Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation

Antoine Miné
Sorbonne Universités, UPMC Univ. Paris 06, CNRS, LIP6
antoine.mine@lip6.fr

the essence of knowledge
Boston — Delft

Foundations and Trends® in Programming Languages

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

A. Miné. *Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation*. Foundations and Trends® in Programming Languages, vol. 4, no. 3-4, pp. 120–372, 2017.

This Foundations and Trends® issue was typeset in LaTeX using a class file designed by Neal Parikh. Printed on acid-free paper.

ISBN: 978-1-68083-386-7

Foundations and Trends® in Programming Languages

Volume 4, Issue 3-4, 2017

Editorial Scope

Topics

Foundations and Trends® in Programming Languages publishes survey and tutorial articles in the following topics:

- Abstract interpretation
- Compilation and interpretation techniques
- Domain specific languages
- Formal semantics, including lambda calculi, process calculi, and process algebra
- Language paradigms
- Mechanical proof checking
- Memory management
- Partial evaluation
- Program logic
- Programming language implementation
- Programming language security
- Programming languages for concurrency
- Programming languages for parallelism
- Program synthesis
- Program transformations and optimizations
- Program verification
- Runtime techniques for programming languages
- Software model checking
- Static and dynamic program analysis
- Type theory and type systems

Information for Librarians

Foundations and Trends® in Programming Languages, 2017, Volume 4, 4 issues. ISSN paper version 2325-1107. ISSN online version 2325-1131. Also available as a combined paper and online subscription.

Foundations and Trends® in Programming Languages
Vol. 4, No. 3-4 (2017) 120–372

DOI: 10.1561/2500000034

Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation

Antoine Miné
Sorbonne Universités, UPMC Univ. Paris 06, CNRS, LIP6
antoine.mine@lip6.fr

Contents

Abstract

Born in the late 70s, Abstract Interpretation has proven an effective method to construct static analyzers. It has led to successful program analysis tools routinely used in avionic, automotive, and space industries to help ensuring the correctness of mission-critical software.

This tutorial presents Abstract Interpretation and its use to create static analyzers that infer numeric invariants on programs. We first present the theoretical bases of Abstract Interpretation: how to assign a well-defined formal semantics to programs, construct computable approximations to derive effective analyzers, and ensure soundness, i.e., any property derived by the analyzer is true of all actual executions — although some properties may be missed due to approximations, a necessary compromise to keep the analysis automatic, sound, and terminating when inferring uncomputable properties. We describe the classic numeric abstractions readily available to an analysis designer: intervals, polyhedra, congruences, octagons, etc., as well as domain combiners: the reduced product and various disjunctive completions. This tutorial focuses not only on the semantic aspect, but also on the algorithmic one, providing a description of the data-structures and algorithms necessary to effectively implement all our abstractions. We will encounter many trade-offs between cost on the one hand, and precision and expressiveness on the other hand. Invariant inference is formalized on an idealized, toy-language, manipulating perfect numbers, but the principles and algorithms we present are effectively used in analyzers for real industrial programs, although this is out of the scope of this tutorial.

This tutorial is intended as an entry course in Abstract Interpretation, after which the reader should be ready to read the research literature on current advances in Abstract Interpretation and on the design of static analyzers for real languages.

A. Miné. *Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation*. Foundations and Trends® in Programming Languages, vol. 4, no. 3-4, pp. 120–372, 2017.

DOI: 10.1561/2500000034.

1

Introduction

While software are naturally meant to be run on computers, they can also be studied, manipulated, analyzed, either by hand or mechanically, that is, by other computer programs. A common example is compilation, which transforms programs in source code form into programs in binary code suitable for direct interpretation by a processor — or by a virtual machine, yet another program. This tutorial concerns *static analysis*, a less common example of computer programs manipulating other programs. A static analyzer is a program that takes as input a program and outputs information about its possible behaviors, without actually executing it.

In a broad sense, static analysis also covers syntactic analyses, that search for predefined patterns, as well as code quality metrics, such as counting the number of comments. However, we focus here on *semantic-based* static analyses. These methods output program properties that are *provably correct* with respect to a clear mathematical formalization of program behaviors. Such a high level of confidence in the analysis results is necessary in many applications, ranging from compiler optimization to program verification. One example property is that two pointers never alias. If proved true, the property can be exploited by a compiler to enable optimizations that would be incorrect in the presence of aliasing. Another example is finding bounds on array index expressions. This can be exploited in program verification to ensure that a program is free from out-of-bound array accesses. For the correctness proof to be valid, it is necessary to ensure that the inferred bounds indeed encompass all possible index values computed in all possible executions of the program.

Formal methods. The idea of reasoning with mathematical rigor about programs dates back from the early days of computers [Turing, 1949] and lead to the rich field of *formal methods* with the pioneering work of Hoare [1969] and Floyd [1967] on program logic. The lack of automation for writing and checking program proofs hindered these early efforts. In fact, Turing famously proved the undecidability of the halting problem, and Rice [1953] generalized this result, stating that all non-trivial properties about programs are undecidable. Hence, program verification cannot be fully automated. This fundamental limitation can be sidestep in different ways, leading to the various flavors of program verification methods used today. Cousot and Cousot [2010] classify current formal methods into three categories, depending on whether automation, generality, or completeness is abandoned:

- *Deductive Methods*, which inherit directly from the work of Hoare [1969] and Floyd [1967], use interactive logic-based tools, including proof assistants such as Coq [Bertot and Castéran, 2004] and theorem provers such as PVS [Owre et al., 1992]. These tools are largely mechanized, but rely ultimately on the user, to a varying degree, to guide the proof.

- *Model Checking*, pioneered by Clarke et al. [1986], restricts program verification problems to decidable fragments. Initially restricted to finite models, it has since been generalized to infinite-state but regular models by McMillan [1993] in symbolic model checking. In practice, this often means that a model must be extracted, by hand, before the analysis can be performed. Alternatively, software bounded model checkers, such as CBMC [Clarke et al., 2004], analyze programs in actual programming languages such as C, but consider only a finite part of their executions.

- *Static Analysis*, studied in this tutorial, performs a direct analysis of the original source code, considering all possible executions and without user intervention, but resorts to approximations and analyzes the program at some level of abstraction that forgets about details that are, hopefully, irrelevant for the kind of properties checked. The abstraction is incomplete and can miss some properties, resulting in false alarms, i.e., the program is correct but the analyzer cannot prove it.

