



STATIC ANALYSIS BY PATH FOCUSING

Julien Henry

M2R Internship Report

2011

e-mail:

Julien.Henry@imag.fr

Tutors:

DAVID MONNIAUX - MATTHIEU MOY

Unité Mixte de Recherche 5104 CNRS - Grenoble-INP - UJF

Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
<http://www-verimag.imag.fr>



STATIC ANALYSIS BY PATH FOCUSING

Julien Henry

Grenoble-INP

2011

Abstract

Program verification aims at statically discovering properties on programs, such as the values that can take the different variables during execution. Abstract Interpretation is a technique that computes an over-approximation of the set of these values, since it is impossible to compute the real set in general. This report takes place in the many attempts to improve the precision of static Analysis by Abstract Interpretation. It proposes a technique that takes benefit of SMT-solving to obtain more precise results at reasonable cost.

Résumé

La vérification de programme consiste à découvrir statiquement des propriétés sur des programmes, comme l'ensemble des valeurs que peuvent prendre les variables durant l'exécution. L'Interprétation Abstraite est une technique permettant de calculer une approximation de cet ensemble, le véritable ensemble étant impossible à calculer en général. Ce rapport s'inscrit dans la lignée des travaux visant à améliorer la précision de l'analyse par interprétation abstraite, et propose une technique tirant parti du SMT-solving pour obtenir de meilleurs résultats à un coût raisonnable.

Keywords: Static Program Analysis, Verification, Abstract Interpretation, Invariant Generation, Path Focusing, SMT-Solving

Mots Clés: Analyse Statique de Programmes, Vérification, Interprétation Abstraite, Génération d'invariants, Découverte de Chemins, SMT-Solving

Tutors: David Monniaux - Matthieu Moy

ACKNOWLEDGEMENTS

I would like to thank my two tutors, David Monniaux and Matthieu Moy, for their direction of my master thesis, and for their help everytime I needed it. I thank them for all their explanations that clarified lots of my questions, and for their advices for the redaction of this report.

I also thank Laure Gonnord for inviting me to present my work at the GDR GPL 2011.

Finally, I thank all the Synchrone team and the Verimag laboratory for their cheerful welcome.

Contents

1	Introduction	4
2	Abstract Interpretation: state of the art	6
2.1	Introductory example	6
2.2	Abstract Interpretation	7
2.2.1	Abstraction of the domain	7
2.2.2	Termination	8
2.3	Linear Relation Analysis	9
2.3.1	Convex polyhedra	9
2.3.2	Program Analysis	10
2.3.3	Precision of the analysis	11
2.4	Example	13
3	Path Focusing technique	15
3.1	Lookahead Widening	15
3.2	Path Focusing	16
3.2.1	Multigraph	16
3.2.2	Choice of the focus paths	19
3.2.3	Algorithm	22
3.2.4	Precision of the analysis	22
3.2.5	Example	22
3.2.6	Disjunctive invariants	25
3.2.7	Removing identity transitions	27
3.3	Efficiency comparison: example	27
4	Implementation	29
4.1	Infrastructure	29
4.1.1	LLVM internal representation	29
4.1.2	Transformation passes	30
4.1.3	Drawbacks	31
4.2	Abstract domain representation	31
4.2.1	Attachment to LLVM Internal Representation	31
4.2.2	Dimensions of the abstract values	32
4.2.3	Diseq comparisons	33
4.3	Unrolling loops	33
4.4	SMT-solving	35

4.5	Limitations	35
4.6	Example	35
4.7	Experiments	36
4.7.1	Benchmarks	36
4.7.2	Analysis of real code	37
5	Future Work	41
5.1	Arithmetic Overflows	41
5.2	Alias analysis	42
5.3	Combining Lookahead Widening and Path Focusing	42
6	Conclusion	43
	Bibliography	47
	Appendices	48
A	Generated ρ formula	49
B	Example of loss of precision in Path Focusing	50

Chapter 1

Introduction

Static analysis aims at automatically computing properties on programs, such as the possible values of their variables during execution. This allows to show for instance that a program will not overflow, will not divide by zero, and to compute loop invariants. . . The main applications of static analysis are compile-time optimisations, and the proof of safety properties in critical systems, such as avionics.

Abstract Interpretation is a general framework used for static analysis. Linear Relation Analysis (LRA) is a direct application of this framework, that computes an upper-approximation of the set of the possible states of a numerical program. The state of a program is defined by the position of the program counter in the code, and the current values of the different numerical variables. The set of possible states is expressed as a least fixpoint of a set of equations defining the semantics of the program.

A fundamental fact in static analysis is that it cannot be perfectly precise: either the analysis is unsound (the set of possible states we compute does not contain all the possible states during execution), or is incomplete (the computed set contains states that can never be reached during execution). Abstract Interpretation is sound but incomplete, since it always computes an upper-approximation of the set of possible states. The challenge is then to be as precise as possible, so that we can prove properties about the program.

Linear Relation Analysis over-approximates the set of possible states as a convex polyhedron where the dimensions are the numerical variables of the program. Instead of convex polyhedra, one could choose intervals, octagons, etc, that are subclasses of the class of convex polyhedra. The fixpoint equation is computed iteratively; that is, successive approximations of the set of reachable states are computed until they converge to a fixpoint.

Satisfiability Modulo Theory (SMT) solving is a technique for deciding the satisfiability of a logic formula containing Boolean predicates and elements of a theory, such as linear arithmetic relations between integers. In this report, we present a novel refinement of the Abstract Interpretation technique, that uses SMT-solving to guide the fixpoint iterations, so that it temporarily focuses on a selected path in the control flow graph (the graph expressing the semantics of the program), in order to obtain more precise results at reasonable cost.

Contribution and Organisation

In this report we make the following contributions:

- We present a new technique, referred as *Path Focusing*, that takes benefits of the recent

advances in the field of SMT-solving to improve precision of static analysis by abstract interpretation. We also present an adaptation of this technique to compute disjunctive invariants.

- We present an implementation of this technique into a small analyser, as well as the implementation of another technique proposed in [GR06], such that we can compare precision and cost of these two methods. This implementation has been tested on a wide range of programs, including small test-cases as well as real-life programs.

The report is organised as follows: Part 2 introduces the state of the art in Abstract Interpretation and more specifically in Linear Relation Analysis. Part 3 describes the two techniques *Lookahead Widening* and *Path Focusing*. Part 4 presents the implementation of these two techniques into an analyser, and the experimental results we obtained. Finally, Part 5 introduces some directions to explore.

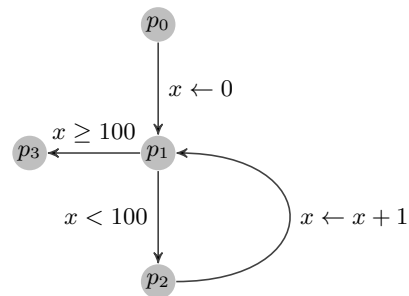
Chapter 2

Abstract Interpretation: state of the art

2.1 Introductory example

We consider the following program:

```
x = 0;
while (x < 100) {
    x++;
}
```



Static analysis is aimed at discovering some properties about programs. In this example, we would like to compute the set of possible values for the variable x during execution. The graph at the right side represents the program. The nodes of the graph are called *program points*. We can compute for instance the set Y_1 of values for x at point p_1 . This set depends on the sets Y_0 and Y_2 , since there are two edges arriving at p_1 : the first one comes from p_0 , and the second one from p_2 .

There is a relation between Y_1 and Y_2 :

$$\begin{aligned}
 Y_1 &= \{x \mid x = 0 \vee \exists x' \in Y_2, x = x' + 1\} \\
 Y_2 &= \{x \mid x \in Y_1 \wedge x < 100\}
 \end{aligned}$$

We see that Y_1 and Y_2 can be computed step by step: we start with $Y_1 = Y_2 = \emptyset$. Then, we add iteratively new elements in the sets.

- We see that $0 \in Y_1$: $Y_1 = \{0\}$.
- Y_1 contains 0. So, Y_2 also contains 0 by definition: $Y_2 = \{0\}$.
- Y_2 contains 0, so $0 + 1 = 1$ is in Y_1 : $Y_1 = \{0, 1\}$.
- Again, we find $Y_2 = \{0, 1\}$.

- Y_2 now contains 1, so $1 + 1 = 2 \in Y_1$: $Y_1 = \{0, 1, 2\}$.
- We continue the iteration until we find no more new elements in the sets...

This computation is a fixpoint computation: the different sets are updated until we find no more elements. These sets are strictly increasing (we always add new elements in the sets), and if we find no new elements to add, that means we have reached a limit. Abstract Interpretation gives a general framework to do such kind of fixpoint computation.

2.2 Abstract Interpretation

Abstract Interpretation [CC77, CC92a] is a general method for finding approximate solutions of fixpoint equations. This method is used for program analysis, since analysing a program often comes down to solving fixpoint equations, because of the loops and recursive procedures.

However, most of the time, the solution of this fixpoint equation must be computed in a complex domain: in program analysis, this could be the state space of the program, i.e the set of all possible states of the program. This computation quickly becomes too costly or does not terminate.

2.2.1 Abstraction of the domain

Abstract Interpretation proposes to represent more efficiently the elements of this complex domain C of concrete values, by choosing a simpler domain A called *abstract domain*.

A concretization function γ maps the elements of A to elements of C :

$$\begin{aligned} \gamma : A &\longrightarrow C \\ x &\longmapsto \gamma(x) \end{aligned}$$

The concrete semantics is a function $\Phi : C \rightarrow C$. In the introductory example, this function Φ is the following:

$$\Phi \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \end{bmatrix} = \begin{bmatrix} x \mid & x \in \mathbb{R} \\ x \mid & x = 0 \vee \exists x' \in Y_2, x = x' + 1 \\ x \mid & x \in Y_1 \wedge x < 100 \\ x \mid & x \in Y_1 \wedge x \geq 100 \end{bmatrix}$$

One can choose an abstract semantics $\Phi^\# : A \rightarrow A$ satisfying this condition:

$$\forall x \in A, \Phi \circ \gamma(x) \subseteq_C \gamma \circ \Phi^\#(x)$$

$\Phi^\#$ is then an abstraction of Φ , that gives an upper approximation of $\gamma(x)$ when composed with γ .

