all its uses. Phi nodes are introduced to construct SSA form from a non-SSA program. They are selectors of values depending on the edge taken to the phi node.

```
1  func f(x) {
2      a := 5;
3
4      if (x > 0)
5          a := 3;
6      else
7          a := 7;
8
9      return a;
10 }
```

FIGURE 2.2: Simple Program

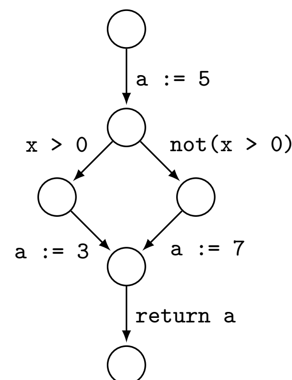


FIGURE 2.3: Non SSA Form CFG of Figure 2.2

The example in Figure 2.2 shows a program and Figure 2.3 its control flow graph (CFG). The SSA form of the control flow graph is shown in Figure 2.4. The introduced phi selects the value for a_3 depending on whether the left-hand side or the right-hand side led to the phi.

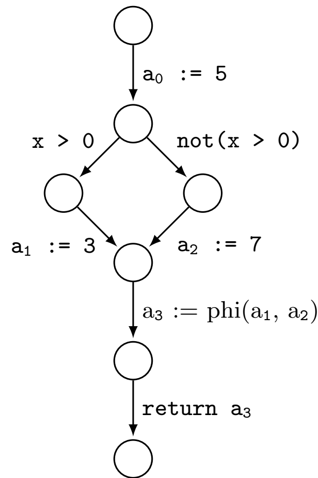


FIGURE 2.4: SSA Form CFG of Figure 2.2

SSA form makes def-use chains explicit and is therefore a helpful property for program analysis.

The instruction set of LLVM-IR contains arithmetic operations, conditional and unconditional branches, phi nodes, cast operations, as well as specialized instructions for calls and pointer arithmetic. The code in Figure 2.5 shows a simple loop, which increments a variable i by 2 in every loop iteration until it reaches the value 10.

```
1 func f(a) {
2     for (i := 0; i < 10; i := i + 2) {
3         a := a + i;
4     }
5     return a;
6 }
```

FIGURE 2.5: Simple Loop

Loops do not have a syntactic representation in LLVM-IR; they are expressed using branches and labels. Instructions are grouped to form basic blocks, each marked with a label. Control flow is represented by conditional and unconditional branches at the end of a basic block. All instructions inside a basic block are unconditionally executed when entering it.

The LLVM intermediate representation for the loop in Figure 2.5 can be seen in Figure 2.6. In the basic block `%for.cond` the values for the uses of a and i inside the loop are selected and the condition is calculated. Depending on the result, either the body of the loop is executed or the loop is left. The loop body calculates the addition `a + i` and branches back to `%for.inc` unconditionally. Here the increment is computed and the unconditional branch to the loop header is taken.

```
define i32 @f(%a) #0 {
entry:
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %a.0 = phi i32 [ %a, %entry ], [ %add, %for.inc ]
  %i.0 = phi i32 [ 0, %entry ], [ %add1, %for.inc ]
  %cmp = icmp slt i32 %i.0, 10
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %add = add nsw i32 %a.0, %i.0
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %add1 = add nsw i32 %i.0, 2
  br label %for.cond

for.end:                                          ; preds = %for.cond
  ret i32 %a.0
}
```

Figure 2.6: LLVM-IR

Loops are not syntactically represented in LLVM, but some helpful analyses for loops, such as the *LoopInfoPass*, are already implemented. This pass provides basic information such as the blocks belonging to the loop and loop nests.

2.3 Scalar Evolution

The Scalar Evolution is a value analysis which aims to find a closed form description for integer or pointer expressions in the program. The closed form expressions extracted by the analysis are represented by so called *SCEVs*. *SCEV Constants* and *SCEV Unknowns* are the smallest SCEVs, a SCEV Constant represents an integer constant (such as 5) and a SCEV Unknown an integer value not known at compile-time. SCEV Unknowns can also occur for undefined values, e.g., a load from memory. All other SCEVs, e.g., the Scalar Evolution equivalent of an addition, the *SCEVAddExp*, are based on them and are recursively defined. Further supported operations are subtraction, multiplication, unsigned division, maximum and cast operations.

The Scalar Evolution [2] is specialized to compute a closed form description of loop variant variables (e.g. induction variables) which they call *Add Recurrences*. Currently only additive recurrences are supported. An example of an Add Recurrence is shown in Figure 2.7.

$$\{\underbrace{0}_{Start}, \underbrace{+}_{Operator}, \underbrace{2}_{Stride}\}\langle\underbrace{\texttt{\%for.cond}}_{Loop}\rangle$$

FIGURE 2.7: Scalar Evolution Add Recurrence

The description of an Add Recurrence contains an initial value for the variable, a stride and the operation performed. They always describe the evolution of a scalar with respect to the surrounding loop, identified by the label of the unique header block. *Start* is a constant in this case, but it can also be a variable. Figure 2.8 shows a loop with an array access.

```
1 for (i := 0; i < 10; i := i + 2) {
2     array[i] := i;
3 }
```

FIGURE 2.8: Loop with Memory Access

In this case the initial value for the Add Recurrence is a pointer variable, so the start is parametric in this variable. The Add Recurrence for this example is shown in Figure 2.9.

$$\{\underbrace{\texttt{@array}}_{Start}, \underbrace{+}_{Operator}, \underbrace{\texttt{2 * sizeof(array[0])}}_{Stride}\}\langle\underbrace{\texttt{\%for.cond}}_{Loop}\rangle$$

FIGURE 2.9: Scalar Evolution Parametric Add Recurrence

The Scalar Evolution can often compute trip counts for loops, which our analysis relies on to widen the results for statements inside the loop. We will go into more detail about this in Section 4.3.4. The analysis is based on the work of Engelen [7, 8] which uses the results of Bachmann et al. [9–11] on chains of recurrences. Further information about the Scalar Evolution can be found in the thesis of Tobias Grosser [12].

2.4 Integer Set Library

The Integer Set Library [1] (ISL) is a C library which provides functionality for program analyses and transformations which rely on the polyhedral model. It provides a framework to represent and manipulate multidimensional affine integer sets and relations. The Add Recurrence from Figure 2.7 can be expressed in ISL as shown in Figure 2.10.

$$\{[i_0] \to [2i_0] \;:\; i_0 \geq 0 \text{ and } i_0 \leq 4\}^1$$

FIGURE 2.10: Affine Relation in ISL

The Add Recurrence represents the values for the program variable i. The ISL description captures all values of i depending on the virtual loop iteration i_0. Due to the upper

[1]Syntax similar to the original ISL syntax. $2i_0$ denotes $2 * i_0$.

bound of 10 and the stride of 2, the loop iterates five times, represented by i_0 ranging from iteration zero to iteration four. For a given iteration number i_0 one can obtain the concrete value for i in this iteration. As an example, in iteration three, the value for i is $2 * 3 = 6$.

There are three kinds of dimensions in ISL: parameter dimensions, input dimensions and output dimensions. An arbitrary amount of dimensions of each kind is possible. Figure 2.11 shows the different types of dimensions in the ISL descriptions.

$$\underbrace{[p_1, p_2]}_{\text{Parameter Dimensions}} \rightarrow \{ \underbrace{[i_0, i_1, i_2]}_{\text{Input Dimensions}} \rightarrow \underbrace{[o_0]}_{\text{Output Dimensions}} : c_0 \texttt{ and } c_1 \}$$

FIGURE 2.11: ISL Dimension Types

The output dimensions describe the values inside the polytope depending on the input and parameter dimensions. The constraints c_0 and c_1 restrict the dimensions.

Chapter 3

Related Work

Memory dependence analysis is a very popular and active area of research. Our work presents a relational, context sensitive inter-procedural analysis, which is able to handle recursion and has a parametric description of the accesses. Currently our analysis is purely static. The following approaches are similar to ours or present interesting ideas on how to extend it.

The work of S. Rus, L. Rauchwerger and J. Hoeflinger [13] about a static and dynamic memory reference analysis addresses the necessity of run-time checks to achieve accurate results. Their approach is similar to ours: they introduce a parametric description of accessed memory regions, so called linear memory access descriptors (LMADs). Additionally, they deal with non-linear access expressions, e.g. for indirect memory accesses. Their analysis works on Fortran '77, hence they do not address problems such as aliasing or recursion, because they do not occur in Fortran. The LMADs are descriptions of memory accesses within loops and can contain conditions, call sites and recurrences. To represent accesses of larger program regions, they collect access information referring to arrays in summary sets. These sets are classified in different categories, distinguishing read and write accesses among others. Using LMADs comes at the cost of redefining operations and optimizations for this description.

The problem of constructing access descriptions for whole program regions is addressed by R. Rugina and M. C. Rinard [14]. Their inter-procedural analysis computes ranges for array accesses using constraint systems for lower and upper bounds. Their goals include finding array bound violations, whereas we assume that there are no out of bounds accesses. Our analysis could be adapted to check array bounds, but this would result in very conservative conclusions about the memory accesses in many cases. To give an accurate description of the accesses within a program region they generate read and write sets. Their analysis is also purely static. The representation of ranges for values of array accesses is strictly weaker than the polyhedral representation when considering

linear-affine bounds, as every such range can be expressed by a polyhedron, but not vice versa.

We provide an intra-procedural value analysis as a first step for the overall analysis, which we want to compare to similar approaches in the following.

P. Cousot and N. Halbwachs developed a value analysis using polyhedra [15]. The general method of handling conditionals and assignments in the paper of Cousot and Halbwachs is similar to our approach; only the handling of loops differs. Cousout and Halbwachs use widening after analyzing the loop once, which is defined as dropping all constraints that are not exactly the same before and after the symbolic execution of the loop. They do not take the loop bounds into account, which is a sound overapproximation. In most cases this is not accurate enough to find precise descriptions of array accesses within the loop. As loop iteration variables are often used for array accesses, the bounds of these variables are an important information. To tackle these problems, we use the Scalar Evolution. It provides information about loop bounds and variant variables, which we use to widen the results of expressions within the loop. This is explained in more detail in Section 4.3.4.

The analysis described by Cousot and Halbwachs works with one polyhedron per program variable describing real numbers. We use a union of polyhedra to avoid the overapproimating join if the union is not a polyhedron again. We constrain our polyhedra to describe integers, as other values are of no particular interest to form the access descriptions.

Halbwachs presents an improved widening operator, *widening up to*, in his work *Delay analysis in synchronous programs* [16]. This widening operator builds invariants about values of variables. It does not skip all constraints about a variable but finds bounds for the variable that changes. This operator is therefore already able to handle simple loop bounds and improves precision of the analysis.