Abstract Interpretation. The theory of *Abstract Interpretation*, introduced by Cousot and Cousot [1977], is a general theory of the approximation of formal program semantics. It is an invaluable tool to prove the correctness of a static analysis, as it makes it possible to express mathematically the link between the output of a practical, approximate analysis, and the original, uncomputable program semantics. Both are seen as the same object, at different levels of abstraction. Additionally, Abstract Interpretation makes it possible to derive, from the original program semantics and a choice of abstraction, a static analysis that is correct by construction. Finally, the notion of abstraction is a first class citizen in Abstract Interpretation: abstractions can be manipulated and combined, leading to modular designs for static analyses. In this tutorial, we will design static analyses by Abstract Interpretation.

The rest of this chapter presents informally static analyses by Abstract Interpretation in order to derive simple numeric properties on the variables of a program.

1.1 A First Static Analysis: Informal Presentation

Let us consider, as first example, the program in Fig. 1.1. The `mod` function takes two arguments, A and B, then computes in Q and R, respectively, the integer dividend A/B and the remainder $A\%B$, and finally returns R. This very naive function is written in a C-like language, and enriched with a `//@requires` annotation, written in the

```
    //@ requires A >= 0 && B >= 0;
    int mod(int A, int B) {
1:    int Q = 0;
2:    int R = A;
3:    while (R >= B) {
4:      R = R - B;
5:      Q = Q + 1;
6:    }
7:    return R;
    }
```

Figure 1.1: A simple C function returning the modulo R of its arguments, with some precondition on the arguments A and B.

ACSL specification language [Cuoq et al., 2012], stating that it is always called with positive values for A and B.

The most straightforward way to model the function behavior is to consider execution traces: we execute the function step by step (where each step is a simple assignment or test) and record, at each step, the current program location and the value of each variable in scope. In our example, a program state would have the form $\langle l : a, b, q, r\rangle$ where l is the line number from Fig. 1.1 and a, b, q, r are, respectively, the values of variables A, B, Q, R. The execution starting with $A = 10$ and $B = 3$ would give the following trace (where variables not yet in scope are not shown in the state):

$$\begin{aligned}
&\langle 1 : 10, 3\rangle \to \langle 2 : 10, 3, 0\rangle \to \langle 3 : 10, 3, 0, 10\rangle \\
&\to \langle 4 : 10, 3, 0, 10\rangle \to \langle 5 : 10, 3, 0, 7\rangle \to \langle 6 : 10, 3, 1, 7\rangle \\
&\to \langle 4 : 10, 3, 1, 7\rangle \to \langle 5 : 10, 3, 1, 4\rangle \to \langle 6 : 10, 3, 2, 4\rangle \\
&\to \langle 4 : 10, 3, 2, 4\rangle \to \langle 5 : 10, 3, 2, 1\rangle \to \langle 6 : 10, 3, 3, 1\rangle \to \langle 7 : 10, 3, 3, 1\rangle
\end{aligned}$$

i.e., the function returns 1, which is indeed the remainder of 10 by 3.

There are many such executions, one for each initial value of A and B, but we can see intuitively that, in each of them, R and Q remain positive. This information can be useful to a compiler (which can then use unsigned types and arithmetic instead of signed ones) or to a program verifier (e.g., if the result of the function is used in an unsigned context).

1.1.1 Sign Analysis

Our first static analysis attempts to establish rigorously the sign of the variables. A naive method, which ensures that all possible program behaviors are considered, is to effectively simulate every possible execution by running the program, and then collect the signs of variable values along these executions. Naturally, this is not very efficient, and we will construct a far more efficient method.

A key principle of Abstract Interpretation is replacing these actual, so-called *concrete*, executions, with *abstract* ones. For a sign analysis, we replace the concrete states mapping each variable to an integer value with an abstract state mapping each variable to a sign. Program instructions can then be interpreted in the world of signs by employing well-known rules of signs, such as $(\geq 0) + (\geq 0) = (\geq 0)$, i.e., positive plus positive equals positive, etc. Starting from positive values of A and B, one possible execution is:

$$\begin{aligned}
&\langle 1 : (\geq 0), (\geq 0)\rangle \to \langle 2 : (\geq 0), (\geq 0), 0\rangle \to \langle 3 : (\geq 0), (\geq 0), 0, (\geq 0)\rangle \\
&\to \langle 4 : (\geq 0), (\geq 0), 0, (\geq 0)\rangle \to \langle 5 : (\geq 0), (\geq 0), 0, \top\rangle \\
&\to \langle 6 : (\geq 0), (\geq 0), (\geq 0), \top\rangle \to \langle 4 : (\geq 0), (\geq 0), (\geq 0), \top\rangle \\
&\to \langle 5 : (\geq 0), (\geq 0), (\geq 0), \top\rangle \to \langle 6 : (\geq 0), (\geq 0), (\geq 0), \top\rangle \\
&\to \langle 7 : (\geq 0), (\geq 0), (\geq 0), \top\rangle
\end{aligned}$$

where $\top$ indicates that the sign is unknown — the variable may be positive or negative. Note that the value of Q, which is 0 when first going through location 4, becomes (≥ 0) at the second passage, which is expected as Q increases. Collecting the sign of the variables at each program point, we can annotate the program from Fig. 1.1 with sign information; the result is shown in Fig. 1.2. These annotations are *invariants*: the values of the variables in every concrete execution passing through a given control location satisfy the sign property we provided at this location.

Note that, at the end of the function, we have no information on R ($R = \top$) while, in fact, R is always positive. We can trace the introduction of an uncertainty, $\top$, to the computation, at line 4, of `R - B` which, in the sign domain, gives $(\geq 0) - (\geq 0) = \top$. Indeed, `R - B` can only be proven to be positive if we know that $R \geq B$, which is

```
      //@requires A >= 0 && B >= 0;
      int mod(int A, int B) {
1:        ≀ A = (≥0), B = (≥0) ≀
          int Q = 0;
2:        ≀ A = (≥0), B = (≥0), Q = 0 ≀
          int R = A;
3:        ≀ A = (≥0), B = (≥0), Q = 0, R = 0 ≀
          while (R >= B) {
4:            ≀ A = (≥0), B = (≥0), Q = (≥0), R = (≥0) ≀
              R = R - B;
5:            ≀ A = (≥0), B = (≥0), Q = (≥0), R = ⊤ ≀
              Q = Q + 1;
6:            ≀ A = (≥0), B = (≥0), Q = (≥0), R = ⊤ ≀
          }
7:        ≀ A = (≥0), B = (≥0), Q = (≥0), R = ⊤ ≀
          return R;
      }
```

Figure 1.2: Modulo function from Fig. 1.1 annotated with the result of a sign analysis in comments.

not a sign information. So, while $R = (\geq 0)$ is an invariant and a sign property, it cannot be found by reasoning purely in the sign domain. Failure to infer the best invariants expressible in the abstract world is common in Abstract Interpretation and motivates the introduction of more expressive domains, as we will do shortly. The reader familiar with deductive methods will have guessed that this is related to the fact that $R = (\geq 0)$ is an invariant but not an *inductive invariant*. We will discuss this connection in depth later.