Proposition 2.2.1 *Suppose C is a complete lattice, and Φ is increasing from C to C . Each $x \in A$, satisfying the condition $\Phi^\#(x) \subseteq x$, is an abstraction of the least fixpoint of Φ .*

Indeed, if $\Phi^\#(x) \subseteq x$, then $\Phi \circ \gamma(x) \subseteq \gamma \circ \Phi^\#(x) \subseteq \gamma(x)$. Abstract interpretation aims at computing such an x . It computes the stationary limit of an ascending sequence $(x_i)_{i \geq 0}$ defined by the induction:

$$\begin{cases} x_0 &= \perp \\ x_{n+1} &= x_n \sqcup \Phi^\#(x_n) \end{cases}$$

where \perp is the least element of A , and \sqcup is an operator verifying $x_1 \cup x_2 \subseteq x_1 \sqcup x_2$ and $x_1 \sqcup x_2 \in A$.

2.2.2 Termination

Termination of the fixpoint computation has to be guaranteed. This termination depends on the properties of the abstract domain: this one should be finite, of finite height, or more generally satisfy the *ascending chain condition*, meaning that there does not exist any sequence $(x_i)_{i \geq 0}$ of elements of the abstract domain A such that $\forall i \geq 0, x_i <_A x_{i+1}$. Indeed, otherwise, the least fixpoint computation could run indefinitely, since the ascending sequence of elements of A is infinite, and the fixpoint is never reached.

Many of the domains that are used for program analysis, including those of intervals and convex polyhedra, do not satisfy this ascending chain condition. To ensure convergence in this case, another approximation is performed: a new operator is defined, called *widening operator*, that extrapolates the limit of a sequence of abstract values [CC77, CC92b]. This *widening operator* is usually noted $\nabla : A \times A \rightarrow A$, and satisfies the following properties:

- $\forall x_1, x_2 \in A, x_1 \leq_A x_1 \nabla x_2$ and $x_2 \leq_A x_1 \nabla x_2$. This guarantees the correctness of the result. Most of the time, this operator is only defined for $x_1 \leq x_2$. In this case, we use $x_1 \nabla (x_1 \sqcup x_2)$ instead.
- For any increasing sequence $x_0 \leq_A x_1 \leq_A \dots$, the sequence defined by

$$\begin{cases} x'_0 &= x_0 \\ x'_{i+1} &= x'_i \nabla x_{i+1}, \quad \forall i \geq 0 \end{cases}$$

is not strictly increasing. Then, applying the widening operator when the sequence may increase indefinitely makes the computation converge to a fixpoint in finite time.

[Mon09] gives a more general definition of the widening operator.

The least fixpoint of a function $\Phi^\#$ is noted \bar{x} . Instead of computing this least fixpoint, one compute an upper approximation of it, by computing the following ascending sequence:

$$\begin{cases} x'_0 &= \perp \\ x'_{i+1} &= x'_i \nabla (x'_i \sqcup \Phi^\#(x'_i)), \quad \forall i \geq 0 \end{cases}$$

which converges towards \tilde{x} , where $\tilde{x} \geq_A \bar{x}$.

\tilde{x} is a correct upper approximation of the least fixpoint of $\Phi^\#$, and the classical technique is to regain some precision lost by the widening operator by computing a descending sequence:

$$\begin{cases} x''_0 &= \tilde{x} \\ x''_{i+1} &= \Phi^\#(x''_i), \quad \forall i \geq 0 \end{cases}$$

Each element of this descending sequence is still an upper approximation of the least fixpoint \bar{x} . So, this sequence often allows to find approximate least fixpoints that are more precise than the one obtained after the ascending sequence.

2.3 Linear Relation Analysis

Linear Relation Analysis [CH78] (also denoted LRA) is a direct application of Abstract Interpretation. It is aimed at computing an upper approximation of the reachable states of a program containing numerical variables. The set of possible assignments for the numerical variables is abstracted by a convex polyhedron. This technique discovers invariant linear relations between the numerical variables at each control point of a program.

This technique is:

- *sound*: Each possible assignment for the variables in the real program is included in the abstract value.
- *incomplete*: The abstract value also contains assignments for the variables that are not possible in the real program.

2.3.1 Convex polyhedra

Let x_1, x_2, \dots, x_n be the numerical variables of a program (assume they are all in \mathbb{Q}). A state of the program is then a point $\vec{x} \in \mathbb{Q}^n$.

In \mathbb{Q}^n , the set of polyhedra ordered by inclusion is a lattice. The least element is \perp (the empty polyhedron), and the greatest element is \top (the whole \mathbb{Q}^n).

The classical union of two convex polyhedra may not be a convex polyhedron. Therefore, we use the convex hull operation, noted \sqcup , for joining polyhedra. If X_1, X_2 are two convex polyhedra, $X_1 \sqcup X_2$ is the smallest convex polyhedron containing both X_1 and X_2 .

Since the domain of convex polyhedra is of infinite height, a widening operator is defined to ensure convergence of the technique. Intuitively, the polyhedron $P \nabla Q$ is obtained after removing from the system of linear equations defining P all the inequalities that are not satisfied by Q (the actual definition is somewhat more involved, see e.g. [BHZ05]). For instance, if $P = \{(x, y) \mid 0 \leq x \leq y \leq 1\}$ and $Q = \{(x, y) \mid 0 \leq x \leq y \leq 2\}$, then the result of the widening operator will be $P \nabla Q = \{(x, y) \mid 0 \leq x \leq y\}$.

There have been various work proposing a definition or a refinement of the widening operator in LRA [CH78, Hal79, HPR97, BCC+03], in order to fight against loss of precision:

- *Delayed widening*: instead of applying widening at the first iteration of the analysis, one apply it after a number n of iterations. During the $n - 1$ first iterations, only a convex hull is computed. In some cases, delaying the widening operation is more precise. The algorithm still terminates, since widening is applied within a finite time after n iterations. In most of the cases, n can be chosen equal to 2.
- *Widening with threshold (or “up to”)*: In some cases, instead of brutally applying the widening operator, a fixed set of linear inequalities T can be chosen, and the widening operator ∇_T is defined as follows: $P \nabla_T Q$ is the intersection of $P \nabla Q$ with all the inequalities in T satisfied by both P and Q .

For instance, consider the program:

```
while (x <= 50) {
    x++;
    y++;
}
```

If $P = \{(x, y) \mid 0 \leq x \leq y \leq 1\}$, after one iteration of the loop, we have $Q = \{(x, y) \mid 0 \leq x \leq y \leq 2\}$. Instead of a classic widening (which gives the result $P \nabla Q = \{(x, y) \mid 0 \leq x \leq y\}$), a widening with the threshold $x \leq 50$ gives $P \nabla_T Q = \{(x, y) \mid 0 \leq x \leq y \leq 50\}$. In such cases, widening with threshold helps to find better invariants.

2.3.2 Program Analysis

Throughout the report, a program will be represented by a control flow graph:

- P is a finite set of control points.
- I_p is the (possibly empty) set of initial values for each control point $p \in P$. For an initial control point p , i.e a starting point of the program, I_p is not empty. It is empty in the other case.
- $E \subseteq P \times P$ is a set of directed edges. Each edge $e \in E$ has a semantics $\tau_e : \mathcal{P}(\mathbb{Q}^n) \rightarrow \mathcal{P}(\mathbb{Q}^n)$, $\mathcal{P}(\mathbb{Q}^n)$ being the set of possible values of \vec{x} . τ_e maps a set of states before the transition to the set of states after the transition.

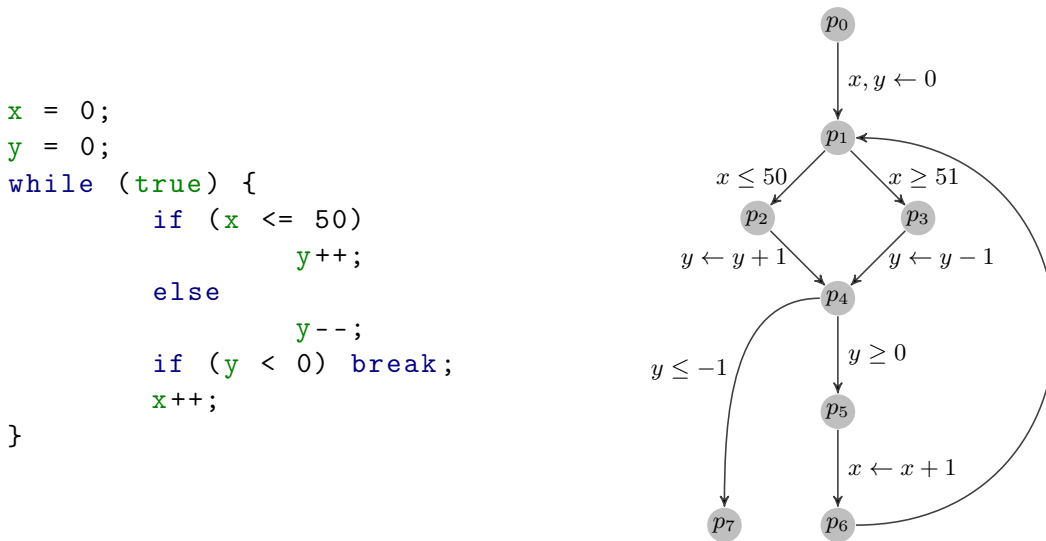
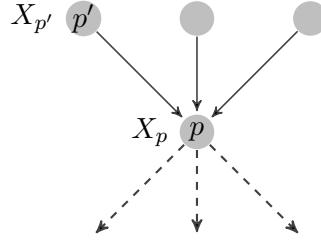


Figure 2.1: Example of program, and its associated control flow graph. This program comes from [GR06].