Henzinger and Ho [17] present an extrapolation operator, which is more precise than Cousot and Halbwachs' widening operator, but lacks the guarantee to terminate. Nevertheless, this operator is useful to gain precise information when granting some additional computation time and can be seen as an alternative operator to improve the precision of Cousot and Halbwachs' analysis.

Chapter 4

Analysis

Our analysis consists of multiple steps as shown in Figure 4.1. In the intra-procedural phase, a value analysis generates polyhedral descriptions for program variables. Afterwards accesses which occur in the function are analyzed separately and summarized to a single memory access description for the whole function.

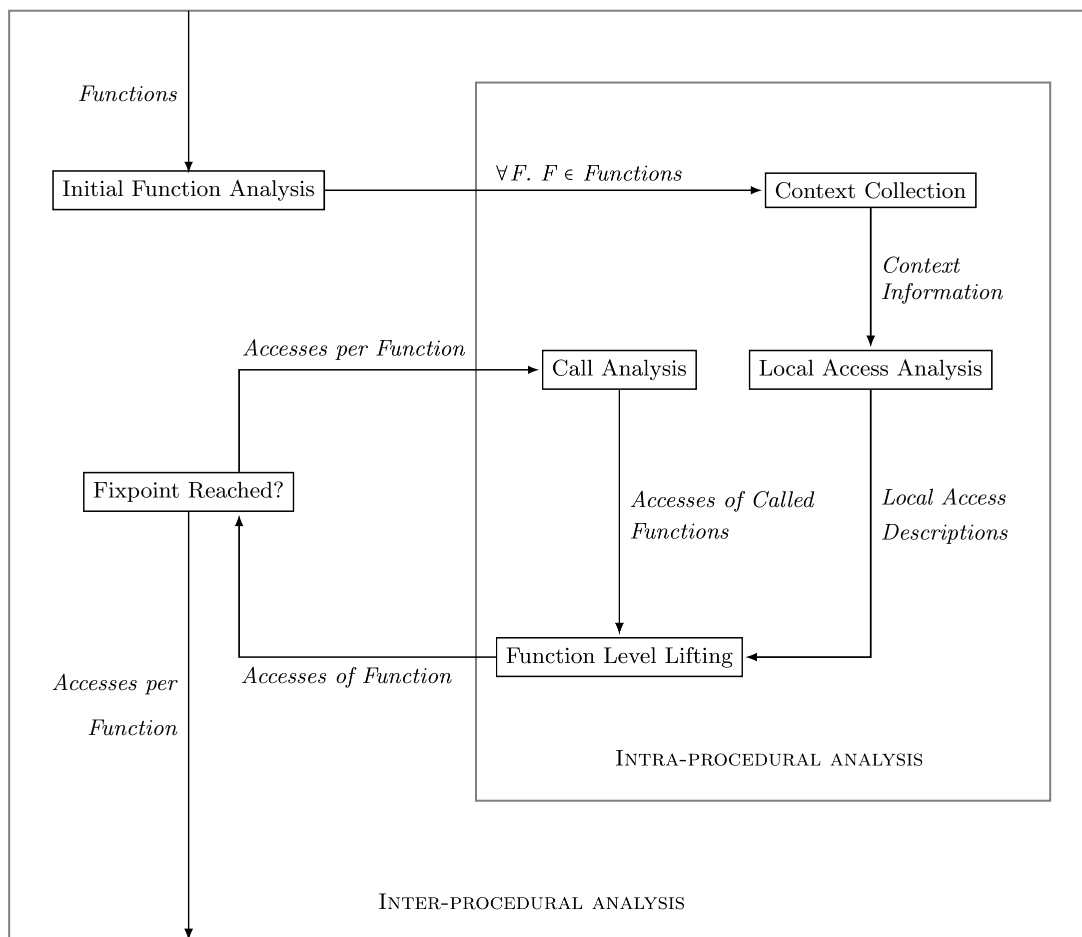


Figure 4.1: Analysis Phases

Function calls are not resolved in the initial intra-procedural analysis, this is done by a fixpoint iteration in the inter-procedural analysis. In this phase each function is again analyzed individually with access descriptions of all functions that are called inside this function. Resolving the calls leads to new access descriptions for each function, which are then propagated to the call sites again. The propagation is done until a fixpoint is reached.

4.1 Parametric Access and Value Description

As motivated in Chapter 1, we want an analysis that generates relational memory access descriptions. We chose a polyhedral representation, as it is parametric and as powerful as Presburger arithmetic. This logic is decidable, consistent and complete, and therefore a solution to a constraint system can always be found if it is solvable. The results of the value analysis as well as the memory access descriptions are polyhedral unions. To explain what will become parameters in our description and how to interpret them, consider Figure 4.2.

```
1 for (i := 0; i < N; ++i) {
2     A[i] := ...;
3 }
```

FIGURE 4.2: Parametric Loop with Memory Access

Our description for the access in line 2 is parametric in the parameters shown in Figure 4.3.

$$\underbrace{[A, A_{end}, N]}_{\text{Parameter Dimensions}} \rightarrow \{ \underbrace{[i_0]}_{\text{Input Dimensions}} \rightarrow \underbrace{[o_0]}_{\text{Output Dimensions}} : ...\}$$

FIGURE 4.3: Parameters for the Access A[i]

A parameter can be an integer variable for which we do not know the exact value, e.g. a function argument. In our example above N is such a parameter. A and A_{end} refer to the location of an array, where A corresponds to the first position of the array and A_{end} to its last position. We do not track memory content, so whenever we refer to arrays we refer to their addresses. The dimension i_0 is introduced in order to express the abstract concept of loop iterations. It enables us to represent values with respect to a loop iteration. For a loop nest of depth three, the description has three dimensions, i_0, i_1 and i_2. This property will be useful for the detection of loop carried memory dependencies, which will be explained in detail in Section 4.5.1. The value and access descriptions contain only a single output dimension. More output dimensions can occur

when expressing accesses to multidimensional arrays. However, the analysis currently supports only one dimensional arrays, hence a single output dimension is sufficient.

The value descriptions are always parametric in all global variables and function parameters. For readability we will only list those that are relevant in the current example.

4.2 Language Definition

We formally define our analysis on an abstract language, though the implementation works on full LLVM-IR.

$$func\ f(v_1, ..., v_n)\ s \quad \text{(Function Definition)}$$

FIGURE 4.4: Functions

Since the only types that can be used to access memory are integers, the language only allows integer values and they are not annotated explicitly. A function in our language (as defined in Figure 4.4) always returns an integer. The statement s is the function body and v_1 to v_n are parameters of the function.

$$
\begin{aligned}
e := \quad & f(e') && \text{(Function Call)} \\
& \mid A[e'] && \text{(Read Access)} \\
& \mid A[e' : e''] && \text{(Slicing)} \\
& \mid e' \circ e'' && \text{(Binary Operation)} \\
& \mid \square\ e' && \text{(Unary Operation)} \\
& \mid v && \text{(Variable)} \\
& \mid c && \text{(Constant)}
\end{aligned}
$$

$$\circ \in \{+, -, *, /, <, \leq, >, \geq, =, \wedge, \vee\} \quad \square \in \{\neg, -\}$$

FIGURE 4.5: Expressions

Figure 4.5 lists the expressions in the language and Figure 4.6 the statements.

$$
\begin{aligned}
s := \quad & A[e] := e' && \text{(Write Access)} \\
& \mid for\ (v := e; e'; \phi^*)\ s' && \text{(Loop)} \\
& \mid (if\ (e)\ then\ s'\ else\ s'')\ \phi^* && \text{(Conditional)} \\
& \mid return\ e && \text{(Return Statement)} \\
& \mid A\ := alloc(e) && \text{(Allocation)} \\
& \mid s'; s'' && \text{(Sequence)} \\
& \mid v := e && \text{(Assignment)}
\end{aligned}
$$

FIGURE 4.6: Statements

The language does not have pointers. For the theoretical part this is fine, as one can rewrite every pointer into an array and every pointer dereference into an array access. The language represents programs in SSA form, therefore on every join point of control flow phi nodes (denoted by ϕ^*) are placed.

4.3 Intra-procedural Analysis

The intra-procedural phase of the analysis describes how to analyze program regions in isolation, i.e., without resolving function calls. It consists of a value analysis and an analysis for all memory accesses inside the function.

The results of both, the value analysis and the inter-procedural analysis are polyhedral unions. In the first case they represent values of variables, in the second case they describe memory accesses.

The polyhedral unions are partially ordered by a subset relation. The most precise information is the empty polyhedron, whereas the least precise is the unconstrained polyhedron (a polyhedron describing the whole parameter space). The most precise information is also referred to as bottom, and the least precise information as top. The lattice we operate on is not finite. Section 4.3.1 defines the meet and join operations as well as the abstract transformers for statements and expressions.

4.3.1 Value Analysis

The results of the memory dependence analysis should be parametric. The value analysis works on the same domain to supply the memory dependence analysis with parametric control flow sensitive value information.

The context $\mathcal{C}$ is a mapping from program variables to a union of polyhedra.

$$\mathcal{C} : Vars \rightarrow \bigcup \mathcal{P}$$

The union of polyhedra is used to avoid overapproximation. Joining two polyhedra is not necessarily a polyhedron again. The convex hull could be computed to gain a single polyhedron, but this is generally overapproximative and might be costly.

Program variables *Vars* are split into array variables *VarsA* and scalar variables *VarsS* (with $VarsS \cap VarsA = \varnothing$). Initially every variable is mapped to the bottom element.

$$C_{initial} = \{(x, \bot) \mid x \in Vars\}$$

The join of two polyhedra is the union containing both polyhedra.

$$\cup : \mathcal{P} \times \mathcal{P} \rightarrow \bigcup \mathcal{P}$$
$$P_1 \cup P_2 := P_u$$
$$\text{where } P \subseteq P_u \Leftrightarrow P \subseteq P_1 \vee P \subseteq P_2$$

The join of two polyhedral unions is a union containing the elements of both unions.

$$\cup : \bigcup \mathcal{P} \times \bigcup \mathcal{P} \rightarrow \bigcup \mathcal{P}$$
$$P_{u1} \cup P_{u2} := P_u$$
$$\text{where } P \subseteq P_u \Leftrightarrow P \subseteq P_{u1} \vee P \subseteq P_{u2}$$

A join of two contexts is the join of every two corresponding polyhedral unions.

$$\cup : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$$
$$C_1 \cup C_2 := \{(x, P_{u1} \cup P_{u2}) \mid (x, P_{u1}) \in C_1,\ (x, P_{u2}) \in C_2,\ x \in \mathit{Vars}\}$$

The intersection of two polyhedra is a polyhedron again, therefore we use the regular polyhedral intersection $\cap_{poly}$. If the polyhedra have different parameters, the missing ones are added as unconstrained parameters.

$$\cap : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P}$$
$$P_1 \cap P_2 := P_1 \cap_{poly} P_2$$

The intersection of a union of polyhedra with another union of polyhedra is computed by intersecting all polyhedra from one union with every polyhedron from the other union.

$$\cap : \bigcup \mathcal{P} \times \bigcup \mathcal{P} \rightarrow \bigcup \mathcal{P}$$
$$P_{u1} \cap P_{u2} := \{P_1 \cap P_2 \mid P_1 \in P_{u1}, P_2 \in P_{u2}\}$$

In some cases the context should be constrained using polyhedra generated from a condition. In order to do this, the union of polyhedra generated from the expression is intersected with all polyhedra in the context.

$$\cap : \mathcal{C} \times \bigcup \mathcal{P} \rightarrow \mathcal{C}$$
$$C \cap P_e := \{(x, P_u \cap P_e) \mid (x, P_u) \in C,\ x \in \mathit{Vars}\}$$

Note that all the operations above are commutative.