Note also that the test `R >= B` is interpreted in the abstract as $\top \geq (\geq 0)$, which is inconclusive. This means that, while we chose, in our abstract execution, to iterate the loop twice, longer executions with more loop iterations are also valid. The program, in the abstract, becomes non-deterministic. We argue, informally for now, that further iterations will not bring any new possible sign values: we have reached a fixpoint. Another key part of Abstract Interpretation is to know how to precisely iterate loops in the abstract, and when to stop, to guarantee that all possible program behaviors have been considered. It will be discussed at length in this tutorial.

```
    //@requires A >= 0 && B >= 0;
    int mod(int A, int B) {
1:      ≀ A ≥ 0, B ≥ 0 ∫
        int Q = 0;
2:      ≀ A ≥ 0, B ≥ 0, Q = 0 ∫
        int R = A;
3:      ≀ A ≥ 0, B ≥ 0, Q = 0, R = A ∫
        while (R >= B) {
4:          ≀ A ≥ 0, B ≥ 0, Q ≥ 0, R ≥ B ∫
            R = R - B;
5:          ≀ A ≥ 0, B ≥ 0, Q ≥ 0, R ≥ 0 ∫
            Q = Q + 1;
6:          ≀ A ≥ 0, B ≥ 0, Q ≥ 1, R ≥ 0 ∫
        }
7:      ≀ A ≥ 0, B ≥ 0, Q ≥ 0, 0 ≤ R < B ∫
        return R;
    }
```

Figure 1.3: Modulo function from Fig. 1.1 annotated with the result of an affine inequality analysis in comments. In red, we show the invariants that were not found by the sign analysis of Fig. 1.2.

1.1.2 Affine Inequalities Analysis

The sign analysis we presented is one of the simplest and least expressive static analysis there is. We illustrate the other end of the spectrum with a static analysis able to infer affine inequalities between variables. The invariants it computes on our modulo example are presented in Fig. 1.3. Its principle remains the same: we propagate an abstract representation of variable values through the program. However, it no longer has the simple form of a map from variables to abstract values, but is rather a conjunction of affine inequalities that delimit the set of possible concrete states the program can be in. As a consequence, the abstraction can represent relations, i.e., it is a *relational analysis*. Geometrically, we obtain a polyhedron.

Program instructions can still be applied on polyhedra. For instance, an assignment `Q = Q + 1` is modeled as a translation, while a test `R >= B` is modeled as adding an affine constraint. The exact algorithms will be described in details in Sect. 5.3. They borrow heavily

```
    A = 0; B = 0;
1:  ⦃ A ∈ [0,0], B ∈ [0,0] ⦄
    while
2:      ⦃ A ∈ [0,100], B ∈ [0,+∞] ⦄
        (A < 100) {
3:      ⦃ A ∈ [0,99], B ∈ [0,+∞] ⦄
        A = A + 1;
4:      ⦃ A ∈ [1,100], B ∈ [0,+∞] ⦄
        B = B + 1;
5:      ⦃ A ∈ [1,100], B ∈ [1,+∞] ⦄
    }
6:  ⦃ A ∈ [100,100], B ∈ [0,+∞] ⦄
```

(a)

iteration	A	B
1	$[0,0]$	$[0,0]$
2	$[0,1]$	$[0,1]$
3	$[0,2]$	$[0,2]$
...	...	
100	$[0,99]$	$[0,99]$
101	$[0,100]$	$[0,100]$
102	$[0,100]$	$[0,101]$
103	$[0,100]$	$[0,102]$

(b)

Figure 1.4: Interval analysis of a simple loop (a) and the detailed iteration for location 2 (b).

from the classic mathematical theory of convex polyhedra. We can see, in Fig. 1.3, that the analysis is now able to exactly represent $R \geq B$, and can thus deduce that R remains positive, which was not possible in the sign analysis.

1.1.3 Iterations

To illustrate more clearly the need to iterate loops in the abstract, we consider the simple loop in Fig. 1.4.(a) that increments A and B from 0 to 100. The program is annotated with invariants computed in yet another abstraction, *intervals*, which infers variable bounds: a lower bound and an upper bound. This popular abstraction will be discussed at length in Sect. 4.5. We can easily compute the abstract effect of instructions using interval arithmetic. For instance, `A = A + 1` adds 1 to both the lower and the upper bounds of A.

Program location 2 in Fig. 1.4.(a) is the location reached just before testing the condition `A < 100` a first time to determine whether to enter the loop at all, and reached again after each loop iteration before testing the condition to determine whether to reenter the loop body for a new iteration. The corresponding invariant is called a *loop invariant*, and provides a convenient summary of the behavior of the loop. A classic execution would have, for A at location 2, the sequence of values: 0, 1, 2, ..., 100. The output of the analysis must, however, provide a single interval for location 2 that takes into account all the reachable values. Hence, the abstract semantics accumulates, at each iteration, every new value with that of preceding iterations. This flavor of semantics, useful for verification, is called a *collecting semantics*.

The iteration is shown in Fig. 1.4.(b). We observe that, for A, the iteration stabilizes at $[0, 100]$ after 101 iteration steps, allowing us to deduce that A equals 100 when the program ends, after the loop exit condition `A >= 100`. Such convergence is long. Another important contribution of Abstract Interpretation is a set of *convergence acceleration* methods, to construct more efficient analyses that use less iterations.

In some cases, the plain abstract iteration may not even converge. This is the case for variable B in Fig. 1.4 as there is no test on B to bound it. Convergence acceleration will ensure that, after a finite number of accelerated iterations, this behavior is detected and we output the stable interval $B \in [0, +\infty]$.