To each state $p \in P$ of the control flow graph, we associate an abstract value $X_p \in D$, D being in our case the domain of convex polyhedra over \mathbb{Q}^n . Since the exact operation τ may not be expressible in the abstract domain, we abstract it into $\tau^\#$ such that $\forall X \in D, \tau(X) \subseteq \tau^\#(X)$.

The computation is aimed at finding a solution for this system of abstract semantic inequalities:

$$\left[\begin{array}{l} \forall p \in P, I_p \subseteq X_p \\ \forall (p', p) \in E, \tau_{(p', p)}^\#(X_{p'}) \subseteq X_p \end{array} \right.$$



The function $\Phi^\#$ previously described is the following:

$$\Phi^\# \begin{bmatrix} \vdots \\ X_p \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ I_p \sqcup \bigsqcup_{(p',p) \in E} \tau^\#(X_{p'}) \\ \vdots \end{bmatrix}$$

The fixpoint computation algorithm consists in replacing iteratively each X_p by its value on the right hand side, until convergence.

In the case of an abstract domain with infinite ascending sequence, we use the previously defined widening operator:

$$\begin{bmatrix} \vdots \\ X_p \\ \vdots \end{bmatrix} \leftarrow \begin{bmatrix} \vdots \\ X_p \nabla \left(I_p \sqcup \bigsqcup_{(p',p) \in E} \tau^\#(X_{p'}) \right) \\ \vdots \end{bmatrix}$$

After a finite number of steps, the computation has reached the fixpoint and the value X_p at the end gives various linear relations between the different numerical variables of the program at the control point p .

2.3.3 Precision of the analysis

In order to guarantee the termination of the computation, there is no need to apply the widening operator at each control point $p \in P$. It is sufficient to apply it on a subset of P , called P_w , such that removing the nodes of P_w disconnects every cycles. For instance, in structured programs, P_w can be chosen as the set of the loop headers.

Iteration strategies

There exist several iteration strategies for computing the fixpoint: whatever order we choose for updating the different $X_p, p \in P$, the result at the end of the analysis is correct. Yet, the precision of the result and the time before convergence can be very different. There have been some work to find efficient iteration strategies [Bou92], such as first stabilising innermost loops, or stabilising strongly connected components of the graph.

Sources of imprecision

Linear Relation Analysis is sound but incomplete: there is loss of precision during the computation, due to various reasons:

1. Since an abstract domain is used to represent the set of possible states, such as the domain of convex polyhedra, there is an upper approximation of the set of possible states to the smallest convex polyhedron including this set. This loss of precision is unavoidable. Still, there are techniques proposing to compute disjunctive invariants [GZ10] to limit this upper approximation: at each program point, a union of convex polyhedra is computed instead of a single one.
2. The widening operator, used to ensure convergence of the technique, also induces a loss of precision. The classical technique is then to compute narrowing iterations, which often recover some precision.

For instance, considering the program:

```
for (int i=0; i < 100; i++) {
}
```

The analysis starts with $i = 0$. After one iteration, $i \in [0, 1]$, then $i \in [0, 2]$ after the second iteration. . . . After a widening operation, the result becomes $i \in [0, +\infty]$, which is an invariant, but very imprecise. We can do a new iteration, with $i \in [0, +\infty]$ at the beginning of the loop, and we find that the values for i at the next step are in $[0, 100]$. This is still an invariant, but much better. We can repeat such iterations as long as the set narrows, and we always obtain a correct over-approximation. In practise, a single narrowing iteration is enough to recover the precision.

3. There is also a loss of precision each time a point in the control flow graph has several incoming edges. This is the case for instance for *if* statements, loops, etc. In this case, the abstract value of this point is a convex hull of several convex polyhedra, inducing an upper approximation. Indeed, if X_1, X_2 are convex polyhedra,

$$X_1 \cup X_2 \subseteq X_1 \sqcup X_2$$

To limit the number of such points, a method could be to expand the control flow graph. For instance, instead of considering a graph with a sequence of n *if-then-else* between points p_1 and p_n , the n merge node of these *if-then-else* could be removed, each of them having two incoming edges, and 2^n edges from p_1 to p_n are added, corresponding to the different paths through the *if-then-else* statements. At the end, p_n is the only point with several incoming edges. This graph transformation results in an exponential blowup, because of the exponential growth of the number of edges in the graph.

Acceleration

Typically, to analyse a program with loops, the approach is to use the widening operator, for example at the head of the loops, to ensure convergence. Since this induces imprecision, another technique called *acceleration* is aimed at computing the exact effect of the loop when possible [Gon07, GH06]. For instance, for the following program, an invariant could be directly computed for the loop:

```

x = 0;
y = 0;
while (x < N) {
    y = y+2;
    x = x+5;
}
    
```

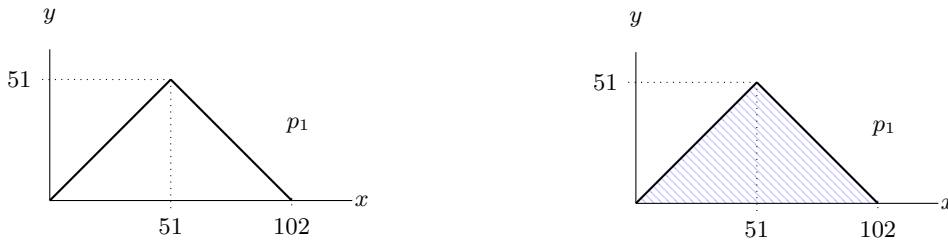
$$\begin{aligned}
 &\exists k \geq 0, && x = k \\
 &\wedge y = 2k \\
 &\wedge \forall k', (0 \leq k' \leq k) \Rightarrow (5k' \leq N)
 \end{aligned}$$

Acceleration is possible when the loop is simple enough, and comes to computing the transitive closure τ_e^+ of τ_e , the transition function associated to the loop.

2.4 Example

We apply the linear relation analysis technique over the program in figure 2.1. The set of control state where we apply widening is $\{p_1\}$: it is sufficient, since removing p_1 cuts all cycles in the graph.

To each point of the control flow graph $\{p_0, p_1, \dots, p_7\}$, we attach a convex polyhedron, whose dimensions are the numerical variables x, y . For instance, at point p_1 , the effective set of possible assignments for (x, y) is the union of the two segments $(0, 0) - (51, 51)$ and $(51, 51) - (102, 0)$, depicted in figure 2.2.a. Since this union of segments is not a convex polyhedron, the best result we can wait for after abstraction is the polyhedron depicted in figure 2.2.b, which is the smallest convex polyhedron containing the two segments.



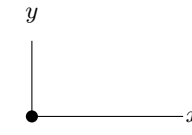
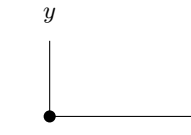
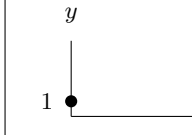
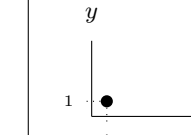
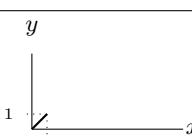
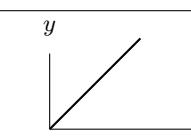
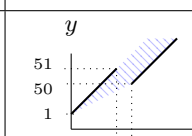
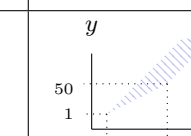
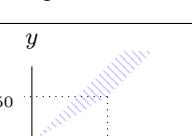
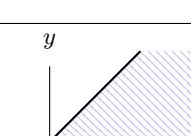
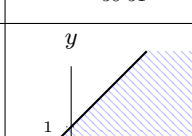
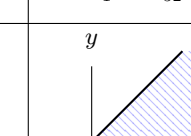
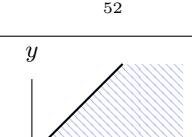
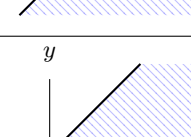
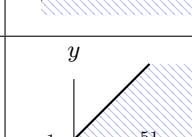
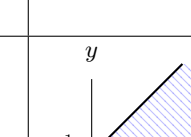
- a. During execution, (x, y) can only be in one of these two segments. b. The best result we can have is the polyhedron in hashed lines.

Figure 2.2: Best results we can get from the analysis

We apply abstract interpretation to this program. Let X_i be the polyhedron attached to point p_i . Initially, $X_i = \perp$ for each $i \in \{1, 7\}$, and $X_0 = \top$. The tabular in figure 2.3 shows the evolution of the polyhedra during the iterations.

After 3 iterations, we have reached an invariant. We can then apply a narrowing iteration, that recovers some precision. Finally, at point p_1 , we see that the obtained result is not as precise as we could wish (Figure 2.2.b). This is due to the widening operator applied at point p_1 , and the convex hull applied at point p_4 .

Figure 2.3: Polyhedron attached to the different points during iterations.

Iteration	$X_0 \sqcup X_6$	X_1 (widening)	X_4	X_6
1				
2				
3				
narrowing				

Chapter 3

Path Focusing technique

In this part, we present two techniques that aim to isolate the different paths of the program to compute more precise invariants.

3.1 Lookahead Widening

In some cases, a loop can have a non-regular behaviour, especially when it has several paths. Some of these paths can stay impossible for a while, and become possible after a certain number of iterations. However, the widening operator extrapolates without taking care of these new possible paths. This can induce a loss of precision, that *Lookahead widening* tries to limit.

Lookahead widening [GR06] is a technique that isolates the different loop phases during the analysis, and stabilises each of these loop phases before proceeding to the next. One only consider paths that are feasible at the first iteration, and forget the others for a while. The iterations end when the analysis of these paths has converged. After that, some narrowing iterations can be computed. Then, paths that become feasible are reinserted into the control flow graph, and we redo iterations.

In the program already depicted in figure 2.1, there is a loop with two perfectly distinct phases:

- During the 51 first iterations, both x and y are incremented.
- During the 51 last iterations, x is incremented and y is decremented.