4.3.1.1 Symbolic Evaluation of Expressions

The evaluation of expressions does not directly influence the context, they are only relevant when used within a statement. Therefore a different evaluation function is used for expressions and statements. The function $\langle\!\langle \cdot \rangle\!\rangle_C : (\mathcal{E} \times \mathcal{C}) \rightarrow \bigcup \mathcal{P}$ evaluates an expression within a context.

$$\langle\langle c\rangle\rangle_C = [\,] \rightarrow \{[\,] \rightarrow [c]\}$$

Constants are represented by a description which maps everything to the constant value.

$$\langle\langle v\rangle\rangle_C = [v] \rightarrow \{[\,] \rightarrow [v]\}$$

Variables are evaluated similarly, everything is mapped to the corresponding parameter.

There are two cases to distinguish for binary operations: either the operation $\circ$ is expressible in Presburger arithmetic and therefore has a corresponding polyhedral operation $\circ_{poly}$, or it is not expressible.

$$\langle\langle e \circ e'\rangle\rangle_C = \; p \circ_{poly} p' \text{ with } p = \langle\langle e\rangle\rangle_C \text{ and } p' = \langle\langle e'\rangle\rangle_C$$

In the case that the expression is representable, both operands are evaluated and the equivalent polyhedral operation is used to combine them.

$$\langle\langle e \circ_{ne} e'\rangle\rangle_C = \; [n] \rightarrow \{[\,] \rightarrow [n]\} \text{ with fresh}^1 \text{ parameter } n$$

For inexpressible operations a parameter is introduced. The evaluation of the expressions returns a description which maps to this parameter.

$$\langle\langle f(e)\rangle\rangle_C = \; [n] \rightarrow \{[\,] \rightarrow [n]\} \text{ with fresh parameter } n$$

$$\langle\langle A[e]\rangle\rangle_C = \; [n] \rightarrow \{[\,] \rightarrow [n]\} \text{ with fresh parameter } n$$

A similar evaluation is defined for function calls and memory accesses. Return values of functions are not taken into account and memory content is not tracked, therefore parameters are introduced for those values.

The parameters for unknown values or inexpressible expressions are useful when comparing accesses of certain regions, this is explained in more detail in Section 5.1.

$$\langle\langle \Box e\rangle\rangle_C = \; \Box_{poly}\, p \text{ with } p = \langle\langle e\rangle\rangle_C$$

The unary operations of the language are all expressible and handled analogously to the binary ones.

4.3.1.2 Symbolic Evaluation of Statements

The function $[\![\cdot]\!]_C : (\mathcal{S} \times \mathcal{C}) \rightarrow \mathcal{C}$ takes a statement and a context and returns a new context.

$$[\![A[e] := e']\!]_C = C$$

[1] The fresh parameters has a unique name.

Writing to some array changes the content of the array. Since the information about array variables that are stored in the context refers to the memory location of the array, not its content, this statement does not influence the context.

$$[\![\mathit{for}\ (v := e; e'; \phi^*)\ s]\!]_C = C \cup C_b \cup C_{SE}$$
$$\text{with}$$
$$C_{SE} : \text{Context Information from Scalar Evolution}$$
$$C_u = C \cup C_{SE}$$
$$C_b = [\![s]\!]_{(C_u \cap \langle\!\langle e' = true \rangle\!\rangle_{C_u})}$$

To symbolically evaluate loops, the loop body is evaluated with respect to the condition that guards the loop. Note that the information becomes flow sensitive with this evaluation function. The Scalar Evolution provides information about the loop iteration count, which is used to determine lower and upper bounds for induction variables. In particular, phi nodes are evaluated by the Scalar Evolution as they are used to obtain the closed form descriptions of loop variant variables.

We use the SSA properties throughout the work, therefore we need to ensure SSA form in our language. For readability, we will omit the phi selections for all examples.

Consider the example in Figure 4.7.

```
1 for (i := 0; i < N; ++i) {
2     ...
3 }
```

FIGURE 4.7: Loop Evaluation

The value of i ranges between 0 and $N - 1$. This information is provided by the Scalar Evolution and is added to the context. Every other statement inside the loop that uses i will have a description that is parametric in i. For the context collection phase this parametric description is kept and no fixpoint iteration or widening is applied. Later on this will become necessary and is explained in Section 4.3.4.

$$[\![(\mathit{if}\ (e)\ \mathit{then}\ s\ \mathit{else}\ s')\ \phi^*]\!]_C = C \cup C_t \cup C_f \cup C_\phi$$
$$\text{with}$$
$$C_t = [\![s]\!]_{(C \cap \langle\!\langle e = true \rangle\!\rangle_C)}$$
$$C_f = [\![s']\!]_{(C \cap \langle\!\langle e = false \rangle\!\rangle_C)}$$
$$C_\phi = \{(p,\ \langle\!\langle x \rangle\!\rangle_{C_t} \cup \langle\!\langle x' \rangle\!\rangle_{C_f}) \mid [p := \phi(x, x')] \in \phi^*\}$$

By evaluating the branches with respect to $e = true$ or $e = false$ respectively, the control flow information is taken into account. Conditions that either always evaluate to $true$ or $false$ will result in $\langle\!\langle e = false \rangle\!\rangle_C$ or $\langle\!\langle e = true \rangle\!\rangle_C$ being empty. The intersection with an empty union of polyhedra is again empty, therefore for every statement within the

branch the information will be bottom. This can only occur for dead statements. The phi nodes select the values that were assigned differently within the consequence and alternative. Again we will omit the phi nodes in the following for readability.

$$[\![return\ e]\!]_C = C$$

The return statement does not influence the context, as it only contains an expression and return values of functions are not used.

$$[\![s; s']\!]_C = [\![s']\!]_{[\![s]\!]_C}$$

To evaluate a sequence, the first statement is evaluated and the resulting context is used to evaluate the second statement.

$$[\![v := e]\!]_C = C \cup \{(v, \langle\!\langle e \rangle\!\rangle_C)\}$$

For an assignment the right-hand side expression is evaluated and the resulting polyhedral union is stored in the context for the variable v. Due to the SSA properties C does not refer to v yet.

$$\begin{aligned}
[\![A\ :=\ alloc(e)]\!]_C = \ & let \\
& \quad C' = C \cap \langle\!\langle A_{end} = A + e \rangle\!\rangle_C \cap \langle\!\langle A \le A_{end} \rangle\!\rangle_C \\
& in \\
& \quad \{(v,\ I \cap \langle\!\langle B_{end} < A \rangle\!\rangle_C) \mid (v, I) \in C',\ B \in VarsA\} \\
& \quad \cup \{(v,\ I \cap \langle\!\langle A_{end} < B \rangle\!\rangle_C) \mid (v, I) \in C',\ B \in VarsA\} \\
& end
\end{aligned}$$

When allocating memory, information about the position of the allocated array is added. No other array can overlap with the new memory region. The location of A is restricted such that all already defined arrays B in the context, are either located before or after A in the contiguous memory.

4.3.2 Local Access Descriptions

Local access descriptions are a polyhedral representation of single access statements. For every program point a set of read and write descriptions is built. Assume the statement $A[i] = B[j] + A[j+3]$ at program point x. The set of read and write descriptions for x are:

$$W_x = \{[A, A_{end}, i] \rightarrow \{[] \rightarrow [A + i] \ : \ A + i \leq A_{end}, \ i \geq 0\}\}$$
$$R_x = \{[A, A_{end}, j] \rightarrow \{[] \rightarrow [A + j + 3] \ : \ A + j + 3 \leq A_{end}, \ j \geq -3\},$$
$$[B, B_{end}, j] \rightarrow \{[] \rightarrow [B + j] \ : \ B + j \leq B_{end}, \ j \geq 0\}\}$$

The constraints $A + i \leq A_{end}$ and $i \geq 0$ are added because we assume the programs to access arrays in bounds. For the same reasons $A + j + 3 \leq A_{end}$ and $j \geq -3$ are added.

Read and write sets contain an access description for every array that is accessed at a program point.

4.3.3 Propagate Access Descriptions

So far we have only derived information about memory accesses of single statements. The accesses are now summarized to express all accesses within a program region. To determine whether two memory regions access the same memory locations, the intersection of the memory access descriptions for the regions is calculated. If this intersection is empty, the accesses are disjoint, thus the regions are independent with regards to memory effects.

```
1  func f(A[], x) {
2
3      // Begin R1
4      for (i := 0; i < 10; ++i) {
5          A[i] := i;
6      }
7      // End R1
8
9      // Begin R2
10     for (j := 10; j < 20; ++j) {
11         A[j] := x;
12     }
13     // End R2
14 }
```

FIGURE 4.8: Regions with Disjoint Memory Accesses

Consider the example in Figure 4.8. Marked are two regions, $R1$ and $R2$, which have disjoint memory accesses. Figure 4.9 shows the control flow graph and Figure 4.10 the control dependence graph for this example.

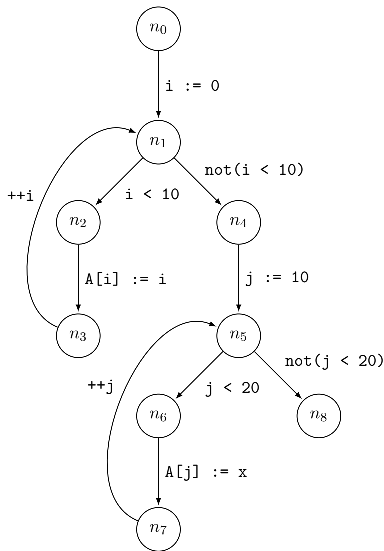


FIGURE 4.9: Control Flow Graph of Figure 4.8

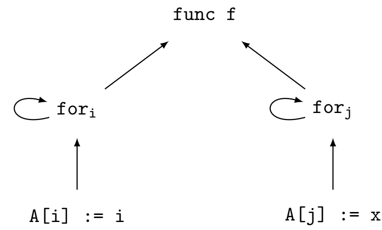


FIGURE 4.10: Control Dependence Graph of Figure 4.8

Assume the propagation of accesses was done along the control flow edges. The access $A[i]$ would propagate along the edge from n_1 to n_4 and over n_5 into the second loop where the access $A[j]$ is made. The results for n_8 would therefore include access descriptions for both $A[i]$ and $A[j]$. To decide whether the accesses of $R1$ and $R2$ overlap, the descriptions at n_4 and n_8 could be compared. However, the intersection of the descriptions at these program points will be non-empty, because both program points include the accesses to $A[i]$. This problem does not occur when propagating along control dependencies.