1.1.4 Precision

As seen on the modulo example from Figs. 1.2–1.3, the result of a static analysis depends on the abstract domain of interpretation, but it will always represent an over-approximation of the set of possible program states. More expressive abstractions generally lead to tighter over-approximations, and so, more precise results.

Figure 1.5 illustrates this by showing a set of planar points (representing, e.g., a set of concrete program states over two variables) and its best enclosing into a polyhedron (in the affine inequality domain), a box (in the interval domain), and a quarter-plane (in the sign domain). Polyhedra add less spurious states with respect to the concrete world, but, as we will see, polyhedra algorithms are also more complex and more costly, leading to a slower analysis. There is a tradeoff to reach between precision and cost.

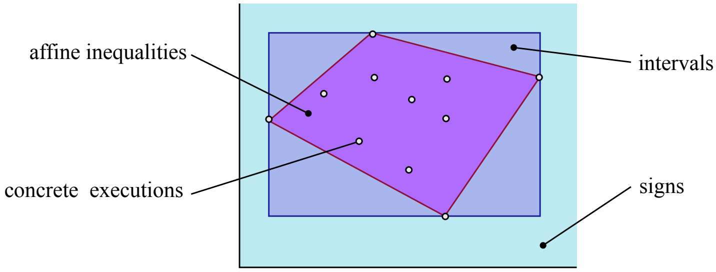


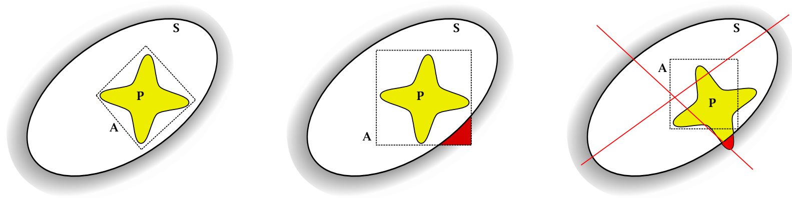
Figure 1.5: A set of points abstracted using affine inequalities (dark polyhedron), intervals (lighter rectangle) and signs (light quarter-plane).

1.1.5 Soundness

In the general sense, *soundness* states that whatever the properties inferred by an analysis, they can be trusted to hold on actual program executions. It is a very desirable property of formal methods, and one we will *always* ensure in this tutorial.

In our case, we expect the analysis to output invariants. It must thus contain at least all actual program states, but it may safely contain more. Computing over-approximations is thus our soundness guarantee. Considering over-approximations allows us to check rigorously so-called *safety* correctness specifications, that is, specifications stating that the set of reachable program states is included in a set of *safe* states — in practice, this set is either specified by the user, through explicit assertions, or specified implicitly by the language, such as the absence of arithmetic overflow. The need for over-approximations is intuitive: if the abstraction is included in the specification then, *a forciori*, the set of actual executions is included in the specification. This is illustrated in Fig. 1.6.(a).

If the abstract state computed does not satisfy the specification, however, the analysis is inconclusive. Either the program is actually flawed, or the program is correct but the abstraction over-approximates its behavior too coarsely for the analysis to prove it. This last case, called *false alarm*, is depicted in Fig. 1.6.(b). Handling this case requires either some investigation of the alarm, either manually or employing some other formal method, or running the analysis again with more



precise analysis	false alarm	unsound analysis
$A \subseteq S \implies P \subseteq S$	$A \not\subseteq S$ but $P \subseteq S$	$A \subseteq S$ but $P \not\subseteq S$
(a)	(b)	(c)

Figure 1.6: Proving that a program P satisfies a safety specification S, i.e., that $P \subseteq S$, using an abstraction A of P: (a) succeeds, (b) fails with a false alarm, and (c) is not a possible configuration for a sound analysis.

precise abstractions. All the analyses we discus here are sound: the case where the program does not satisfy its safety specification while the analysis reports no specification violation, illustrated in Fig. 1.6.(c), will never occur.

1.2 Scope and Applications

This tutorial focuses on sound static analysis based on Abstract Interpretation in order to infer numeric invariants. For the sake of a pedagogical presentation, we analyze a simple toy-language missing many features from real-life languages, such as: functions, arrays, pointers, dynamic memory allocation, objects, exceptions, etc. We refer the reader to other publications and tool presentations, such as [Bertrane et al., 2015], to explain how to adapt the ideas presented here to the analysis of real-life languages and software. Nevertheless, in this section, we justify the interest of numeric invariants by showing analysis applications that are based on, or parameterized with, numeric abstractions. As programs manipulate, at their core, numbers, it is natural to think about numeric abstractions as a key component in most value-sensitive program analyses.

Full text available at: http://dx.doi.org/10.1561/2500000034

(a)

	`int delay[10], i;`
	`i = 0;`
	`while (1) {`
$\langle\, i \in [0,9] \,\rangle$	`    int y = delay[i];`
$\langle\, i \in [0,9] \,\rangle$	`    delay[i] = input();`
$\langle\, i+1 \in [-2^{31}, 2^{31}-1] \,\rangle$	`    i = i + 1;`
	`    if (i >= 10) i = 0;`
	`}`

(b)

	`int delay[10], i;`
	`i = 0;`
$\wr\, i = 0 \,\int$	`while (1) {`
$\wr\, i \in [0,9] \,\int$	`    int y = delay[i];`
$\wr\, i \in [0,9] \,\int$	`    delay[i] = input();`
$\wr\, i \in [0,9] \,\int$	`    i = i + 1;`
$\wr\, i \in [1,10] \,\int$	`    if (i >= 10) i = 0;`
	`}`

Figure 1.7: A C-like program manipulating an array annotated with: (a), correctness verification conditions implied by the language; and (b), invariants inferred by an interval static analysis.

1.2.1 Safety Verification

Figure 1.7.(a) gives an example program together with the verification conditions it must satisfy at various program locations in order to be free from arithmetic overflows and out-of-bound array accesses. These conditions can be derived easily and purely mechanically from the syntax of the program, and they have a purely numeric form.

Figure 1.7.(b) shows the invariants inferred at these points by a static analysis based on intervals. The invariants clearly imply the verification conditions. Hence, the program is free from the errors we target. As we have employed an interval analysis, and the verification conditions can be expressed exactly as intervals, checking the conditions can be done without leaving the abstract world of intervals.