When computing the fixpoint iterations, we only consider a subset of the graph for a while, until the convergence. In our example:

- Step 1: we only consider the path of the loop that is possible at the first iteration. The second path is ignored, and will be treated later. We compute the iterations, and we obtain an first invariant for this part of the graph: at point p_1 , $x = y \wedge x \geq 0$. Then, we can do some narrowing steps in order to recover some precision, before adding new paths in the graph. At point p_1 after narrowing, $x = y \wedge 0 \leq x \leq 51$.
- Step 2: Now, we add the new paths into our graph if they have become possible. In our case, the second path of the loop is now possible, because x can be equal to 51, so we add the path to the graph and compute new iterations until convergence... After

convergence, at point p_1 , $x + y - 102 \leq 0 \wedge x \leq y$. We then apply a narrowing step and at p_1 , $x + y - 102 \leq 0 \wedge x \leq y \wedge y \geq 0$.

- Step 3: finally, the path from p_4 to p_7 becomes possible, so we add it into the graph and compute the invariant for p_7 .

Finally, at p_1 , the result we obtain is the best we can have using convex polyhedra (see Figure 2.2.b). This method allows to find the constraint $x + y - 102 \leq 0$, which is not the case in the classical abstract interpretation technique.

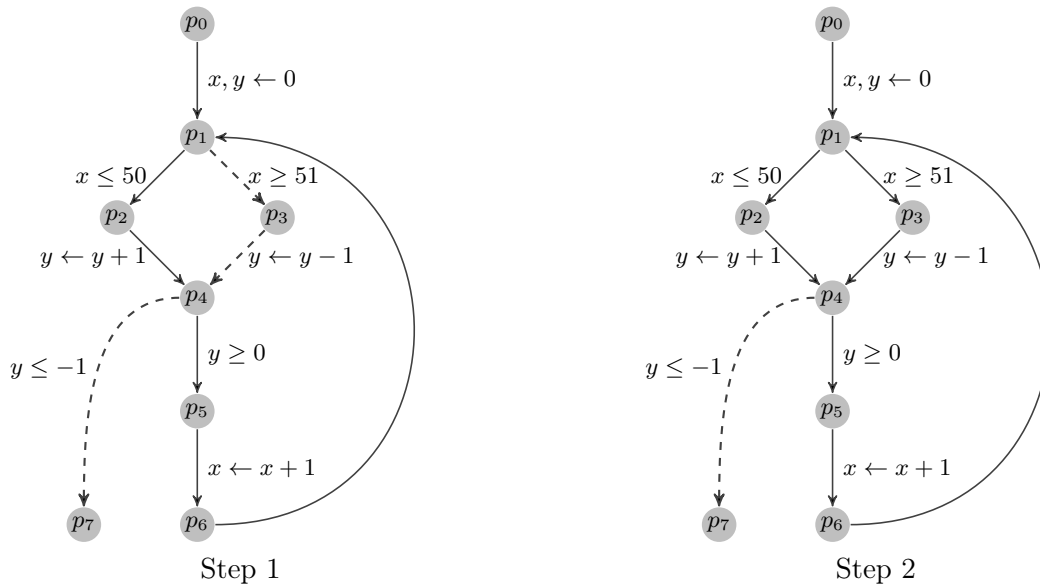


Figure 3.1: Dotted arrows are ignored by the analysis.

In the classical abstract interpretation technique, narrowing iterations are computed at the end of the analysis, when all the fixpoint computation has converged. The interest of this method is to apply narrowing iterations right after each path computation. This allows to recover precision before analysing new paths, and then to be more precise.

3.2 Path Focusing

In this section, we present a refinement of Abstract Interpretation, referred as *Path Focusing* technique [MG11].

3.2.1 Multigraph

Expanding the control flow graph

The main idea of the technique is to compute the fixpoint iterations on an expanded multigraph instead of the classical control flow graph. A multigraph is a graph that can have several edges from a point p to a point q .

Intuitively, expanding the graph comes back to consider independently the different paths in the control flow graph, and thus to be more precise, because paths that are not feasible will

be ignored. Additionally, the number of control points with several incoming transitions will be reduced, so the loss of precision due to the convex hull of polyhedra may be minimised.

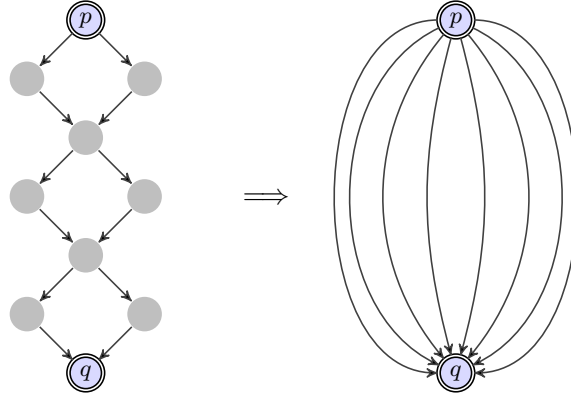


Figure 3.2: One can expand the graph in the left and obtain the associated graph in the right.

Let (P, E) be the control flow graph of the program. First, one chooses a set of widening points $P_W \subseteq P$, so that the graph obtained after removing these points has no cycle. The widening operator will be applied at these points. The choice of P_W can be guided by [Bou92]. Then, another set $P_R \subseteq P$ of nodes is chosen, satisfying the following properties:

- $I_p = \emptyset$ for each $p \in P \setminus P_R$, meaning that the initial points of the program are included in P_R .
- $P_W \subseteq P_R$: all the widening points are in P_R .

The multigraph is then the graph such that:

- each element of P_R is a point of the graph.
- for each path $p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_k$ in the control flow graph, such that $p_1 \in P_R, p_k \in P_R$, there is a transition in the multigraph from p_1 to p_k with the semantics

$$\tau_{p_1 \rightarrow \dots \rightarrow p_k} = \tau_{p_{k-1} \rightarrow p_k} \circ \dots \circ \tau_{p_1 \rightarrow p_2}$$

where $\tau_{p_i \rightarrow p_{i+1}}$ is the transformation associated to the semantics of $p_i \rightarrow p_{i+1}$.

As explained in 2.3.3, separating the different possible paths between p and q reduce the number of points with a join operation, and results in a better precision. This can result in an exponential blowup in the size of the graph. To avoid it, the technique is to never construct the multigraph, and only compute parts of it when needed.

Intuitively, choosing a small set P_R will result in a better precision, but a higher cost. Indeed, the number of paths between two points of P_R will grow if P_R contains few elements. A good choice for P_R requires to find a good compromise between precision and cost of the technique.

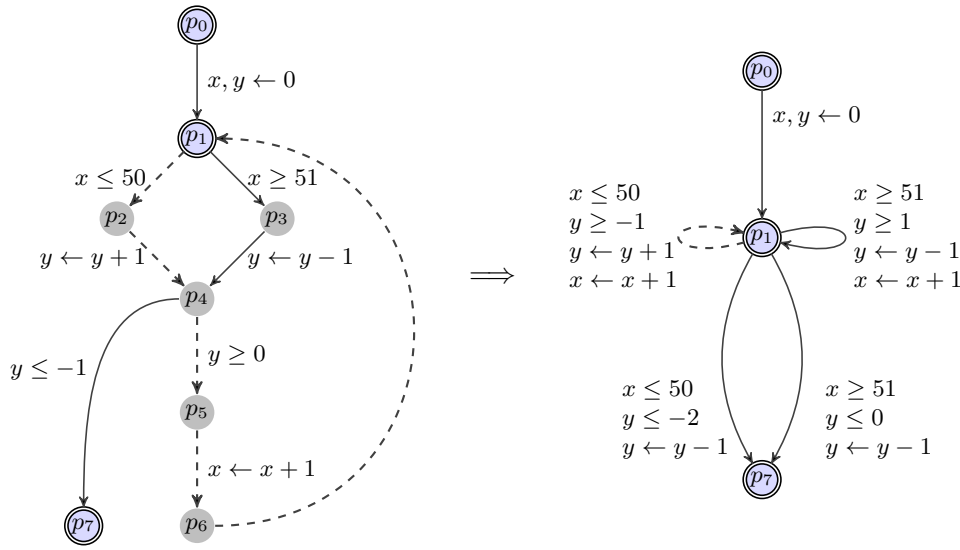


Figure 3.3: Example of classical control flow graph, and its associated expanded multigraph, where $P_R = \{p_0, p_1, p_7\}$. The path dashed from p_1 to p_1 in the control flow graph corresponds to a simple dashed transition in the multigraph, having the same semantics.

Static Single Assignment form

Path focusing technique uses control flow graphs in static single assignment (SSA) form. The main concept behind the SSA form is that, syntactically, every variable is only being assigned once.

$$\begin{array}{l}
 x = 0; \\
 y = 0; \\
 x = x + 2; \\
 y = x + 1;
 \end{array}
 \quad \Rightarrow \quad
 \begin{array}{l}
 x.0 = 0; \\
 y.0 = 0; \\
 x.1 = x.0 + 2; \\
 y.1 = x.1 + 1;
 \end{array}$$

Figure 3.4: On the left side, a simple program. On the right side, the same program in SSA form.

This single definition property cannot be achieved with only the renaming of variables assigned most than once. In some cases, two distinct definition reach the same use, depending on the actual execution. To solve this issue, SSA form uses Φ -functions.

A Φ -function is usually inserted at points of the control flow graph that have several incoming transitions, and has the same number of arguments as the point has incoming transitions. Assuming these transitions are ordered, the Φ -function returns the value of its i -th argument if the point is reached from the i -th transition.

```

if (c) x = 0;
else x = 1;

y = x+1;

```

 \implies

```

if (c) x.0 = 0;
else x.1 = 0;

x.2 = Phi (x.0, x.1)
y.0 = x.2 + 1;

```

This static single assignment form is used for the computation of the focus path.

3.2.2 Choice of the focus paths

Reachability problem

Path focusing technique tries to discover which paths to focus on for the computation of the fixpoint iterations. Since the multigraph is not constructed explicitly, the different paths starting and finishing on node in P_R are not known. The technique proposes to express the focus path as the solution of a satisfiability modulo theory (SMT) problem:

“Is there a path starting on control point p , with numerical variables in X_p , that ends on point q , with variables that are not in X_q ?”