In the given example, the propagation starts at the access level. The accesses are lifted to loop level and then to function level. The different levels of lifting correspond to different granularities a parallelizer may be interested in. At access level, different accesses of the same loop can be compared to each other (which corresponds to comparing them within the same loop iteration in our case). At loop level loop carried dependencies can be detected (explained in detail in Section 4.5.1). Finally, at function level all effects of the function are summarized. The usage of control dependencies rather then control flow is only interesting for the intermediate steps; if we were only interested in function level accesses we could also use the CFG to propagate the accesses.

4.3.3.1 Propagation over Loops

Statements inside of loops can be executed more then once. If the index of an array access depends on variables that change within the loop, more than one position of the array is accessed during the execution of the loop. Consider Figure 4.11.

```
func f(A[]) {
    for (i := 0; i < 20; i := i + 5) {
        A[i] := i;
    }
}
```

FIGURE 4.11: Propagation of Access Polyhedra for Loops

The local access description for line 3 looks like this:

$$W_3 = \{[A, A_{end}, i] \to \{[] \to [A + i] \ : \ A + i \leq A_{end}, \ 0 \leq i\}\}$$

Considering only the statement in line 3, exactly one position, namely position i of the array A, is accessed. The fact that the statement is executed multiple times needs to be taken into account to describe all accesses of the loop. The function parameter A is a fixed parameter inside the function, it is always the same array with the same logical address. This is different for i, which is not fixed, its value changes in every iteration. Therefore the access description should also depend on the loop iteration. The iteration dependent description for the loop looks like this:

$$W_{2-4} = \{[A, A_{end}] \to \{[i_0] \to [A + 5i_0] \ : \ A + 5i_0 \leq A_{end}, \ 0 \leq i_0\}\}$$

The value of i is incremented by 5 in every iteration, hence the access position is $5i_0$. With the additional context information about the loop bound the following description is gained:

$$W_{2-4} = \{[A, A_{end}] \to \{[i_0] \to [A + 5i_0] \ : \ A + 5i_0 \leq A_{end}, \ 0 \leq i_0 \leq 3\}\}$$

Access descriptions at function level are parametric in the global variables and function parameters. This description should be independent from loop iterations but still capture all accesses of a loop inside the function. In order to lift the access description W_{2-4} to function level the input dimension i_0 needs to be removed. An existentially quantified variable is introduced in order to eliminate i_0. The access description at function level looks like this:

$$W_f = \{[A, A_{end}] \to \{[] \to [A + x] \ : \ \exists y. \ 5y = x, \ A + x \leq A_{end}, \ 0 \leq x \leq 15\}\}$$

ISL [1] provides methods to eliminate dimensions, which either introduce existentially quantified variables or apply a Fourier-Motzkin-Elimination. More details on this can be found in the ISL Manual [18] and the source code documentation [19].

4.3.3.2 Propagation over Conditionals

When propagating descriptions from inside the consequence or alternative of a conditional to the outside, the condition can be used to constrain the accesses. Consider Figure 4.12.

```
1  func f(A[], x) {
2      res := 0;
3      if (x < 5)
4          res := A[1];
5      return res;
6  }
```

Figure 4.12: Presburger Condition

The local access description for Figure 4.12 is shown in Figure 4.13.

$$R_4 = \{[A, A_{end}] \rightarrow \{[] \rightarrow [A+1] \ : \ A \leq A_{end}\}\}$$

Figure 4.13: Local Access Polyhedron for line 4 in Figure 4.12

The access in line 4 happens only if $x < 5$ holds, therefore this constraint is added to the description.

$$R_{3-4} = \{[A, A_{end}, x] \rightarrow \{[] \rightarrow [A+1] \ : \ A \leq A_{end}, \ x \leq 4\}\}$$

Figure 4.14: Local Access Polyhedron with Condition

Note that this restriction makes the polyhedron smaller. Before using the information of the condition, x was unconstrained, now it is constrained.

Lifting the description in Figure 4.14 to function level is easy: nothing changes. All parameters in the description correspond to function parameters.

There are conditions that cannot be expressed using Presburger arithmetic. Consider Figure 4.15.

```
1  func f(A[], x) {
2      res := 0;
3      z := x * x;
4      if (z < 5)
5          res := A[1];
6      return res;
7  }
```

Figure 4.15: Non-Presburger Condition

As mentioned in Section 4.3.1.1, a parameter is introduced for expressions that cannot be accurately represented in the polyhedral union. Since the multiplication in line 3 of Figure 4.15 is not expressible, a parameter for z is introduced. The local description of the access is shown in Figure 4.16.

$$R_5 = \{[A, A_{end}, z] \rightarrow \{[] \rightarrow [A+1] \ : \ A \leq A_{end}, \ z \leq 4\}\}$$

Figure 4.16: Local Access Polyhedron with Non-affine Condition

At function level, the description should no longer be parametric in z as it is neither a function parameter nor a global variable. There is no further information about z in the context. It cannot be determined whether $z \leq 4$ holds, so the conservative assumption is that the condition may be true and the access happens. Eliminating z leads to losing this constraint and assuming that whenever the function is called, the access is made. This is a sound overapproximation on the accesses within the function.

The purpose of introducing a parameter for non-affine expressions is explained in Section 5.1.

4.3.4 Overapproximation and Intra-procedural Widening

In most static intra-procedural analysis widening is crucial for the precision of the results. We have not seen widening so far, as it happens implicitly.

The polyhedral analysis is a strong relational analysis, which allows modeling the program exact for all linear affine parts. Most of the presented examples contained accurately expressible accesses and conditions, as those are the ones which are interesting to discuss. For all other expressions an overapproximation is applied. One example for this was presented in Figure 4.15. The condition could not be expressed in terms of the parameters of the function and the truth value could not be computed. The assumption for such conditions is that both, the consequence and the alternative of the conditional, are executed. This soundly overapproximates the accesses of a conditional.

Parameters introduced for non-linear-affine expressions are always assumed to have the value top. Whenever accesses are lifted and this parameter is projected out, the expression behind this parameter is approximated by assuming all possible values for it.

The Scalar Evolution can calculate bounds on the iteration count of a loop. This is achieved by computing how often the back edges to the loop header are taken. The bound on the iterations does not need to be a simple constant, but can be an arbitrary Scalar Evolution expression. This expression can especially refer to a program variable, which can be correlated to a parameter in the polyhedral description. This enables parametric bounds on loop iterations.

$$[A, N] \rightarrow \{[i_0] \rightarrow [A + i_0] \ : \ A \leq A_{end}, \ 0 \leq i_0 \leq N\}\}$$

FIGURE 4.17: Description of an Access within a Loop with Parametric Bound N

Figure 4.17 describes an access to an array within a loop. The number of loop iterations is bounded by N. Assume that N is a function parameter, then the function level description is given by Figure 4.18.

$$[A, A_{end}, N] \rightarrow \{[] \rightarrow [y] \ : \ A \leq y \leq A + N, \ A \leq A_{end}, \ y \leq A_{end}\}$$

FIGURE 4.18: Lifted Description of the Access in Figure 4.17

The access description is not overapproximated by the lifting, due to the calculated bound.

However, the Scalar Evolution is not always able to calculate the iteration count. In this case an overapproximation is made when lifting the accesses. Figure 4.19 shows an access without a bound on the iteration count, and Figure 4.20 the access widened to function level.

$$[A, A_{end}] \rightarrow \{[i_0] \rightarrow [A + i_0] \ : \ A \leq A_{end}, i_0 \geq 0\}$$

FIGURE 4.19: Description of Access without Loop Bound

$$[A, A_{end}] \rightarrow \{[] \rightarrow [y] \ : \ A \leq y \leq A_{end}, \ A \leq A_{end}\}$$

FIGURE 4.20: Lifted Description of the Access in Figure 4.19

Note that in both lifted descriptions the accesses are constrained by $y \leq A_{end}$ because accesses are assumed to be inbound.

Widening in an intra-procedural analysis is only necessary for statements within loops. Due to the parametric description and the concept of loop iterations, an explicit widening is not necessary. Nevertheless, the results are sound, as values for variables inside a loop (or accesses made inside a loop) are overapproximated if the loop iteration count is unknown.

4.4 Inter-procedural Analysis

In order to generate sound results for the accesses of a function, calls inside the function need to be considered. The presented intra-procedural analysis generates access descriptions for a function under the assumption that called functions do not access any memory. This is not sound for a standalone intra-procedural analysis, because it underapproximates the accesses of a function. The intra-procedural analysis is embedded in an inter-procedural analysis in our case. The underapproximation is intended to compute the least (not greatest) fixpoint in the inter-procedural analysis.

An inter-procedural analysis generates access descriptions for functions with respect to called functions. In the inter-procedural analysis the results of the intra-procedural analysis can be used as initial access descriptions for each function. In order to describe the accesses made by a call, the current access description of the called function is instantiated with the arguments handed over at the call site. The resulting description is added to the access description of the caller.

Note that the inter-procedural analysis described above is context sensitive. It aims to find the least fixpoint by starting with the assumption that no memory is accessed within callees.

4.4.1 Application of Inter-procedural Analysis

We will demonstrate our inter-procedural analysis using the example in Figure 4.21.

```
1  G[];
2
3  func g(m, n) {                                func findMin(A[], m) {
4
5      a := 0;                                       min := A[0];
6      b := 0;                                       for(i := 1; i < m; ++i){
7                                                        elem := A[i];
8      if (m < n) {                                      if(elem < min){
9          a := findMin(G, m);                               min := elem;
10         b := findMin(G[m + 1 : ], n);                 }
11     }                                             }
12
13     return a + b;                                 return min;
14 }                                             }
```

FIGURE 4.21: Example Inter-procedural Analysis

The function $findMin$ searches the minimal element in the array A starting at position 0 up to position $m-1$. Function g calculates the sum of the minima for the parts 0 up to $m-1$ and $m+1$ up to n of the global array G. In the case where m is greater than or equal to n, no memory accesses are made.