1.2.2 Pointer Analysis

Numeric invariants are not only useful to analyze numeric variables, but also any variable with a numeric aspect. Consider the program in

```
float* p = q;                 unsigned off_p = off_q;
for (i = 0; i < 10; i++)      for (i = 0; i < 10; i++)
    if (...) p++;                 if (...) off_p += 4;
```

$\wr\, \mathit{off}_q \leq \mathit{off}_p \leq \mathit{off}_q + 4\,i + 4 \,\wr$

(a) (b)

Figure 1.8: A C-like program manipulating a pointer p (a) and its translation into a numeric program manipulating its offset off_p (b). Program (b) also shows the numeric invariants inferred on off_p.

Fig. 1.8.(a) employing pointer arithmetic on a pointer p to traverse data in a loop. We can view a pointer value as a pair composed of a variable, and a numeric offset counting a number of bytes from the first byte of the variable — offset 0. Pointer arithmetic will only operate on the offset part, and in a way similar to integer arithmetic. We can transform this program into a purely numeric program operating on synthetic offset variables, such as off_p, instead of pointers, as shown in Fig. 1.8.(b). We can then apply a standard numeric static analysis to infer numeric invariants on offsets. On the example of Fig. 1.8.(b), an affine inequalities analysis would find a relation between the pointer offset and the loop counter i.

Some information about pointer alignment, namely the fact that the offset is a multiple of 4, is missing, because it cannot be represented using affine inequalities. We will see, in Sect. 4.8, a *congruence abstraction* that solves this issue. In fact, each inference problem, for each required property can be solved by designing some adapted abstraction. Finally, note that, in practice, a numeric analysis is combined with a non-numeric points-to analysis [Balakrishnan and Reps, 2004, Miné, 2006b] that infers the first component of pointer values, i.e., the identity of the variables the pointers may point into.

Another, related class of analyses is that of C strings, for instance the analyses by Dor et al. [2001] or by Simon and King [2002]. In this case, a string buffer and a pointer into such a buffer are translated into purely numeric synthetic variables. In addition to offset variables, we need to insert instrumentation variables tracking the position of the first occurrence of the null character (i.e., the string length) and the

```
cell *x, *head = NULL;
for (i = 0; i < n; i++) {
    x = alloc();
    x->next = head; head = x;
}
for (i = 0, x = head; x; x = x->next, i++) {
    ⦃ ∀k ∈ [0, i − 1] : a[k] = head(->next)^k->data ⦄
    a[i] = x->data;
}
⦃ ∀k ∈ [0, n − 1] : a[k] = head(->next)^k->data ⦄
```

Figure 1.9: A C-like program manipulating a linked list and an array, annotated with non-uniform invariants stating a relation between the contents of the array at position k and the list at the same position k.

number of bytes available until the end of the buffer. We also need to modify the program to update them. Using a relational analysis, such as affine inequalities, allows inferring non-trivial relationships, such as a relation between the lengths of the strings used as arguments and return in a string concatenation function such as `strcat`.

1.2.3 Shape Analysis

Beyond pointer analyses, *shape analyses* are a sophisticated family of analyses targeting programs with dynamic memory allocation and recursive data-structures, such as lists or trees. Such analyses also benefit from instrumenting numeric quantities to discuss about, for instance, list length or tree height. Additionally, a *non-uniform* analysis, as proposed by Venet [2004], is able to express properties that distinguish between different instances of a recursive data-structure. Figure 1.9 presents an application to the allocation of a linked list followed by a copy from an array into the list. The loop invariant states that, at loop step i, the $k-$th element of the linked list, pointed to by $head(\texttt{->}next)^k\texttt{->}data$, equals $a[k]$. This very symbolic logic predicate is complemented by the numeric invariant $0 \leq k \leq i - 1$, which restricts the predicate to elements at indices up to i. This numeric invariant can be inferred using the numeric abstractions presented in this tutorial.

```
cost = 0;
for (i = 0; i < n-1; i++) {
    ⦃ cost = i × n − i × (i + 1)/2 ⦄
for (j = i+1; j < n; j++) {
        ⦃ cost = i × (n − i) × (i + 1)/2 + j − i − 1 ⦄
        if (tab[i] > tab[j]) swap(tab[i],tab[j]);
        cost = cost+1;
    }
}
⦃ cost = (n + 1) × (n − 2)/2 ⦄
```

Figure 1.10: A sorting algorithm, with an instrumentation variable, *cost*, added to help compute the time complexity.

1.2.4 Cost Analysis

Numeric invariants do not necessarily refer to quantitative information on the memory state, but can also refer to quantitative information about execution traces, such as their length. This provides some information about the time complexity of the program. One prime example is the Costa analyzer, introduced by Albert et al. [2007].

Figure 1.10 shows a very simple method for obtaining such a bound: the program is instrumented with a synthetic variable, named *cost*, which is incremented at each step. A numeric invariant analysis can then be used to infer properties on *cost*, including an upper bound which is symbolic in the arguments of the function, thanks to a relational analysis. Note that the invariants here are far more complex than those we encountered before as they are not affine, but polynomial. In this tutorial, we will limit ourselves to affine invariants, which are generally not sufficient for cost analyses, but are much simpler and can be inferred more efficiently.

Another, related application is proving termination. Classic termination proofs require finding a decreasing ranking function that is bounded below, and numeric properties can help with that [Urban and Miné, 2014].

(a)	(b)	(c)
`x = input(`$[-10,10]$`)`	$\wr\, x \in [-10,10] \,\wr$	$\wr\, \perp \,\wr$
`if (x == 0) z = 0;`		
`else {`	$\wr\, x \in [-10,10] \,\wr$	$\wr\, x = 0 \,\wr$
`    y = x;`	$\wr\, x \in [-10,10], y \in [-10,10] \,\wr$	$\wr\, y = 0 \,\wr$
`    if (y < 0) y = -y;`	$\wr\, x \in [-10,10], y \in [0,10] \,\wr$	$\wr\, y = 0 \,\wr$
`    z = x / y;`	$\wr$ division by zero $\wr$	
`}`		

Figure 1.11: A program (a); the result of a forward analysis (b); and the result of a backward analysis assuming a division by zero (c).