This problem could be rephrased as: “Is there a path that starts on p , that can make the fixpoint computation progress ?” Indeed, the idea is to only focus on paths that make the abstract values grow.

This is a reachability problem, that can be expressed as an SMT-formula, which is a logic formula with Boolean variables, and elements of a certain theory T . Depending on the program, different theories may be used:

- For programs with rational variables, whose operations (instructions, transition conditions. . .) are all linear arithmetic, T can be the theory of linear real arithmetic (LRA).
- For programs with integer variables (with a numerical state space in \mathbb{Z}^n) T can be the theory of linear integer arithmetic (LIA).

Although deciding the satisfiability of such formula is NP-complete, there has been much research on decision procedures [KS08] and there exist nowadays efficient programs, known as SMT-solvers, that can decide the satisfiability of LRA or LIA formulae. Well-known efficient SMT-solvers are Z3 [dMB08] and Yices [DdM06].

If the problem has a solution, the SMT-solver will be in position to show which path in the control flow graph is selected, by giving a model: an assignments of the Boolean variables so that the formula is true. Among these Boolean variables, some of them are associated to a transition in the control flow graph:

- if the model has set to *true* the Boolean variable b_e associated to the transition e , then the focus path goes through this transition.
- in the other case, the focus path does not go through this transition.

Some others are associated to the points of the control flow graph. These are called *reachability predicates*, and are set to *true* in the model if and only if the focus path goes through these points.

Construction of the formula

The first point of the Path Focusing technique is to compute the SMT formula ρ expressing the semantics of the program. For doing so, all the operations in the program have to be expressible within the theory we choose. Otherwise, the SMT-solver would not be able to decide if the formula is *sat* or not.

Since we want to find a path starting in a control point in P_R and ending in a point in P_R , we need to disconnect in the formula the points p_i in P_R into a source point p_i^s , with only outgoing transitions, and a destination point p_i^d , with only incoming transitions.

We construct the ρ formula as follows:

1. Each numerical SSA variable of the control flow graph has its definition in the ρ formula.
2. For each operation in the program, we encode its semantics, expressed in terms of the Boolean and numerical SSA variables. For instance, if the CFG contains the assignment $x.2 = x.1 + 1$, then we simply add $x.2 = x.1 + 1$ to the formula. This equality expression is noted $assign(x.2)$. $assign(x)$ is defined for each numerical SSA-variable of the program.
3. For each transition $(p_i, p_j) \in E$, we set $t_{i,j} = B_i \wedge c(i, j)$, where $B_i = b_i^s$ if $p_i \in P_R$, or b_i if not. $c(i, j)$ expresses the condition that needs to be true to go through the transition. For instance, $c(i, j)$ could be of the form $x < y, \dots$. If the transition is non-deterministic, then $c(i, j) = c_{i,j}$, where $c_{i,j}$ is a Boolean variable left non-deterministic.
4. For each point $p_i \notin P_R$, we set the reachability predicate $b_i = \bigvee_{(p_j, p_i) \in E} t_{j,i}$.
5. For each point $p_i^d \in P_R$, we set the reachability predicate $b_i^d = \bigvee_{(p_j, p_i) \in E} t_{j,i}$.
6. Numerical Φ -variables assignments have to be expressed in the ρ formula. We simply use ite (*if-then-else*) statements, a standard feature of the SMT-Lib format [BST10], for giving the right value to the variable, depending on the incoming transition we come from. Generally, for a Φ -variable $x = \Phi(x_1, x_2, \dots, x_n)$ in control point p_i with the ordered incoming transitions $t_{j_1, i}, t_{j_2, i}, \dots, t_{j_n, i}$, we set:

$$x = \text{ite } (t_{j_1, i}) (x_1) (\text{ite } (t_{j_2, i}) (x_2) (\text{ite } \dots))$$

where $\text{ite } (c) (x) (y)$ is equal to x if $c = \text{true}$, else y .

If the point where the Φ -variable x is defined is in P_R , x needs to be renamed into x' :

- x will be the value of x at the beginning of the path.
- x' will be the value of x at the end of the path.

Indeed, there can be uses of x along the path, and these uses refer to the old value of x .

Finally, the ρ formula is the following:

$$\rho = \bigwedge_{p_i \notin P_R} \left[\left(b_i = \bigvee_{(p_h, p_i) \in E} t_{h,i} \right) \wedge \left(\bigwedge_{(p_i, p_j) \in E} t_{i,j} = b_i \wedge c(i, j) \right) \right] \wedge$$

$$\bigwedge_{p_i \in P_R} \left[\left(b_i^d = \bigvee_{(p_h, p_i) \in E} t_{h,i} \right) \wedge \left(\bigwedge_{(p_i, p_j) \in E} t_{i,j} = b_i^s \wedge c(i, j) \right) \right] \wedge$$

$$\bigwedge_{x \in \Sigma} \text{assign}(x)$$

where Σ is the set of the numerical SSA-variables of the program.

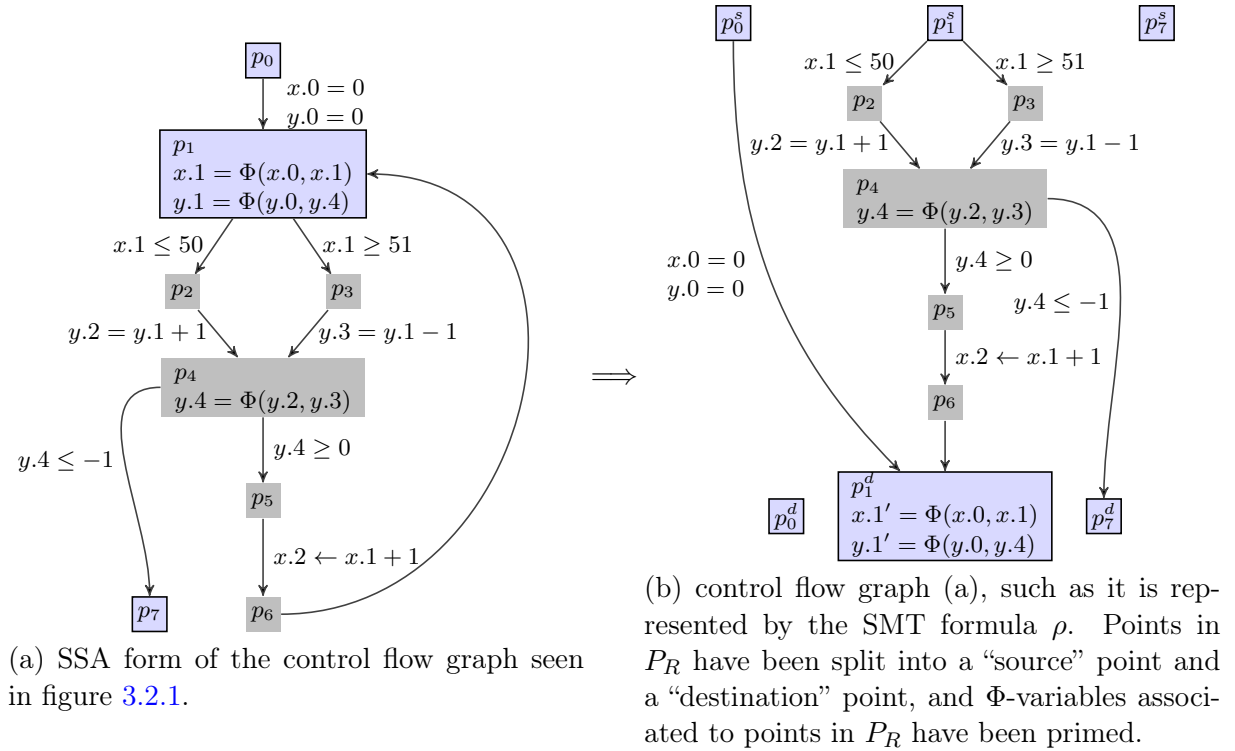


Figure 3.5: Example (cont'd)

Expression of the reachability problem

The SMT-formula ρ express the semantics of the program. We still need to create the formula expressing the reachability problem defined in 3.2.2.

The abstract domain D we use is those of convex polyhedra, so an abstract value $X \in D$ is simply a system of linear inequalities between the different SSA-variables of the program. Then, it is easy to encode the property $x \in X$ into an SMT-formula: one simply writes the conjunction of these inequalities, with the name of the variables in the vector x .

To compute the reachability starting from point p_i , the formula given to the SMT-solver is:

$$\rho \wedge b_i^s \wedge \bigwedge_{j \neq i, j \in P_R} (\neg b_j^s) \wedge x_i \in X_{p_i} \wedge \bigvee_{j/(p_i, p_j) \in E} (b_2^d \wedge \neg(x'_j \in X_{p_j}))$$

where x_i is the vector of the SSA-variables at the beginning of the path, and x'_j is the vector of the SSA-variable at the end: in this second vector, Φ -variables defined in p_j are primed.

- we search a path starting in p_i : b_i^s has to be *true*, and all other source points are *false*.
- the vector x_i of the numerical variables at point p_i is in our abstract value X_{p_i}
- we search a path arriving in a successor p_j of p_i , such that the vector of the numerical variables x'_j is not yet included in the current abstract value X_{p_j} of p_j . b_j^d is *true*, and the conjunction of the inequalities given by X_{p_j} is *false*.

3.2.3 Algorithm

The Path Focusing algorithm is described in Algorithm 1. We maintain a set A of points in P_R that need to be treated. We select iteratively one of these control points, and search for new focus paths starting from this point, until there is no more path to be treated. At the end of the analysis, some narrowing iterations can be performed in order to recover some precision.

3.2.4 Precision of the analysis

One can refine the *Path-focusing* algorithm by treating specially the self loops of the multigraph. Indeed, instead of simply applying the widening operator, some narrowing steps could be performed to regain precision before focusing on another path. This relates to the ideas in *Lookahead widening* [GR06].