The intra-procedural analysis calculates descriptions for all explicit accesses inside a function, considering no memory access is done by called functions. The results of the intra-procedural analysis are shown in Figure 4.22.

$$R_{findMin} = \{[A, A_{end}, m] \to \{[] \to [y] \ : \ A \le y < A + m, \ A \le A_{end}, \ y \le A_{end}\}\}$$
$$W_{findMin} = \{\}$$
$$R_g = \{\}$$
$$W_g = \{\}$$

FIGURE 4.22: Intra-procedural Access Descriptions for g and $findMin$

Inside function g are no memory accesses, so both the read and the write ployhedral unions are empty. In $findMin$ only read accesses occur which read from positions $A[0]$ up to $A[m-1]$.

These results are used as initial access descriptions for the inter-procedural analysis. As $findMin$ does not call any functions, the overall accesses of $findMin$ are the results

from the intra-procedural analysis. Note that we omit the *end*-constraints for better readability in the following examples.

$$R_{findMin} = \{[A,m] \rightarrow \{[] \rightarrow [y] \ : \ A \le y < A+m\}\}$$
$$W_{findMin} = \{\}$$

FIGURE 4.23: Accesses for $findMin$

For g we resolve the function calls to $findMin$. The first call is $a = findMin(G,m)$. We adapt the access description of $findMin$ to express this call using the following steps:

1. Create a primed version of the original access.
$$\{[A',m'] \rightarrow \{[] \rightarrow [y] \ : \ A' \le y < A'+m'\}\}$$
2. Generate equality constraints for every function parameter with its corresponding argument.
$$A' = G, \ m' = m$$
3. Add those constraints to the primed version of the access.
$$\{[G,A',m',m] \rightarrow \{[] \rightarrow [y] \ : \ A' \le y < A'+m', \\ A' = G, \ m' = m\}\}$$
4. Project all primed parameter out of the polyhedral description.
$$\{[G,m] \rightarrow \{[] \rightarrow [y] \ : \ G \le y < G+m\}\}$$
5. Add all context constraints which hold for the relevant parameters at the position of the function call.
$$\{[G,m,n] \rightarrow \{[] \rightarrow [y] \ : \ G \le y < G+m, \ m < n\}\}$$

FIGURE 4.24: Function Call Resolution

After applying these steps, the fixpoint is reached, nothing changes in the next iteration. Hence, the final results are empty write sets for both functions and the read accesses are as shown in Figure 4.25.

$$R_g = \{[G,m,n] \rightarrow \{[] \rightarrow [y] \ : \ (G \le y < G+m, \ m < n) \\ or \\ (G+m+1 \le y < G+m+1+n, \ m < n)\}\}$$
$$R_{findMin} = \{[A,m] \rightarrow \{[] \rightarrow [y] \ : \ A \le y < A+m\}\}$$

FIGURE 4.25: Accesses of g and $findMin$

The description of the access in R_g is a union of two polyhedra. The union is constructed by the join of the access descriptions of the calls. A polyhedron is a conjunction of constraints. For readability the conjunctions are separated by commas. Disjunctions

cannot occur within a polyhedron, only inside a polyhedral union. They are expressed by an "or".

4.4.2 Fixpoint Iteration

A fixpoint iteration in the inter-procedural analysis is necessary, because changes in the access descriptions of a function need to be propagated to the call sites to obtain sound results. The iteration order does not influence the results directly, but it influences the number of iterations required to reach the fixpoint.

For acyclic call graphs the fixpoint iteration will always terminate. Within strongly connected components the fixpoint iteration might not terminate, therefore widening is applied if the fixpoint iteration does not terminate within a fixed amount of iterations.

The example in Figure 4.26 motivates the need of widening.

```
1 func f(A[], x) {
2
3     if (x > 0) {
4         A[x] := 7;
5         return f(A, x-1);
6     }
7
8     return x;
9 }
```

FIGURE 4.26: Recursive Function

The initial description for the accesses in f is given by:

$$W_f = \{[A, x] \rightarrow \{[] \rightarrow [A + x] \ : \ x \geq 1\}\}$$

This access describes exactly one position that is written, namely the position $A + x$. The function call inside f need to be analyzed in the next step. To resolve the call $f(A,\ x - 1)$ the steps described in Figure 4.24 are used to gain the following result:

$$W_{f(A,\ x-1)} = \{[A, x] \rightarrow \{[] \rightarrow [A + x - 1] \ : \ x \geq 2\}\}$$

We get $A + x - 1,\ x \geq 2$ from $A' + x',\ x' \geq 1$ as we used $x' = x - 1$, which matches the intuition that the recursive call only accesses memory again if x was large enough initially. The description is exact so far, when considering only one recursive call. The new access description for f is:

$$W_f = \{[A, x] \rightarrow \{[] \rightarrow [y] \ : \ y = A + x,\ x \geq 1 \ or \ y = A + x - 1,\ x \geq 2\}\}$$

This description for the accesses of f differs from the ones before. Therefore callers of the function f need to update their result again. The function is recursive, so it needs to be updated itself. In the next iteration the access $A + x - 2,\ x \geq 3$ is discovered. Here a problem can be detected: in every further iteration the bound on x increases and one additional position of A is accessed. The fixpoint iteration does not terminate for this example, thus widening is needed.

4.4.3 Widening

Our widening is applied after a fixed amount of iterations without reaching a fixpoint. This can only happen within a strongly connected component (SCC). The widening we present is applied at call sites. It is only applied for calls from a member of a SCC to a member of the same SCC.

In the following description we refer to the values handed over at a call site as arguments and to function parameters as parameters.

If the arguments are independent from parameters of the function, no special widening needs to be applied. There are two cases where this can occur: A local variable or a constant is passed to the function. Constants do not need to be overapproximated as they cannot influence the termination behavior of the presented fixpoint iteration. Local variables do not influence the termination in our case, as we eliminate (and therefore already overapproximate) them when lifting the access to function level (c.f. Section 4.3.3).

The interesting case are parameter dependent arguments. To avoid divergence of the fixpoint iteration, within the SCC the evolution of arguments should be considered, rather than the exact expression at the call sites. If we have a parameter b and a call site with the argument $b + x$ this call site can be overapproximated by assuming that b evolves in a certain direction. The evolution depends on the information given for x. In the case that x is positive, the value for $b + x$ is larger than b. To resolve the function call containing the argument $b + x$ the constraint that it is equal to the called function parameter is not added. The widening adds a constraint to express that “something larger than b” is handed over.

Consider Figure 4.26 again to get an intuition how this widening works and how it is implemented. The fixpoint iteration did not terminate for this example. For a human it is easy to see that x is getting smaller and A stays the same, but the analysis needs an algorithmic solution to detect this.

Every argument is compared to every parameter of the caller. First A and $x - 1$ are compared. They are considered distinct, as they refer to different function parameters.

Then x and $x - 1$ are compared, which both refer to x. In this case the polyhedral description of both expressions is used to test how they can be correlated. The value of $x - 1$ is always smaller than the value of x. The value for A stays the same.

Overall four types of evolutions are considered: the parameter is passed to the function as is, it will be larger, smaller, or the evolution could not be determined. To give an example of the last case consider Figure 4.27.

```
1 func f (A, x) {
2     z = A[0];
3     return g(x + z);
4 }
```

FIGURE 4.27: Call Argument Depends on Function Parameter

In this case the relation between x and $x + z$ cannot be determined. The variable z can have an arbitrary value, hence the evolution of x is unknown.

For the example in Figure 4.26 the relation between A and the argument handed over to f again is equal, for x and $x - 1$ the relation is smaller. With these relations, the results of the called function can be widened. First, a primed version for the description is constructed:

$$W_f = \{[A', x'] \rightarrow \{[] \rightarrow [y] \; : \; y = A' + x', \; x' \geq 1 \; or \; ...\}\}$$

The constraint $x' < x$ is added, which corresponds to the intuition that x will be smaller in the following iterations. For the array $A' = A$ is added.

$$W_f = \{[A, A', x, x'] \rightarrow \{[] \rightarrow [y] \; : \; y = A' + x', \; x' \geq 1, \; x' < x, \; A' = A \; \texttt{or} \; ..., \; x' < x, \; A' = A\}\}$$

To express that x should be an arbitrary value that is smaller, x' is projected out. This is also done for all other primed parameters. The result describes accesses with not only a single position x that is accessed, but accesses from 0 up to x.

$$W_f = \{[A, x] \rightarrow \{[] \rightarrow [y] \; : \; A \leq y \leq A + x, \; x \geq 1\}\}$$

For this example the upper bound x and the lower bound of 0 are kept, which describes the bounds accurately. The resulting access description is a tight description of all accesses within f.

The analysis will not always be able to find the upper bound. If the bound is given by a non-affine expression, it will be overapproximated when lifting the access to function level. In this case the widening is overapproximative, it assumes that every element up to the end of the array is accessed.

If we cannot determine the evolution, widening in both directions is applied. Values for parameters that are passed to the function unchanged are not widened.

We do not consider modulo so far, therefore monotonicity for all our operations is given.

To show that the widening is sound, it needs to be shown that the widening is going up in the lattice and that it is terminating. For the three cases where widening is applied, the easiest case is the one assuming all values for the argument. We directly go to the top element of the lattice, therefore it directly terminates and also goes up in the lattice. For both other cases we know the evolution and apply the widening depending on it. As our operations are monotone, dropping the bound in this direction will capture the values of the evolution and is therefore a progress and terminates directly.

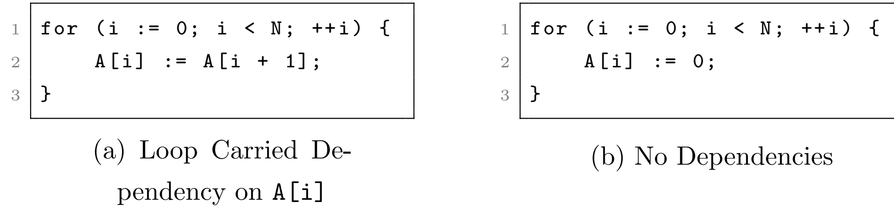
4.5 Memory Dependence Analysis

The memory dependence analysis should give information about accesses made in functions or regions. It should also be able to decide whether accesses in different regions must, may, or must not overlap. In our analysis the decision of memory accesses being disjoint reduces to the question if the intersections of polyhedra are empty. Given two program regions we can intersect the read and write polyhedra and if anything other than the read polyhedra overlap, we cannot guarantee disjoint memory accesses statically. There might be conditions under which the intersection would be empty, but which cannot statically be proven. In these cases run-time checks could be generated, for which our analysis already holds all necessary information.

For arbitrary program parts the analysis can directly provide summarized access descriptions and therefore show program regions disjoint. Loop carried dependencies need an additional processing step, which is explained in the following.