1.2.5 Backward Analysis

We return to purely numeric properties and intervals to show another flavor of analysis, which goes backward. Instead of inferring the value of variables by propagating forward an abstract memory state from the beginning of the program, an analysis can start from a program point of interest and an abstract property on the memory state, and go backward to derive necessary conditions so that the executions reach the given program point satisfying the given abstract state property. In fact, backward analysis is most often used in combination with a preliminary forward analysis, to refine and focus its results. This scheme is developed for instance by Bourdoncle [1993a].

Figure 1.11.(a) shows a simple C program that divides x by its absolute value $y = |x|$. As the division is guarded by the test `x == 0`, there is no division by zero. Figure 1.11.(b) annotates the program with the result of an interval analysis, starting from $x \in [-10, 10]$. As the interval domain cannot represent $[-10, 10] \setminus \{0\}$, it cannot exploit the fact that $x \neq 0$, and so, $y \neq 0$, when the division `x / y` occurs. The analysis outputs an alarm, which is actually a false alarm. To help the user reason about this alarm, a backward analysis is performed starting just before the error, at the division, with the erroneous state $y = 0$. This state is propagated backward, in the interval domain. We deduce, in particular, that $x = 0$ must hold just after the test `x == 0` has returned false. Propagating backward one more step, the analysis infers that there is no possible program state, denoted here as $\perp$. In

our case, the backward analysis has proved automatically that the error is spurious. In more complex cases, the analysis would simply find a restriction of the state space that would help the user, or another formal method, decide whether the alarm is false or justified.

In the rest of the tutorial, all our examples concern forward analyses to infer invariants. Nevertheless, backward analyses are very similar, and require only a few additional operators.

1.3 Outline

This chapter provided an informal introduction to numeric invariant inference and its applications. The rest of the tutorial will present inference methods in a rigorous way, based on the theory of Abstract Interpretation.

Chapter 2 presents the mathematical tools that will be needed in our formal presentation, including a short course on Abstract Interpretation. Chapter 3 presents our target programming language: a toy language tailored to illustrate numeric invariants. It presents not only the language syntax, but also its concrete semantics in a mathematical, unambiguous way. It then presents how abstractions can be applied to derive an effective static analysis that is sound with respect to the concrete world: we state the operators and hypotheses required on the abstraction, and then develop an analysis that is fully parametric in the choice of the abstraction. Chapters 4 and 5 present two families of such abstractions: firstly, non-relational domains, including signs, constants, intervals, and congruences; secondly, relational domains, including affine equalities, affine inequalities, and weakly relational domains (zones, octagons, and templates). Chapter 6 discusses abstract domain combiners that improve the precision of existing domains: firstly, the reduced product, a technique to combine two or more existing abstractions and design a more expressive analyzer in a modular way; secondly, three methods that improve the precision of a given abstraction by allowing it to express symbolic disjunctions (powerset completion, state partitioning, and path partitioning). To close this tutorial, Chap. 7 provides concluding remarks.

Naturally, we devote a large amount of time presenting the data-structures and algorithms necessary to implement effectively these abstractions in a static analyzer, and we discuss their relative merits in terms of precision, cost, and expressiveness. Each chapter ends with bibliographical notes recalling major articles the reader is invited to consult to complete this necessarily superficial survey.

1.4 Further Resources

To end our introduction, we list additional resources available on-line that can be used as a complement to this tutorial.

For an informal introduction to Abstract Interpretation and links to selected technical resources — including articles, slides, and video presentations — we refer the reader to Patrick Cousot's web-page.[1]

This tutorial is based on several Master-level courses, at École Normale Supérieure, Paris 6, and Paris 7 Universities in France.[2] A programming project focusing on the development, in OCaml, of a simple static analyzer for numeric properties on a toy-language, not unlike the language studied here, is also available.[3] We also refer the reader to Master-level courses by Patrick Cousot at MIT[4] and at Marktoberdorf Summer School.[5]

Implementations of numeric static analyses are also available. The Interproc analyzer[6] is a simple, open-source numeric analyzer on a toy-language, for educational and scientific demonstration purposes. It demonstrates the use of some of the abstract domains we will present in this tutorial: intervals, linear equalities, linear inequalities, and octagons. It additionally features backward and modular inter-procedural analyses, which we will not present formally here. Its most notable feature is that it can be used on-line, through a web interface. The Apron

[1] `http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`

[2] Course slides in English are available at: `https://www-apr.lip6.fr/~mine/enseignement/mpri/2016-2017/`

[3] English version available at: `https://www-apr.lip6.fr/~mine/enseignement/l3/2015-2016/project`

[4] `http://web.mit.edu/16.399/www/`

[5] `http://www.di.ens.fr/~cousot/Marktoberdorf98.shtml`

[6] `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`

library[7] [Jeannet and Miné, 2009], on which Interproc is based, is an open-source library implementing classic numeric domains; it can be used in static analysis projects. Industrial-strength commercial static analyzers include the Astrée analyzer for C [Bertrane et al., 2010], which was used to analyze the run-time errors in avionics software. Evaluation versions are freely available from AbsInt.[8] Julia[9] is a commercial static analyzer for Java. Frama-C[10] [Cuoq et al., 2012] is an open-source program analyzer for C incorporating Abstract Interpretation.

[7] `http://apron.cri.ensmp.fr/library/`

[8] `http://www.absint.com/astree`

[9] `https://www.juliasoft.com/`

[10] `https://frama-c.com/`

References

E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java bytecode. In *Proc. of the 16th European Symposium on Programming*, LNCS, pages 157–172. Springer, 2007.

G. Amato and F. Scozzari. The abstract domain of parallelotopes. *Electronic Notes in Theoretical Computer Science*, 287:17–28, 2012. Proceedings of the Fourth International Workshop on Numerical and Symbolic Abstract Domains, NSAD 2012.

C. Ancourt, F. Coelho, and F. Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science*, 267(1):3 – 16, 2010. Proceeding of the Second International Workshop on Numerical and Symbolic Abstract Domains: NSAD 2010.

R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *Static Analysis: Proceedings of the 9th International Symposium*, volume 2477 of *LNCS*, pages 213–229. Springer, 2002.

R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. In *Proc. of the 5h Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2477 of *LNCS*, pages 135–148. Springer, 2004.

R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1–2):28–56, October 2005a.

R. Bagnara, E. Rodríguez-Carbonell, and E. Zaffanella. Generation of basic semi-algebraic invariants using convex polyhedra. In *Static Analysis: 12th International Symposium, SAS 2005*, pages 19–34. Springer, 2005b.