So, the algorithm has a special case if the path we focus on goes from p_i to itself: instead of computing $\tau_e^\#(X_{p_i})$, one compute *SelfLoop*($\tau_e^\#, X_{p_i}$), defined in algorithm 2.

The *SelfLoop* procedure actually computes a widening operation, and computes a narrowing step in order to recover precision. In some cases, instead of performing widening/narrowing iterations, acceleration techniques could be used to find even more precise invariants [GH06].

In order for this refinement to be precise, the widening operator needs to be delayed when we consider a self-loop for the first time. Then, our algorithm uses a set U of paths that have already been treated. If the self-loop we focus on is not in U , we apply a simple union instead of widening. Widening will be applied next time to guarantee the convergence.

When the self loop can be accelerated [Gon07], instead of doing a widening/narrowing sequence, acceleration directly finds its transitive closure.

3.2.5 Example

We apply the *Path Focusing* technique to our running example from Figures 2.1, 3.3 and 3.5. The control points in P_R are p_0 , p_1 and p_7 . The ρ formula associated to this program is depicted in Figure A (see appendices). One computes iteratively the associated abstract values X_0 , X_1 and X_7 .

- Step 0: Initially, $X_0 = \top$, $X_1 = \perp$, $X_7 = \perp$, and $A = \{p_0\}$.

Algorithm 1 Path Focusing

```

1: procedure PATHFOCUSING( $P, E, P_R$ )
2:   for all  $p \in P_R$  do
3:     Compute  $Succ(p)$ , the set of the successors of  $p$  in the multigraph
4:   end for
5:    $\rho \leftarrow \text{computeRho}(P, E)$ 
6:    $A \leftarrow \emptyset$ 
7:   for all  $p \in P_R / I_p \neq \emptyset$  do
8:      $A \leftarrow A \cup p$ 
9:   end for
10:  while  $A \neq \emptyset$  do
11:    Select  $p_i \in A$ 
12:     $A \leftarrow A \setminus \{p_i\}$ 
13:    while true do
14:       $res \leftarrow \text{SmtSolve} \left[ \rho \wedge b_i^s \wedge \bigwedge_{\substack{j \neq i \\ j \in P_R}} (\neg b_j^s) \wedge x_i \in X_{p_i} \wedge \bigvee_{\substack{j \\ p_j \in Succ(p_i)}} (b_2^d \wedge \neg(x_j' \in X_{p_j})) \right]$ 
15:      if  $res = \text{unsat}$  then
16:        break
17:      end if
18:      Compute the focus path  $e$  from  $p_i$  to  $p_j$ 
19:       $Y \leftarrow \tau_e^\#(X_{p_i})$ 
20:      if  $p_j \in P_W$  then
21:         $X_{p_j} \leftarrow X_{p_j} \nabla (X_{p_j} \sqcup Y)$ 
22:      else
23:         $X_{p_j} \leftarrow X_{p_j} \sqcup Y$ 
24:      end if
25:       $A \leftarrow A \cup \{p_j\}$ 
26:    end while
27:  end while
28:  Possibly perform some narrowing steps
29:  Compute  $\{X_{p_i}, i \notin P_R\}$ 
30:  return  $\{X_{p_i}, i \in P\}$ 
31: end procedure

```

Algorithm 2 SelfLoop

```

1: procedure SELFLOOP( $\tau_e^\#, X_{p_i}$ )
2:    $Y \leftarrow \tau_e^\#(X_{p_i})$ 
3:    $X' \leftarrow X_{p_i} \nabla (X_{p_i} \sqcup Y)$ 
4:    $Y \leftarrow \tau_e^\#(X')$ 
5:   return  $Y$ 
6: end procedure

```

Algorithm 3 Path Focusing with special treatment for self loops

```

1: procedure PATHFOCUSING( $P, E, P_R$ )
2:   for all  $p \in P_R$  do
3:     Compute  $Succ(p)$ , the set of the successors of  $p$  in the multigraph
4:   end for
5:    $\rho \leftarrow \text{computeRho}(P, E)$ 
6:    $A \leftarrow \emptyset$ 
7:   for all  $p \in P_R / I_p \neq \emptyset$  do
8:      $A \leftarrow A \cup p$ 
9:   end for
10:  while  $A \neq \emptyset$  do
11:    Select  $p_i \in A$ 
12:     $A \leftarrow A \setminus \{p_i\}$ 
13:     $U \leftarrow \emptyset$ 
14:    while true do
15:       $res \leftarrow \text{SmtSolve} \left[ \rho \wedge b_i^s \wedge \bigwedge_{\substack{j \neq i \\ j \in P_R}} (\neg b_j^s) \wedge x_i \in X_{p_i} \wedge \bigvee_{\substack{j \\ p_j \in Succ(p_i)}} (b_2^d \wedge \neg(x'_j \in X_{p_j})) \right]$ 
16:      if  $res = \text{unsat}$  then
17:        break
18:      end if
19:      Compute the focus path  $e$  from  $p_i$  to  $p_j$ 
20:      if  $p_i = p_j$  then
21:         $Y \leftarrow \text{SelfLoop}(\tau_e^\#, X_{p_i})$ 
22:      else
23:         $Y \leftarrow \tau_e^\#(X_{p_i})$ 
24:      end if
25:      if  $(p_j \in P_W) \wedge (p_i \neq p_j \vee e \in U)$  then
26:         $X_{p_j} \leftarrow X_{p_j} \nabla (X_{p_j} \sqcup Y)$ 
27:      else
28:         $X_{p_j} \leftarrow X_{p_j} \sqcup Y$ 
29:         $U \leftarrow U \cup \{e\}$ 
30:      end if
31:       $A \leftarrow A \cup \{p_j\}$ 
32:    end while
33:  end while
34:  Possibly perform some narrowing steps
35:  Compute  $\{X_{p_i}, i \notin P_R\}$ 
36:  return  $\{X_{p_i}, i \in P\}$ 
37: end procedure

```

- Step 1: We start in p_0 . Is there a path starting in p_0 , that arrives in a state in P_R and makes its abstract value grow? The SMT-solver answers *yes*, and gives the path $p_0 \rightarrow p_1$. We update X_1 : it is the polyhedron $x = y = 0$. We update $A = \{p_1\}$. We do another SMT query: is there another path starting in p_0 that makes an abstract value grow? The answer is no.
- Step 2: We now work on p_1 . Is there a path starting in p_1 that arrives in a successor of p_1 , and makes its abstract value grow? Answer is yes, and the model given by the SMT-solver is the path $p_1 \rightarrow p_2 \rightarrow p_4 \rightarrow p_5 \rightarrow p_6 \rightarrow p_1$. Actually, this is the first phase of the loop. The image of $x = y = 0$ by this transition is $x = y = 1$. The new X_1 is then $x = y \wedge 0 \leq x \leq 1$. The path is a self loop, but is seen for the first time, so we do not apply widening yet. So, we redo an SMT-query, that gives us exactly the same path. Then, we can do a widening/narrowing sequence. After widening, we have $x = y \wedge x \geq 0$, and narrowing gives $X_1 = (x = y) \wedge (0 \leq x \leq 51)$.
- Step 3: We still are in p_1 : we ask for another focus path, and the SMT-solver finds the path $p_1 \rightarrow p_3 \rightarrow p_4 \rightarrow p_5 \rightarrow p_6 \rightarrow p_1$. This is the second phase of the loop, which is possible since $(x, y) = (51, 51) \in X_1$. Again, this is a self-loop, but we do not widen yet. The image of X_1 by the transition is the single point $(52, 50)$. X_1 is then the convex hull of this point with the old X_1 . A new SMT query gives the same path, and we can now apply widening/narrowing. We obtain $X_1 = (x \leq y) \wedge (102 - x - y \geq 0)$ after widening, and narrowing gives $X_1 = (x \leq y) \wedge (102 - x - y \geq 0) \wedge y \geq 0$. We can see that X_1 is the smallest polyhedron containing the possible values for (x, y) (see Figure 2.2.b).
- Step 4: Again, we ask the SMT-solver for a new focus path, and it returns $p_1 \rightarrow p_3 \rightarrow p_4 \rightarrow p_7$. The image of the polyhedra X_1 after this path transformation is the single point $y = -1 \wedge x = 102$. We can update $X_7 = (102, -1)$. Path focusing technique gives us the precise result of this program. After that, the SMT-solver cannot find any new path, so the analysis terminates.

3.2.6 Disjunctive invariants

In this subsection, we propose an extension of the path focusing technique to compute disjunctive invariants, in order to improve precision of the analysis.

[GZ10] proposes a technique to compute transitive closure of a loop, i.e compute the invariants for a loop having its semantics expressed by a transition system. The definition of a transition system is the following:

Definition 3.2.1 *A transition system for a control point p is a DNF formula (i.e a disjunction of conjunctions), where each disjunct is the semantics of a path from p to itself. It is of the form*

$$\bigvee_{1 \leq i \leq n} \tau_{p,i}$$

where n is the number of paths, and $\tau_{p,i}$ is the semantics of the i -th path from p to itself.

The technique is aimed at computing a disjunctive invariant for this control point:

$$\bigvee_{1 \leq j \leq m} X_{p,j}$$

where $X_{p,j}$ is a conjunction of linear inequalities, i.e a convex polyhedra.

The principle of the method is to choose an integer $\delta \in \{1, \dots, m\}$, and a mapping function $\sigma : \{1, \dots, m\} \times \{1, \dots, n\} \mapsto \{1, \dots, m\}$. For each polyhedra of the disjunctive invariant, and for each path in the graph, the image of the polyhedra $X_{p,j}$ by the transition $\tau_{p,i}$ is joined with $X_{p,\sigma(j,i)}$.

Algorithm 4 Transitive closure

```

1: procedure TRANSITIVECLOSURE( $\bigvee_{i=1}^m X_{p,i}$ )
2:   for all  $j \in \{1, \dots, m\} \setminus \{\delta\}$  do
3:      $X_{p,j} \leftarrow \perp$ 
4:   end for
5:    $X_{p,\delta} \leftarrow Id$ 
6:   repeat
7:     for  $i \in \{1, \dots, n\}$  and  $j \in \{1, \dots, m\}$  do
8:        $X_{p,\sigma(j,i)} \leftarrow X_{p,\sigma(j,i)} \sqcup \tau_{p,i}(X_{p,j})$ 
9:     end for
10:  until no change in  $\bigvee_{j=1}^m X_{p,j}$ 
11: end procedure

```

There are heuristics to choose m , δ and σ [GZ10, Section 5].