4.5.1 Loop Carried Dependencies

If a loop has loop carried dependencies, one loop iteration uses values that are calculated in an earlier iteration of the loop. Figure 4.28 shows two examples. The one on the left-hand side shifts the positions of the values in the array A by one. The example on the right-hand side zero-initializes an array from position 0 to position $N-1$.



(a) Loop Carried Dependency on `A[i]`

(b) No Dependencies

FIGURE 4.28: Loop Carried Dependencies

In the absence of loop carried dependencies parallel execution of each loop iteration is possible. In Figure 4.28 (b) the order in which zero is written to the array does not matter, whereas in Figure 4.28 (a) the result depends on the order of the execution of the loop iterations.

Our analysis can support parallelization with information about memory access dependencies in different loop iterations. What we need to show is that there are no dependencies between any two (or more) iterations of the loop. Therefore we compare the access descriptions for one iteration with those of an arbitrary subsequent iteration.

To explain how this is achieved consider Figure 4.29:

```
1 for (i := 0; i <= N; i := i + 2) {
2     A[i] := A[i + 1];
3 }
```

FIGURE 4.29: Loop without Loop Carried Dependencies

In the example we have one loop variant variable i. A variable is loop variant, if it can change within the loop, all other variables are considered loop invariant. In each loop iteration i is incremented by 2. The array is accessed at positions i and $i+1$. Because the loop stride is 2, we do not have any memory dependencies between two loop iterations.

We describe how to detect this using our analysis results. The initial access descriptions are shown in Figure 4.30.

$$W_2 = \{[A, A_{end}, N] \to \{[i_0] \to [A + 2i_0] \ : \ A \le A_{end}, \ A + 2i_0 \le A_{end}\}\}$$
$$R_2 = \{[A, A_{end}, N] \to \{[i_0] \to [A + 2i_0 + 1] \ : \ A \le A_{end}, \ A + 2i_0 + 1 \le A_{end}\}\}$$

FIGURE 4.30: Local Accesses for Figure 4.29

During the analysis a bound on the loop iteration count is collected. The values for i are constrained using this bound. Figure 4.31 shows the value description for i with bounds.

$$[N] \to \{[i_0] \to [2i_0] \ : \ 0 \le i_0 \le \lfloor N/2 \rfloor\}$$

FIGURE 4.31: Value Description for i

Remember that i_0 describes the concept of a virtual loop iteration, so this description expresses the value of i depending on the current iteration. In iteration 0 the value of i is 0, in the next iteration it is 2 and so on. The maximal number of loop iterations for the loop is $\lfloor N/2 \rfloor + 1$, therefore $i0$ can range from 0 up to $\lfloor N/2 \rfloor$.

In order to compare one iteration with an arbitrary other iteration of the same loop, a description is constructed that contains all possible further values for the loop variant variables. To obtain this description, we first represent all possible increments as shown in Figure 4.32.

$$[] \to \{[] \to [inc] \ : \ inc \ \texttt{mod} \ 2 = 0, \ inc > 0\}$$

FIGURE 4.32: Value Description for the Increment of i

By adding these increments to the variant variable, every subsequent value for the variable is expressed.

$$[N] \to \{[i_0] \to [2i_0 + inc] \ : \ 0 \le 2i_0 + inc \le \lfloor N/2 \rfloor, \ inc \ \texttt{mod} \ 2 = 0, \ inc > 0\}$$

FIGURE 4.33: Value Description for Subsequent Iterations of i

The value description for i in Figure 4.30 can be replaced by this description to get all subsequent iterations. Figure 4.34 shows the result. The + in W_{2+} and R_{2+} indicates that all following accesses are described.

$$\begin{aligned} W_{2+} = \{[A, A_{end}, N] \to \\ \{[i_0] \to [A + 2i_0 + inc] \ : \\ A \le A_{end}, \\ A + 2i_0 + inc \le A_{end}, \\ 0 \le 2i_0 + inc \le \lfloor N/2 \rfloor, \\ inc \ \texttt{mod} \ 2 = 0, \\ inc > 0\}\} \end{aligned}$$

$$\begin{aligned} R_{2+} = \{[A, A_{end}, N] \to \\ \{[i_0] \to [A + 2i_0 + inc + 1] \ : \\ A \le A_{end}, \\ A + 2i_0 + inc + 1 \le A_{end}, \\ 0 \le 2i_0 + inc + 1 \le \lfloor N/2 \rfloor, \\ inc \ \texttt{mod} \ 2 = 0, \\ inc > 0\}\} \end{aligned}$$

FIGURE 4.34: Subsequent Accesses for Figure 4.30

In order to find out whether there are dependencies, we intersect the write accesses W_{2+} and W_2 with all other accesses (W_2, R_2, R_{2+} and W_{2+}, R_2, R_{2+}). In this case the intersections are all empty, hence we have proven the accesses disjoint and can conclude that there are no loop carried memory dependencies.

We only get tight descriptions if we know all loop variant variables and their evolution. If there are loop variant variables for which the Scalar Evolution does not have an Add

Recurrence we need to overapproximate them. To show the difficulties including loop variant variables, consider the example in Figure 4.35:

```
1 x := 0;
2 for (i := 0; i < N; ++i) {
3     x := 1 - x;
4     A[i + x] := A[i + x] + 7;
5 }
```

FIGURE 4.35: Loop Variant Variable without Add Recurrence

The loop in the example carries dependencies. There is a value dependence on x and the accesses are not disjoint in every iteration. In iteration zero $A[1]$ is accessed, which is accessed again in iteration one. The value of $A[1]$ in the second iteration depends on the value in the first iteration and is both read and written.

The description for both accesses is given by:

$$W_4 = R_4 = \{[A, A_{end}, x] \to \{[i_0] \to [A + i_0 + x] \ : \ A \leq A_{end}, \ A + i_0 + x \leq A_{end}\}\}$$

The interpretation of the description needs to be done carefully in this case. When just comparing the accesses of one iteration with its successors, $i_0 + x$ and $i_0 + x + inc$ are compared, which are always different. The problem is that x changes in every iteration, but there is no monotone pattern to describe the change. In this case the value of x needs to be overapproximated before the comparison. This overapproximation leads to losing all information about the access:

$$W_{2-5} = R_{2-5} = \{[A, A_{end}] \to \{[] \to [y] \ : \ A \leq y \leq A_{end}\}\}$$

The loop iterations cannot be proven disjoint, which is sound and even exact because the loop has loop carried memory dependencies.

We shortly sum up the interference check for loops again in Figure 4.36.

Our detection of loop carried dependencies is based on comparing an iteration with subsequent iterations. This is sufficient for two reasons: we have monotone operations and the variant variables evolution is also monotone (or overapproximated otherwise). By considering the evolution of the variant variable every possible behavior is covered. We do not need to compare an iteration with any previous iterations, as this is captured by comparing the previous iterations with subsequent iterations.

1. Collect all local access descriptions for accesses within a loop.
2. Overapproximate loop variant variables where no Add Recurrence can be found.
3. Create the description for all further loop iterations. Look up loop variant variable with Add Recurrences VA. For every $v \in VA$:
   (a) Construct a polyhedron with the constraints ($inc = v \; \texttt{mod} \; stride$) and ($inc > 0$)
   (b) Look up the value description for v in the context
   (c) Add the constructed polyhedron to this description and get the next iteration accesses by replacing the variant variable with those descriptions
4. Intersect write descriptions with all other descriptions

FIGURE 4.36: Check for Loop Carried Dependencies

We want to discuss our introductory example shortly to show the limitations of the static analysis part and explain ideas for run-time. The example is shown again in Figure 4.37.

```
1 for (i := 0; i < N; ++i) {
2     A[i] := A[i + C];
3 }
```

FIGURE 4.37: Parametric Loop Carried Dependencies

The loop has loop carried memory dependencies if C is less than N. Without further information about C our analysis cannot show that $A[i]$ and $A[i + C]$ are disjoint. The subsequent iteration accesses are described by $A[i + inc]$ and $A[i + C + inc]$. The introduced inc is bounded by N, but there is no bound for C, so the write and read accesses can overlap when inc is equal to C. The accesses are disjoint when $C > N$ holds. By synthesizing constraints similar to this one and using them as run-time checks, we could gain less conservative results.

This feature is currently not supported by the implementation. The theory is elaborated and the descriptions are given in the correct form, but the interpretation and interface to query loop dependencies need to be implemented.

Chapter 5

Limitations

The values of our analysis are constrained to integer values, as only those can occur in memory accesses. Neither our value analysis nor the further parts are able to handle floating point values.

Working with polyhedra with lots of parameters is slow. For every global array we add two parameters (encoding begin and end). Access descriptions for accesses within a function contain parameters for non-linear affine expressions and functions parameters. Algorithms solving problems given in Presburger arithmetic have doubly exponential complexity [20] in the number of parameters involved.

Non-linear access patterns are not expressible within our polyhedral description. The following section discusses how we can sometimes bypass overapproximation and when we need to overapproximate.

5.1 Non-affine Expressions

The polyhedral representation only captures Presburger arithmetic, so multiplications of variables or division are not accurately expressible. In Section 4.3.3.2 we already mentioned how we overapproximate them to get access descriptions for whole functions. We have not explained why we introduce parameters for them in the first place, which is what we are going to do now.

Consider example 5.1. There are two loops which write results of function calls to the array A.

```
n := x*x;

for (i := 0; i < n; i++) {
    A[i] := f(i); \\ f(i) does not access A
}

for (i := n; i < n + 20; i++) {
    A[i] := g(i); \\ g(i) does not access A
}
```

FIGURE 5.1: Usage of a Parameter for Non-Affine Expressions

The loops do not access the same elements in A: one starts at 0 and writes up to position $n-1$ and the other starts at n and writes up to $n+19$. Considering only the shown region, we can prove that those accesses to A do not overlap as we have the fixed parameter n in our polyhedron. We do not know anything about its value, but we do not have to in order to prove the accesses disjoint. Conceptually the parameter n is similar to a function parameter, but with a different scope.

For some regions we can make use of the strengths of the parametric representation to be less overapproximative. This advantage is lost when looking at the memory accesses at function level: n is only a local parameter and we need to overapproximate in order to remove it from the access description.

Again, for more precise results we could add run-time checks for the affected expressions. However, this creates new problems; consider Figure 5.2.

```
func f(a[], n) {

    if (n * n < 15) {
        // some memory accesses
    }
}
```

(a)

```
func f(a[]) {
    n := a[0];
    if (n * n < 15) {
        // some memory accesses
    }
}
```

(b)

FIGURE 5.2: Functions with Non-affine Expressions

In example 5.2 (a) we could make use of the check whether $n * n < 15$ holds and provide two different polyhedral descriptions depending on on the result. This produces some overhead without the guarantee of being profitable. For example 5.2 (b) it is even more complicated: We cannot use n inside a constraint at function level as it is a local variable and therefore not in scope at function level. The call context of the function and scoping of the variables used in the non-affine expressions need to be taken into account, so the position of the checks need to be calculated carefully.