R. Bagnara, P. M. Hill, and E. Zaffanella. An improved tight closure algorithm for integer octagonal constraints. In *Verification, Model Checking and Abstract Interpretation: Proceedings of the 9th International Conference (VMCAI 2008)*, volume 4905 of *LNCS*, pages 8–21. Springer, 2008a.

R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008b.

R. Bagnara, P. M. Hill, and E. Zaffanella. Weakly-relational shapes for numeric abstractions: Improved algorithms and proofs of correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.

R. Bagnara, P. M. Hill, and E. Zaffanella. Exact join detection for convex polyhedra and other numerical abstractions. *Computational Geometry: Theory and Applications*, 43(5):453–473, 2010.

G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. of the Int. Conf. on Compiler Construction (CC'04)*, number 2985 in LNCS, pages 5–23. Springer, 2004.

V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *ACM PLDI'89*, pages 41–53. ACM Press, 1989.

F. Banterle and R. Giacobazzi. A fast implementation of the octagon abstract domain on graphics hardware. In *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings*, pages 315–332. Springer, 2007.

C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 14(4):605–624, 2003.

C. Bartzis and T. Bultan. Widening arithmetic automata. In *Computer Aided Verification, 16th International Conference, CAV*, volume 3114 of *LNCS*, pages 321–333. Springer, 2004.

F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revisiting hull and box consistency. In *Proc. of the 16th Int. Conf. on Logic Programming*, pages 230–244, 1999.

F. Benoy, A. King, and F. Mesnard. Computing convex hulls with a linear solver. *Theory and Practice of Logic Programming*, 5(1–2):259–271, 2005.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385 in AIAA, pages 1–38. AIAA (American Institute of Aeronautics and Astronautics), April 2010.

J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends in Programming Languages (FnTPL)*, 2(2–3):71–190, 2015.

G. Birkhoff. Lattice theory. In *Colloquium Publications*, volume 25. Amer. Math. Soc., 3. edition, 1967.

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of the ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'03)*, pages 196–207. ACM, June 2003.

M. Bouaziz. TreeKs: A functor to make numerical abstract domains scalable. In *4th International Workshop on Numerical and Symbolic Abstract Domains (NSAD 2012)*, volume 287, pages 41–52. Elsevier, September 2012.

F. Bourdoncle. Abstract interpretation by dynamic partitioning. *J. Funct. Program.*, 2(4):407–423, 1992.

F. Bourdoncle. Abstract debugging of higher-order imperative languages. *SIGPLAN Not.*, 28(6):46–55, June 1993a.

F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications (FMPA'93)*, volume 735 of *LNCS*, pages 128–141. Springer, June 1993b.

R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35:677–691, 1986.

R. M. Burstall. Program proving as hand simulation with a little induction. *Information Processing*, pages 308–312, 1974.

L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *Proc. of the Sixth Asian Symp. on Programming Languages and Systems (APLAS'08)*, volume 5356 of *LNCS*, pages 3–18. Springer, December 2008.

L. Chen, A. Miné, J. Wang, and P. Cousot. Linear absolute value relation analysis. In *Proc. of the 20th European Symp. on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 156–175. Springer, March 2011.

N. V. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282 – 293, 1968.

C. K. Chiu and J. H. M. Lee. Efficient interval linear equality solving in constraint logic programming. *Reliable Computing*, 8(2):139–174, April 2002.

E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8:244–263, 1986.

E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

A. Cortesi and M. Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.

A. Cortesi, G. Filé, F. Ranzato, R. Giacobazzi, and C. Palamidessi. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19(1):7–47, January 1997.

A. Cortesi, G. Costantini, and P. Ferrara. A survey on product operators in abstract interpretation. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday*, pages 325–336, 2013.

A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *Computer Aided Verification: 17th International Conference, CAV 2005*, pages 462–475. Springer, 2005.

P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Res. rep. R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, September 1977. 15 p.

P. Cousot. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

P. Cousot. Types as abstract interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM Press.

P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.

P. Cousot. Abstracting induction by extrapolation and interpolation. In *Verification, Model Checking, and Abstract Interpretation: 16th International Conference, VMCAI 2015*, pages 19–42. Springer, 2015.

P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. of the 2d Int. Symp. on Programming*, pages 106–130. Dunod, Paris, France, 1976.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, January 1977.

P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conf. Rec. of the 6th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'79)*, pages 269–282. ACM Press, 1979a.

P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979b.

P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In *Proc. of the Int. Workshop on Programming Language Implementation and Logic Programming (PLILP'92)*, volume 631 of *LNCS*, pages 269–295. Springer, 1992a.

P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992b.

P. Cousot and R. Cousot. "à la Burstall" intermittent assertions induction principles for proving inevitability properties of programs. *Theoret. Comput. Sci.*, 120(1):123–155, 1993.

P. Cousot and R. Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. In *Logics and Languages for Reliability and Security*, NATO Science Series III: Computer and Systems Sciences, pages 1–29. IOS Press, 2010.

P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. of the 5th Annual ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In *Proc. of the 11th Annual Asian Computing Science Conf. (ASIAN'06)*, volume 4435 of *LNCS*, pages 272–300. Springer, December 2006.

P. Cousot, R. Cousot, and L. Mauborgne. A scalable segmented decision tree abstract domain. In *Pnueli Festschrift*, volume 6200 of *LNCS*, pages 72–95. Springer, 2010.

R. Cousot. Reasoning about program invariance proof methods. Res. rep. CRIN-80-P050, Centre de Recherche en Informatique de Nancy (CRIN), Institut National Polytechnique de Lorraine, Nancy, France, July 1980.

P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. In *Proc. of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247. Springer, 2012.

E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Static Analysis: 8th International Symposium, SAS 2001 Paris, France, July 16–18, 2001 Proceedings*, pages 194–212. Springer, 2001.

J. Feret. Static analysis of digital filters. In *Proc. of the 13th European Symp. on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 33–48. Springer, March 2004.

J. Feret. The arithmetic-geometric progression abstract domain. In *Proc. of the 6th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 42–58. Springer, January 2005.

G. Filé and F. Ranzato. The powerset operator on abstract interpretations. *Theoretical Computer Science*, 222(1):77–111, 1999.

R. W. Floyd. Assigning meanings to programs. In *Proc. of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–32, Providence, USA, 1967.