This algorithm requires to enumerate all the paths and to compute their semantics, which may result in an exponential blowup. In addition, the *for* loop at line 7 iterates on all the $n \times m$ values of (i, j) : some of the join operations may be useless, in the sense that the value of $X_{p,\sigma(j,i)}$ may not change. Here, the same technique as Path Focusing is achievable to avoid these drawbacks: SMT-solving is used to find the paths $\tau_{p,i}$.

Algorithm 5 Transitive closure with implicit transition system

```

1: procedure TRANSITIVECLOSUREIMPLICIT( $p$ )
2:   for all  $j \in \{1, \dots, m\} \setminus \{\delta\}$  do
3:      $X_j \leftarrow \perp$ 
4:   end for
5:    $X_\delta \leftarrow Id$ 
6:   while true do
7:      $res \leftarrow SmtSolve \left[ \rho \wedge b_k^s \wedge \bigvee_{j=1}^m (B_j \wedge x \in X_j) \wedge b_k^d \wedge \neg \left( \bigvee_{j=1}^m x' \in X_j \right) \right]$ 
8:     if  $res = unsat$  then
9:       break
10:    end if
11:    Compute  $\tau_{p,i}$  from  $res$ 
12:    Take  $j \in \{k | B_k = true\}$ 
13:     $X_{p,\sigma(j,i)} \leftarrow X_{p,\sigma(j,i)} \sqcup \tau_{p,i}(X_{p,j})$ 
14:  end while
15: end procedure

```

The new algorithm we propose is described in Algorithm 5. A difference is that here, we do not know the number of paths. Then, the mapping function σ has to be defined on $\{1, \dots, m\} \times \mathbb{N}$. We could also easily compute this number of paths when computing the ρ formula.

Computing disjunctive invariants is a field we have to work on. The work described here is only preliminary, and shows it is possible to improve existing techniques with our ideas, so that we avoid explicit enumeration of exponential-sized sets of paths.

3.2.7 Removing identity transitions

When computing iterations, some of the loop paths may not change the state of the variables, i.e the transition function associated to this path is the identity function. This happens in the case of expansive transition relations Φ , defined by the property $X \subseteq \Phi(X)$.

Considering the set of initial states I , and the transition function τ mapping a state to its successor, we can define:

$$\Phi(X) = \{x' \mid \exists x \in X, x' = \tau(x)\} \cup I$$

Φ is the function we classically use to find an inductive invariant X , satisfying the property $\Phi(X) \subseteq X$. To improve the analysis, we propose to compute this invariant using another function Ψ , by removing the transitions that are actually the identity. Indeed, these transitions make no change to the set X . The Ψ function could be expressed as follows:

$$\Psi(X) = \{x' \mid \exists x \in X, x' = \tau(x) \wedge x' \neq x\} \cup I$$

It is easy to prove that $\Psi(X) \subseteq X \Leftrightarrow \Phi(X) \subseteq X$, so we can use Ψ instead of Φ to compute our invariant.

In the case of path focusing algorithm, we simply conjunct our formula (see 3.2.2) with $(i \neq j) \vee (x'_j \neq x_i)$, such that the SMT-solving only proposes paths that are not the identity. Actually, it never focuses on such paths during the ascending sequence, but during narrowing iterations that occur at the end of the algorithm.

Ignoring these identity transitions makes the narrowing sequence recover more precision. Indeed, narrowing sequence updates the invariant X with $\Phi(X)$, which is still an invariant, while precision is recovered, meaning that $\Phi(X) \subset X$. In the case of an expansive function, where $X \subseteq \Phi(X)$, we never have $\Phi(X) \subset X$, so narrowing iterations are useless.

3.3 Efficiency comparison: example

In this subsection, we propose an example showing that *Path Focusing* technique performs better than *Lookahead Widening* in the case of loops with intermediate control-flow merges.

We consider the program depicted in Figure 3.6, page 28. Here, we would like to show that the returned value r is zero.

Lookahead Widening The analysis begins with $p_1 : x = i = r = 0$. Then, the path $p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_5 \rightarrow p_7 \rightarrow p_1$ is the only one possible. The technique is then to stabilise the iterations over this subgraph, removing p_4 and p_6 for a while. After stabilisation and narrowing phases, we obtain $0 \leq i \leq 50 \wedge r = 0 \wedge -1 \leq x \leq 0$ at control point p_1 , and $0 \leq i \leq 49 \wedge r = 0 \wedge x = -1$ at point p_5 .

Then, point p_4 becomes reachable. We continue the iterations and have to do a convex hull of polyhedra at point p_5 , because of the two possible incoming transitions. We obtain the polyhedron $0 \leq i \leq 50 \wedge r = 0 \wedge -1 \leq x \leq 1$. Point p_6 now becomes reachable, since $x = 0$ is

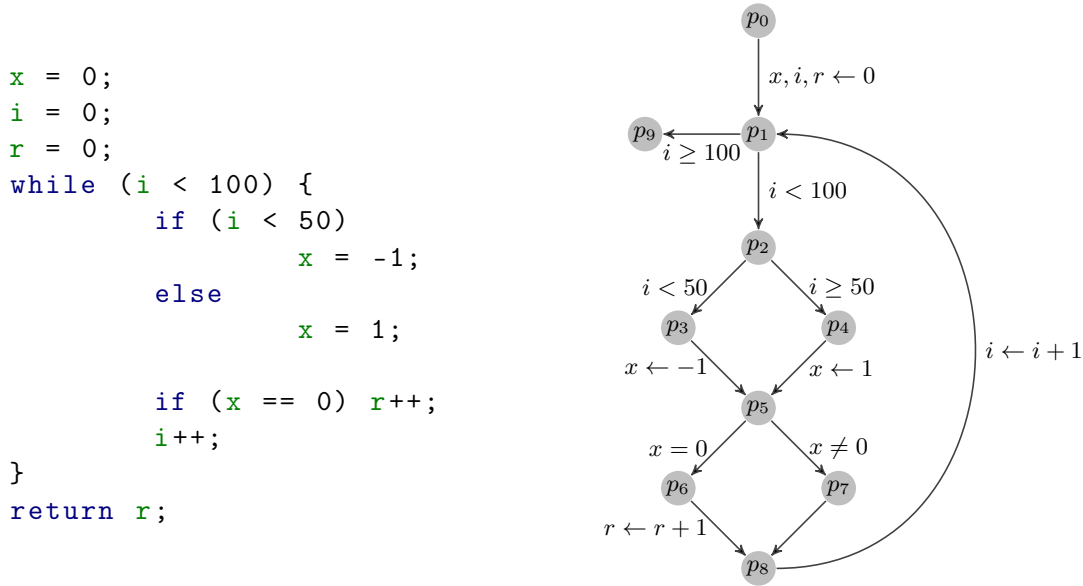


Figure 3.6: Example of program. p_5 is a control-flow merge point, which is in the middle of the containing loop.

inside our polyhedron. Finally, the analysis terminates after some steps and finds $0 \leq r \leq i$. *Lookahead Widening* is not able to prove that $r = 0$ at the end of the function.

Path Focusing The analysis of the loop starts with $p_1 : x = i = r = 0$. Then, the SMT-solver discovers the path $p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_5 \rightarrow p_7 \rightarrow p_1$. We do a widening/narrowing sequence and we find $0 \leq i \leq 50 \wedge r = 0 \wedge -1 \leq x \leq 0$ at p_1 . The SMT-solver gives another path: $p_1 \rightarrow p_2 \rightarrow p_4 \rightarrow p_5 \rightarrow p_7 \rightarrow p_1$. After analysing this path, we obtain $0 \leq i \leq 100 \wedge r = 0 \wedge x = 0$. Analysis terminates since the SMT-solver gives *unsat* answers. Here, We never focused on paths going through point p_6 , so our analysis gives us the invariant $x = 0$ at the end of the program. Indeed, there is no feasible paths going through p_6 , but *Lookahead Widening* do not know it because of the convex hull at point p_5 .

Chapter 4

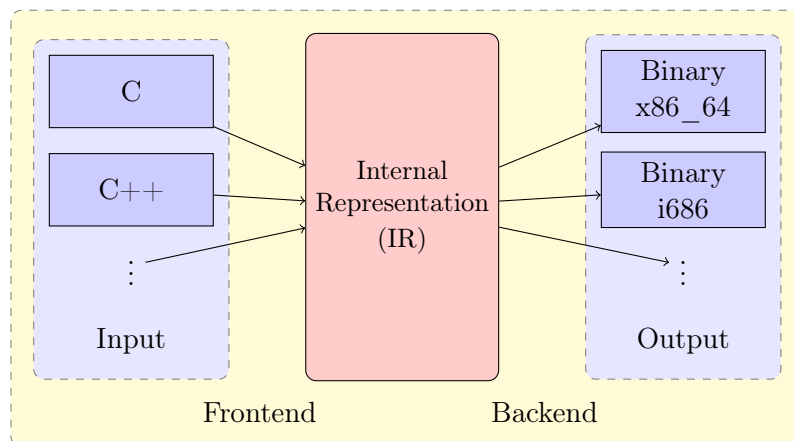
Implementation

Lookahead Widening and *Path Focusing* techniques have been implemented into a small analyser, so that we can compare the precision of their results. In this part, we explain some details about these implementations.

4.1 Infrastructure

The analyser is based on the *Low Level Virtual Machine* (LLVM) [LA04], which is a compilation infrastructure used for instance by the *clang* compiler.

4.1.1 LLVM internal representation



The advantage of using such an infrastructure is that the analyser can check programs written in various languages: C, C++, ..., without needing to write a specific frontend for each of them. Here, the analyser directly works on the internal representation (IR), which is a graph representation of the program. The control flow graph can easily be extracted from this representation.