Chapter 6

Evaluation

In the introduction we presented some examples we want to tackle with our analysis. In Chapter 4 we defined the analysis and the resulting descriptions for memory accesses. Now we have another look at the challenges of automatic parallelization presented in Chapter 1 and evaluate if and how our results can help to enable parallelization.

As mentioned before, loop carried dependencies are not included in the evaluation as they are not supported by the implementation as of now.

6.1 Enabling Parallelization

In order to parallelize program regions, a variety of conditions need to hold. Our analysis should help to find out whether the memory accesses within the regions overlap.

6.1.1 Aliasing

Accesses made using different pointers do not necessarily lead to disjoint memory accesses. Figure 6.1 shows two variants of the aliasing example from the introduction.

```
1 func f (A[], B[]) {
2
3     for (i := 0; i < N; ++i) {
4         for (j := 5; j < M; j := j + 2) {
5             A[i] := B[j];
6         }
7     }
8 }
```

(a) Arrays as Arguments

```
1 A := alloc(N);
2 B := alloc(M);
3
4 for (i := 0; i < N; ++i) {
5     for (j := 5; j < M; j := j + 2) {
6         A[i] := B[j];
7     }
8 }
```

(b) Locally Allocated Arrays

Figure 6.1: Aliasing Case Study

The accesses in Figure 6.1 (a) are independent if A and B do not alias. In Figure 6.1 (b) A and B cannot alias due to the allocations above the accesses. The results for the write accesses for both examples are shown in Figure 6.2.

$$
\begin{aligned}
W_5 = \{[A, A_{end}, B, B_{end}, N, M] \rightarrow \\
\{[i_0, i_1] \rightarrow [A + i_0] \ : \\
0 \leq i_0 < N, \\
A + i_0 \leq A_{end} \\
A \leq A_{end},\ B \leq B_{end}\}\}
\end{aligned}
$$

(a) Arrays as Arguments

$$
\begin{aligned}
W_5 = \{[A, A_{end}, B, B_{end}, N, M] \rightarrow \\
\{[i_0, i_1] \rightarrow [A + i_0] \ : \\
0 \leq i_0 < N, \\
A + N = A_{end},\ B + M = B_{end}, \\
A \leq A_{end},\ B \leq B_{end}, \\
(B_{end} < A \ or \ A_{end} < B)\}\}
\end{aligned}
$$

(b) Locally Allocated Arrays

FIGURE 6.2: Write Access Descriptions for Figure 6.1

The accesses are parametric in all program variables and additionally in two artificial parameters which encode the respective *end* positions of the arrays. The position of the access is $A + i_0$ and the bounds of i_0 are computed as described in Section 4.3.4.

The generation of constraints to express disjoint memory regions such as $B_{end} < A \ or \ A_{end} < B$ in Figure 6.1 (b) has not been demonstrated so far. When analyzing an allocation, the bound on the size of the array is added to the context. Additionally, constraints that the newly allocated array does not overlap with any other array are added. The formal definition of how the constraints are generated is shown in Section 4.3.1.2. The accesses in Figure 6.1 (a) are not constrained this way, since the allocation of the arrays is not within the analyzed region.

These differences in the description are important when resolving aliasing. In the case of the locally allocated arrays we can intersect the descriptions for the write access to A and the read access to B and the result is empty. We can statically guarantee that these accesses must not overlap. The intersection of the read and write accesses of Figure 6.1 (a) is not empty. It depends on the values for A, B, i and j.

The analysis is purely static so far, in order to give a sound result for the case of non-empty intersections we need to assume that the accesses may overlap. A different way of interpreting the results is to require that there is no aliasing (maybe because another analysis has computed those results). Accesses through different pointers are then considered distinct and we can prove the accesses disjoint.

Our description is strong enough to express aliasing, but the interpretation of the results is very conservative when assuming aliasing. This is a consequence of the fact that aliasing is usually not statically decidable. Run-time checks to resolve aliasing would make the analysis less conservative. These checks could be synthesized from the access descriptions, which is left for future work.

6.1.2 Recursion

Our analysis is inter-procedural and therefore needs to be capable of cycles in the call graph. Cycles occur in strongly connected components, recursion is a special case of such a cycle.

In Section 4.4.2 we have already seen an example where we gain accurate results for a recursive function.

```
1  void merge (int *a, int n, int m) {
2      int i, j, k;
3      int *x = malloc(n * sizeof (int));
4      for (i = 0, j = m, k = 0; k < n; k++) {
5          x[k] = j == n       ? a[i++]
6               : i == m       ? a[j++]
7               : a[j] < a[i] ? a[j++]
8               :               a[i++];
9      }
10     for (i = 0; i < n; i++) {
11         a[i] = x[i];
12     }
13     free(x);
14 }
15
16 void merge_sort (int *a, int n) {
17     if (n < 2)
18         return;
19     int m = n / 2;
20     merge_sort(a, m);
21     merge_sort(a + m, n - m);
22     merge(a, n, m);
23 }
```

Figure 6.3: Merge Sort[1]

We want to consider merge sort, a recursive sorting algorithm as shown in Figure 6.3. Merge sort uses the divide and conquer principle to sort an array. It goes in recursion for the first and second half of the array, and merges the presorted parts afterwards. Since the two recursive calls work on disjoint parts of the array, they can be parallelized.

[1] http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort#C

We would expect our analysis to detect this, but it does not. The reason for this is the loss of the upper bound for the call to *merge*. If we look at the loop in line 4, we can see that k is incremented in every iteration, but i and j need not to be incremented in every iteration. The scalar evolution is restricted to affine recurrences, which cannot be found for i and j. Therefore only the value of k is bounded by n and the bounds on i and j are missed. This leads to an imprecise result for *merge* which propagates to the results of *merge_sort*. With information about piecewise affine Add Recurrences we would be able to prove the calls to *merge_sort* disjoint.

6.2 Benchmarks

The implementation was tested using the official LLVM test suite [21]. It is designed for performance testing.

Our focus is not primarily the performance, but rather the granularity of the access descriptions. Nevertheless, of the 525 executed test 435 passed and 90 failed the performance testing with respect to compile-time. Timeouts were expected, because of the super exponential complexity.

Count	Description
2200	Number of Modules
42537	Total number of functions
5341	Functions that can be analyzed

TABLE 6.1: Total Numbers of Modules, Functions and Amount of Analyzable Functions

Table 6.1 shows the overall test results. Modules in LLVM can be correlated to translation units in C. Out of 2200 Modules containing 42537 functions definitions, the analysis was able to gain results for 5341 functions. There are several reasons why the analysis cannot compute results. First of all, structs, which are widely used throughout the tests cases, are currently not supported. Functions including a struct access cannot be analyzed. This propagates to through the call tree. Table 6.2 gives an overview of the most problematic cases for the analysis.

Count	Description
7340	Load or Store to struct, union or multidimensional array
2224	Indirect calls (e.g. trough function pointer)
1490	Function parameter, argument or global variable cannot be expressed
1071	Irregular control flow

TABLE 6.2: Factors for Overapproximations

For indirect function calls we do not know which function is called, and in particular, whether a function within our module is called. This function can have access to all global arrays and to everything accessible from the arguments it is called with.

We rely on the Scalar Evolution to evaluate all kinds of expressions, though the Scalar Evolution does not handle floating point expressions. Within the value analysis, these values are directly overapproximated to top. This can also happen for global variables and function arguments. At this point the analysis stops. Row 3 in Table 6.2 lists those cases.

The first three problems listed in Table 6.2 are assumed to be not accurately analyzable, as an overapproximation of those would result in overapproximating all reachable arrays to a full access.

The irregular control flow entry represents functions where the analysis fails to fully understand the structure. This influences mostly the value analysis, in particular the condition propagation (for more details see Section 7.1). In this case, instead of giving up, the results of the value analysis are overapproximated by mapping every value to top. Accesses within the function are still analyzed afterwards.

Count	Description
23673	Functions outside translation unit
10058	Overapproximated calls to functions outside translation unit

Table 6.3: Calls to Functions Outside the Translation Unit

For all functions of which we can see only the declaration, we need to consider that they may touch memory. We overapproximate every call to such a function, to accesses for function arguments and global arrays. Table 6.3 shows the number of overapproximations and the number of unknown functions.

Count	Description
361	Modules
791	Analyzable functions

Table 6.4: Modules not Containing Problems Described in Table 6.2

Consider Table 6.4. Out of the 2200 modules of all test cases, only 361 neither contain any struct or union accesses nor indirect function calls. Within these tests, many function calls to functions outside the translation unit occur (see Table 6.5), which makes the results for those modules overapproximate as well.

Count	Description
838	Functions outside translation unit
1349	Overapproximated calls to functions outside translation unit

Table 6.5: Calls to Function Outside the Translation Unit

Overall, the results are inconclusive which is mostly due to missing implementation features. With these features and the loop carried memory dependencies, more comprehensive results could be achieved by using the analysis within an automatic parallelizer.

Chapter 7

Technical Details

The analysis is implemented in C++ and analyzes programs given in LLVM-IR. The concept of a module in LLVM can be correlated to a translation unit in C. Modules can hold global variables and functions, functions consist of basic blocks whereas they are constructed from single instructions. LLVM provides an interface to write analyses which work on the intermediate representation, the LLVM Pass Framework. This framework allows to add passes easily. Each analysis, transformation or optimization in LLVM is a different pass and is scheduled by the Pass Manager. Passes can have a different granularity depending on which logical unit they belong to. If it is sufficient to analyze one function at a time and the results do not relay on other functions, a function pass is a reasonable solution. As we need to iterate over the whole call graph within a module, our pass is a module pass.

The module pass is responsible for the coordination of the analysis. The first step is running the intra-procedural analysis on every function of the module. Within the intra-procedural analysis a value analysis is performed first.

The value analysis looks at every instruction and extracts useful information. We have described the value analysis on our high level language in Section 4.3.1. For the parts of the analysis that diverge a lot in the implementation will now explain how we translated the concepts to fit the low level LLVM-IR equivalents.

7.1 Condition Propagation on the CFG

There is no syntactic representation for conditionals or loops in LLVM-IR. In our language it was easy to find the scope of a condition introduced by one of those concepts, which is not that obvious in the intermediate representation.