G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Interval analysis and machine arithmetic: Why signedness ignorance is bliss. *ACM Trans. Program. Lang. Syst.*, 37(1):1:1–1:35, January 2015.

K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain Taylor1+. In *Proc. of the 21st Int. Conf. on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 627–633. Springer, June 2009.

R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In *Automata, Languages and Programming: 24th International Colloquium, ICALP '97*, pages 771–781. Springer, 1997.

R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Science of Computer Programming*, 32(1):177–210, 1998.

R. Giacobazzi and F. Scozzari. A logical model for relational abstract domains. *ACM Trans. Program. Lang. Syst.*, 20(5):1067–1109, September 1998.

R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, March 2000.

P. Granger. Static analysis of arithmetic congruences. *Int. Journal of Computer Mathematics*, 30:165–199, 1989.

P. Granger. Static analysis of linear congruence equalities among variables of a program. In *Proc. of the Int. Joint Conf. on Theory and Practice of Soft. Development (TAPSOFT'91)*, volume 493 of *LNCS*, pages 169–192. Springer, 1991.

P. Granger. Improving the results of static analyses of programs by local decreasing iterations. In *Foundations of Software Technology and Theoretical Computer Science: 12th Conference*, pages 68–79. Springer, 1992.

P. Granger. Static analyses of congruence properties on rational numbers (extended abstract). In *Static Analysis: 4th International Symposium, SAS '97*, pages 278–292. Springer, 1997.

N. Halbwachs and J. Henry. When the decreasing sequence fails. In *Static Analysis: 19th International Symposium, SAS 2012*, pages 198–213. Springer, 2012.

M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis: 5th International Symposium, SAS'98*, pages 200–214. Springer, 1998.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

ISO/IEC JTC1/SC22/WG14 working group. C standard. Technical Report 1124, ISO & IEC, 2007.

B. Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, 23 (1):5–37, July 2003.

B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 661–667. Springer, June 2009.

G. Kahn. Natural semantics. Technical Report 601, INRIA, 1987.

M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6: 133–151, 1976.

G. Kildall. A unified approach to global program optimization. In *Proc. of the 1st Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL'73)*, pages 194–206. ACM, 1973.

J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, 1976.

S. C. Kleene. *Introduction to metamathematics*. Bibliotheca mathematica. North-Holland Pub. Co., 1964.

S. Lang. *Introduction to Linear Algebra*. Undergraduate Texts in Mathematics. Springer, 1997.

H. LeVerge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Science of Computer Programming*, 75(9):796–807, 2010.

A. Maréchal, D. Monniaux, and M. Périn. Scalable minimizing-operators on polyhedra via parametric linear programming. In *Static Analysis - 24th International Symposium*, pages 212–231, 2017.

I. Mastroeni. Algebraic power analysis by abstract interpretation. *Higher-Order and Symbolic Computation*, 17(4):297–345, December 2004.

L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In *Proc. of the 14th European Symp. on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 5–20. Springer, April 2005.

K. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proc. of the Second Symposium on Programs as Data Objects (PADO II)*, volume 2053 of *LNCS*, pages 155–172. Springer, May 2001.

A. Miné. Relational abstract domains for the detection of floating-point runtime errors. In *Proc. of the European Symp. on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 3–17. Springer, March 2004.

A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006a.

A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of the ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63. ACM, June 2006b.

A. Miné. Abstract domains for bit-level machine integer and floating-point operations. In *Proc. of the 4th Int. Workshop on Invariant Generation (WING'12)*, number HW-MACS-TR-0097 in EPiC Series in Computing, page 16. Computer Science, School of Mathematical and Computer Science, Heriot-Watt University, UK, June 2012.

R. E. Moore. *Interval Analysis.* Prentice Hall, Englewood Cliffs N. J., USA, 1966.

M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *Proc. of the 14th European Symp. on Prog. (ESOP'05)*, volume 3444 of *LNCS*, pages 46–60. Springer, April 2005.

D. Nguyen Que. *Robust and generic abstract domain for static program analysis: The polyhedral case.* PhD thesis, École des Mines de Paris, 2010.

H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. *SIGPLAN Not.*, 47(6):229–238, June 2012.

S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proc. of the 11th Int. Conf. on Automated Deduction (CADE'92)*, volume 607 of *LNAI*, pages 748–752. Springer, June 1992.

G. D. Plotkin. A structural approach to operational semantics, 1981.

W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Commun. of the ACM*, 8:4–13, August 1992.

H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54 – 75, 2007.

S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. of the 6th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 21–47. Springer, 2005.

D. Schmidt. Abstract interpretation from a topological perspective. In *Static Analysis: 16th International Symposium, SAS 2009*, pages 293–308. Springer, 2009.

P. Schrammel and B. Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In *Static Analysis: 18th International Symposium, SAS 2011*, pages 233–248. Springer, 2011.

A. Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., 1986.

D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. Technical Report PRG-6, Oxford U. Computing Lab, 1971.

Y. Seladji. Finding relevant templates via the principal component analysis. In *Verification, Model Checking, and Abstract Interpretation, VMCAI 2017*, pages 483–499. Springer, 2017.

A. Simon and A. King. Analyzing string buffers in C. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, AMAST '02, pages 365–379. Springer, 2002.

A. Simon and A. King. Exploiting sparsity in polyhedral analysis. In *Proc. of the 12th Int. Symp. on Static Analysis (SAS'05)*, volume 3672 of *LNCS*, pages 336–351. Springer, September 2005.

A. Simon and A. King. Taming the wrapping of integer arithmetic. In *Proc. of the 14th Int. Symp. on Static Analysis (SAS'07)*, volume 4634 of *LNCS*, pages 121–136. Springer, August 2007.

A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In *Proc. of the 12th Int. Conf. on Logic based program synthesis and transformation (LOPSTR'02)*, volume 2664 of *LNCS*, pages 71–89. Springer, 2002.

G. Singh, M. Püschel, and M. Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 46–59. ACM, 2017.

A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.

A. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, 1949.

C. Urban and A. Miné. A decision tree abstract domain for proving conditional termination. In *Proc. of the 21st International Static Analysis Symposium (SAS'14)*, volume 8373 of *LNCS*, pages 302–318. Springer, September 2014.

A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *Proc. of the Int. Symp. on Static Analysis (SAS'04)*, number 3148 in LNCS, pages 149–164. Springer, 2004.