Consider the function in Figure 7.1.

```
func f () {
    ...
    if (x < y)
        ...
    else
        ...
}
```

FIGURE 7.1: Example Condition Propagation

The consequence is executed when $x < y$ holds, otherwise the alternative is executed. Figure 7.2 shows the corresponding LLVM-IR code.

```
define i32 @func() #0 {
entry:
  ...
  %cmp = icmp slt i32 %x, %y
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  ...
  br label %if.end

if.else:                                          ; preds = %entry
  ...
  br label %if.end

if.end:                                           ; preds = %if.else,
  ...                                                       %if.then
}
```

FIGURE 7.2: Example Condition Propagation in LLVM-IR

The analysis starts at the entry block and follows branches along the control flow. When reaching a conditional branch, such as the one in line five, the condition and the negated condition are computed. Then the immediate post-dominator of the branch destinations is computed. In the example the destinations are `%if.then` and `%if.else`, the immediate post-dominator is `%if.end`. The `%if.then` block is evaluated under the assumption that our condition holds, analogously for the else branch with the negated condition. The algorithm for this computation is recursive which resolves nesting and the scoping of the conditions. After both branches have computed their results, they are joined. Remember that we have polyhedral unions, which allows us to not overapproximate when joining these results. For the *then* branch we will get results that contain the collected values with the additional constraint $x < y$ and for the *else* branch with the constraint $x >= y$, so they are disjoint. After the join the analysis continues at the post-dominator.

The described propagation corresponds to a propagation along the control dependencies.

There are a couple of corner cases to take care of: First of all, the immediate post-dominator does not need to exist. This can occur if there is a return statement inside a conditional our due to control flow induced by exception handling. Another LLVM pass to normalize the CFG is executed before our pass. This pass cannot always normalize the CFG, in these cases our analysis will drop all value analysis results for the function.

The condition cannot always be expressed with a polyhedral union. In this case we assume that both branches are executed unconditionally, which is a sound overapproximation. On the other hand, we may detect that a condition cannot hold. We have found a dead branch, and the bottom value is assigned to all values inside the corresponding blocks.

For non-SSA programs the condition of a conditional branch does not need to hold until the immediate post-dominator is reached. One of the variables inside the condition may be overwritten within a block. This is not a problem in our case since the analysis always operates on an SSA form program and therefore it is clear which values are addressed within the condition.

Switch statements in LLVM-IR are handled using the same concepts.

7.2 Further Remarks

Our analysis requires some properties on the given LLVM-IR in order to compute useful results. One requirement has been mentioned in Section 7.1, namely to be able to find post-dominators for basic blocks. A second requirement is, that the *Promote Memory To Register Pass* of LLVM is executed before our analysis. This pass eliminates unnecessary reads and writes to memory, e.g. for local variables that are stored to memory in one instruction and are immediately loaded again for the next instruction. This elimination is crucial for the Scalar Evolution as it does not reason about values stored to or loaded from memory. Our results strongly dependent on those of the Scalar Evolution and therefore we require this pass.

In our language we did not introduce the concept of pointer but rather only referred to arrays. In LLVM-IR we handle every pointer variable as an array. Accesses are always relative to some pointer or array variable. The intermediate representations uses special instructions to address a certain position relative to a pointer, called *GetElementPointerInstruction*. This instruction can also describe an access to a multidimensional array or a struct. We do not support structs or multidimensional arrays, so our analysis gives up on a function when they are accessed. To overapproximate them, we would need

to exclude the possibility that they hand out pointers to functions outside the module, which could result in accesses we cannot detect.

In the last phase of our analysis we resolve function calls and iterate until results stabilize. We have already described how we handle strongly connected components and functions within our module, but we have not talked about functions outside the module that are called. These functions cannot be analyzed as we only see the declarations, so we need to overpproximate them soundly. This is done by generating whole array accesses for all global arrays and for all pointers handed over to these functions. In a couple of cases we can do better: Functions in LLVM can be annotated with *read none*, which indicates that this function does not accesses any memory. Those functions are not overapproximated when called (and also not analyzed when within our module). LLVM provides a module pass named `TargetLibraryInfo` which, in combination with `Memory Builtins`, enables us to detect functions like `malloc` or `free`. Especially `malloc` is interesting for our analysis, as a call to malloc creates a new array. Therefore instead of handling `malloc` as a regular function, we have a special treatment: We handle the pointer we receive as a new array and do not assume memory accesses to arrays when `malloc` is called.

Chapter 8

Conclusion and Future Work

8.1 Future Work

There are several approaches to improve analysis precision as well as applicability. The support of multidimensional arrays could be added without the need of new logic. Return values of functions are not supported but may be a nice additional feature.

Currently structs are not supported at all, we need to give up when structs are involved. To make the approximation for structs better, one could look at the fields of the struct. When the struct contains only primitive data types and no pointer, it could be handled similar to arrays. An approximation for other structs could also be thought of.

The handling for function calls to functions outside the translation unit is quite overapproximative. A more advanced approach for the approximation of these functions could be thought of, e.g., link time optimizations.

For simplicity we consider that no overflow occurs. The theory is based on integers of unlimited size, so $x < x + 1$ is always true. Polly [3] is aware of possible overflows. LLVM as well as the Scalar Evolution support integer overflow detection by marking values that cannot overflow. For all other computations Polly adds constraints describing that the results belong to a residue class. This residue class is calculated by *value* `mod` *maxInt*.

As we have seen before, the Scalar Evolution is able to calculate loop bounds. With these loop bounds we constrain loop variant variables for which the Scalar Evolution provides an Add Recurrence. But, as described in Section 6.1.2, we still miss loop bound information on variables for which the Scalar Evolution could not find an affine recurrence. An additional detection for upper bounds of loop variant variables is desirable.

We do not work with modulo during the analysis[1], although it is part of Presburger arithmetic. This could be added in the analysis, but then the inter-procedural widening is no longer sound. It relies on the monotonicity property of the operations, that the modulo operation does not have.

Loop carried dependencies are challenging for every parallelizer. Detecting those dependencies is a desirable feature. This is elaborated in the theory and supported by the descriptions, but not yet by the implementation of the dependence analysis.

An interesting new extension would be to use the gained access descriptions to extract run-time checks. These checks could be placed in front of a critical section to select either a parallelized version or a sequential version at run-time. This includes two major tasks: constructing the run-time checks and finding out when they are actually useful for a parallelizer.

8.2 Conclusion

The work presents a context and flow sensitive inter-procedural analysis for memory dependencies. The main contribution is, that the analysis works at an inter-procedural level and provides parametric accesses descriptions for different kinds of granularities of program regions. The descriptions are carefully designed to address a variety of different problems. In order to compare regions within a function, control flow sensitive information is used. Loop carried memory dependencies can be detected using the access descriptions as they encode values with respect to loop iterations. The summarized descriptions for functions enable the comparison of regions that contain function calls. There are no restrictions on the program under analysis. Solutions for recursions, function calls outside the module, loops and other language elements are provided.

Although the descriptions for memory accesses are powerful, the practical results so far are not as good as expected. First, the analysis is purely static which makes it conservative. Run-time checks would improve on the results. The main problems for a loss of precision in practice are missing information on loop variant variables and the expensive calculations for intersections and joins. The last aspect is the one we assume to be the most critical: To reduce the time for the calculations, a restriction on the number of parameters needs to be considered. Sometimes the introduction of a parameter does not pay off, an overapproximation is therefore profitable to reduce the time for the computations. The question remains how to detect those parameters.

[1] For the detection of loop carried dependencies modulo is used, but these calculations are done when the analysis has already collected all access descriptions.

Overall the approach has a lot of potential for further research.

Bibliography

[1] Sven Verdoolaege. ISL: An Integer Set Library for the Polyhedral Model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15581-2, 978-3-642-15581-9. URL http://dl.acm.org/citation.cfm?id=1888390.1888455.

[2] Scalar Evolution Pass. URL http://llvm.org/docs/Passes.html#scalar-evolution-scalar-evolution-analysis. [Accessed 2015-12-07].

[3] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04):1250010, 2012. doi: 10.1142/S0129626412500107. URL http://www.worldscientific.com/doi/abs/10.1142/S0129626412500107.

[4] Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du I Congrés des Mathématiciens des Pays Slaves*, pages 292–101, 1929.

[5] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL http://doi.acm.org/10.1145/115372.115320.

[7] Robert A. Van Engelen. Symbolic Evaluation of Chains of Recurrences for Loop Optimization. Technical report, Computer Science Department, Florida State University, 2000.

[8] Robert van Engelen. Efficient Symbolic Analysis for Optimizing Compilers. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 118–132, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41861-X. URL http://dl.acm.org/citation.cfm?id=647477.727776.

[9] Johnnie L. Birch. Using the Chains of Recurrences Algebra for Data Dependence Testing and Induction Variable Substitution, 2002.

[10] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of Recurrences - a Method to Expedite the Evaluation of Closed-Form Functions. In *In International Symposium on Symbolic and Algebraic Computing*, pages 242–249. ACM Press, 1994.

[11] Eugene V. Zima. On Computational Properties of Chains of Recurrences. In *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, ISSAC '01, pages 345–, New York, NY, USA, 2001. ACM. ISBN 1-58113-417-7. doi: 10.1145/384101.384148. URL http://doi.acm.org/10.1145/384101.384148.

[12] Tobias Grosser. *Enabling Polyhedral Optimizations in LLVM.* Diploma thesis, University of Passau, April 2011. URL http://polly.llvm.org/publications/grosser-diploma-thesis.pdf.

[13] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. *Int. J. Parallel Program.*, 31(4):251–283, August 2003. ISSN 0885-7458. doi: 10.1023/A:1024597010150. URL http://dx.doi.org/10.1023/A:1024597010150.

[14] Radu Rugina and Martin C. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. *ACM Trans. Program. Lang. Syst.*, 27 (2):185–235, March 2005. ISSN 0164-0925. doi: 10.1145/1057387.1057388. URL http://doi.acm.org/10.1145/1057387.1057388.

[15] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM. doi: 10.1145/512760.512770. URL http://doi.acm.org/10.1145/512760.512770.

[16] Nicolas Halbwachs. Delay Analysis in Synchronous Programs. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 333–346, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-56922-7. URL http://dl.acm.org/citation.cfm?id=647762.735515.

[17] Thomas A. Henzinger and Pei-Hsin Ho. A Note on Abstract Interpretation Strategies for Hybrid Automata. In *Hybrid Systems II*, pages 252–264, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-60472-3. URL http://dl.acm.org/citation.cfm?id=646875.709992.

[18] Sven Verdoolaege. ISL Manual. URL http://isl.gforge.inria.fr/manual.pdf. [Accessed 2015-12-07].

[19] Sven Verdoolaege. ISL Source. URL http://repo.or.cz/w/isl.git. [Accessed 2015-12-07].

[20] Michael J. Fischer and Michael O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. In *Proceedings of the SIAM-AMS Symposium in Applied Mathematics. 7*, pages 27–41, February 1974.

[21] LNT Testsuite for LLVM. URL http://llvm.org/docs/lnt/intro.html. [Accessed 2015-12-07].