Our analyser takes as parameter the LLVM internal representation of a program, and returns linear relations between the numerical variables at each control point of this program.

It first applies some optimisation passes over the IR, and then computes the linear relations by abstract interpretation.

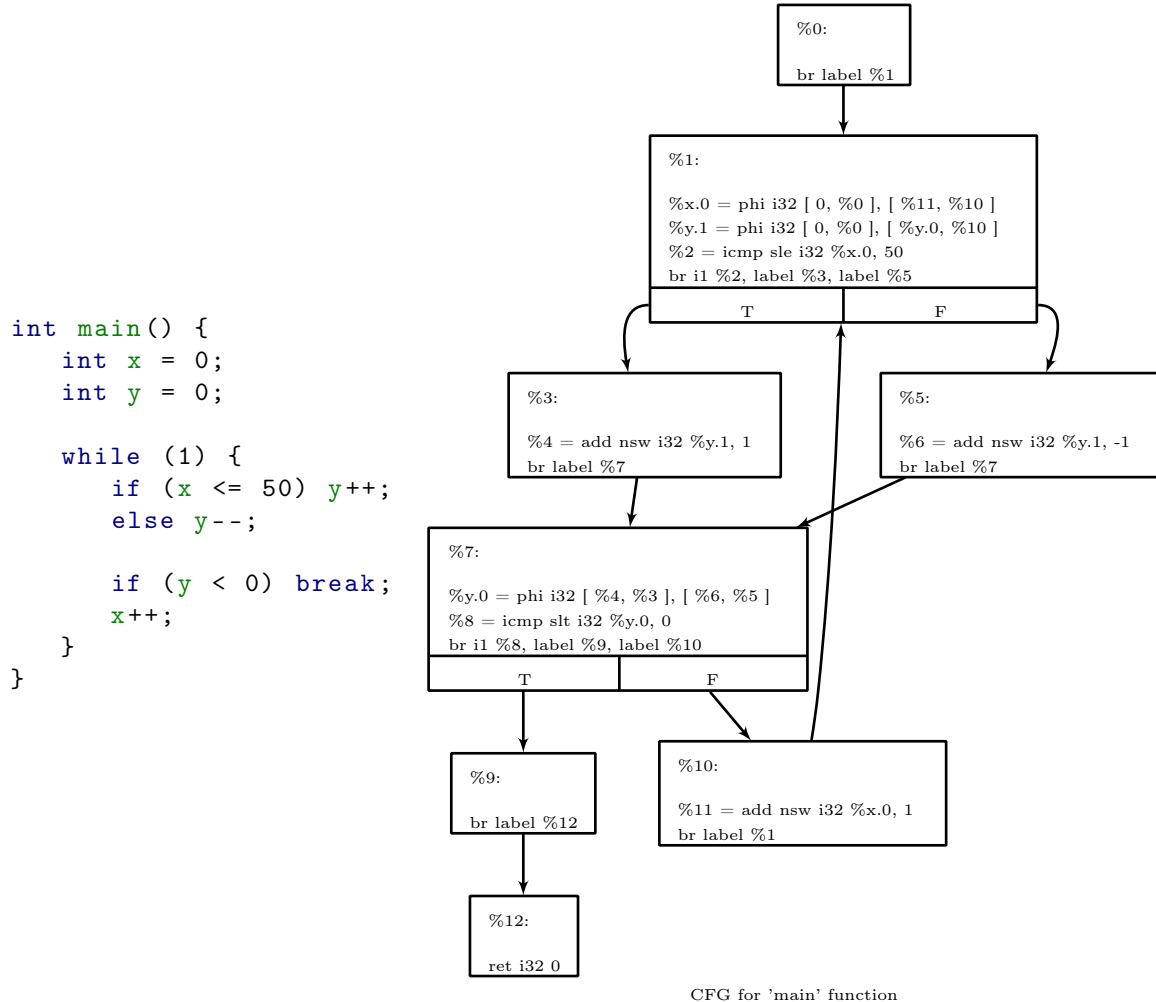


Figure 4.1: Our running example and its internal representation in SSA form

As we see in Figure 4.1, LLVM IR is slightly different from our definition of control flow graph: it uses the notion of *BasicBlock*. A *BasicBlock* is a sequence of instructions that are necessarily executed one after the others, meaning that if the first instruction of a BasicBlock is executed, then the rest of the instructions are necessarily executed exactly once, in order.

In our case, a control point will be a BasicBlock of the program.

4.1.2 Transformation passes

LLVM internal representation can be applied transformation passes that modify parts of it. These passes are mostly used for optimisation purposes. In our analyser, we apply some already existing passes before running our abstract interpretation pass. Among the passes we run, we could cite:

- the *PromoteMemoryToRegister* Pass (-mem2reg), that transforms the internal representation by promoting the memory variables to Static Single Assignment (SSA) registers.

This pass transform the control flow graph such that all the variables are in SSA form, and removes most of the *Load/Store* operations into memory. It is useful since our analyser does not have a memory model, and then loses lots of precision in the case of *Load/Store* instructions.

- the *LoopSimplify* (-loopsimplify) and *LoopInfo* (-loopinfo) Passes, that transform all loops into a canonical format, and compute some information related to loops (for instance, the set of BasicBlocks that are parts of the loop, etc.).
- the *LowerSwitch* Pass (-lowerswitch), that transform a switch instruction into its corresponding *if-then-else* instructions. After running this pass, there is no more *Switch* instruction in the control flow graph, so the implementation of the static analysis is easier.

4.1.3 Drawbacks

During the implementation, we encountered problems due to the use of the LLVM internal representation. Indeed, this format is not provided for static analysis, but for code generation. Then, some informations we need for a precise static analysis are lacking.

This is the case for the distinction between *signed* and *unsigned* integers, which is non-existent in the representation, since the compiler does not need this information to generate code. Yet, for static analysis, this distinction is required to improve precision: for instance, if n is unsigned, the constraint $n \geq 0$ could be added in our abstract value at the beginning of the analysis. This lack of precision may be limited by using disjunctive invariants: one could attach two polyhedra to the control point: the first one with the constraint $n \geq 0$, and the second one with $n < 0$.

Another example of information that was missing is the set of live variables. Indeed, in LLVM, live variables computation is processed during the register allocation. Then, the data structure we work on does not have live variables informations yet. To solve this problem, we coded our own pass, that is aimed at computing these live variables after the transformation in SSA form.

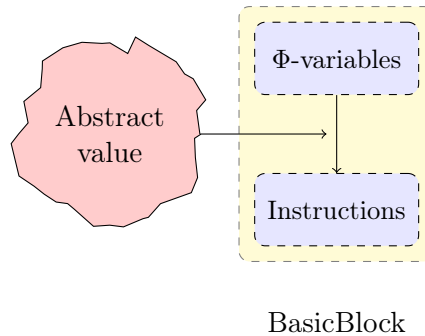
4.2 Abstract domain representation

The analyser is implemented using the APRON library [JM09], which is a common interface to libraries implementing all the features for several abstract domains, such as convex polyhedra (NewPolka), octagons (OCT), intervals (BOX). In our case, we used the convex polyhedra library, but we could switch to octagons or intervals easily.

4.2.1 Attachment to LLVM Internal Representation

We attach only one abstract value to each BasicBlock of the IR. Indeed, there would be a huge memory consumption if we would attach a polyhedron after each instruction. A BasicBlock begins with all its Φ -variables, and ends with the other instructions. We choose to attach our abstract value just after the Φ -variables assignments. In this way, the Φ -variables of the associated BasicBlock will be part of the dimensions of the polyhedron.

Morally, this means the Φ -variables are on incoming transitions, whereas the other instructions are on outgoing transitions.



4.2.2 Dimensions of the abstract values

In our implementation, we tried to reduce as much as possible the dimensions of the abstract values at each program point. Indeed, there is no need to consider the whole set of the program numerical variables at each point p , since lots of these variables are not *live* in p . The definition of a *live* variable is the following:

Definition 4.2.1 (Liveness) *A variable is live at the control point p iff:*

- *its value is available at p , meaning that there is a path from the point defining the variable to p .*
- *its value might be used in the future, meaning that there exist a path starting in p , that arrives in a point using the variable.*

LLVM internal representation allows to find the control point where a variable is defined and used, thanks to def-use chains: we can easily get the set of points where a variable is used, and we directly have a pointer to its definition. With all these informations, we implemented an LLVM pass that computes the set of live variables in SSA, using the liveness check algorithm described in [Boi10, section 3.3].

At a program point p , the dimensions of the associated abstract value could only be the numerical variables that are live at p . This is not exactly the case in our implementation, because of another optimisation: we only consider variables that are not a linear combination of other variables.

For instance, assume that x, y, z are numerical variables of a program, x is defined as $x = y + z$, and x, y, z are live at point p . Instead of having x as a dimension for X_p , we only have y and z . All the properties for x can be directly extracted from X_p and the information $x = y + z$. This is an optimisation in the sense that there is redundant information in the abstract value if both x, y and z are dimensions of X_p .

Then, liveness definition can be adapted in our case:

Definition 4.2.2 (Liveness by linearity) *A variable v is live by linearity at the control point p iff:*

- *its value is available at p , meaning that there is a path from the point defining the variable to p ,*
- *one of these conditions holds:*

- v is live in p .
- There is a variable v' , defined as a linear combination of other variables v_1, v_2, \dots, v_n , so that $\exists i \in \{1, \dots, n\}, v = v_i$, and v' is live by linearity in p .

Finally, a variable is a dimension of X_p iff it is live by linearity and it is not defined as a linear combination of program variables.

There is a special case for Φ -variables: in some cases, a Φ -variable can be considered as a linear combination of program variables, when only a single incoming value is possible. This is the case when all but one polyhedra associated to a predecessor are at bottom. For instance, let's say point p has p_1, p_2, \dots, p_n as ordered predecessors. At point p , we have $x = \Phi(x_1, x_2, \dots, x_n)$. If $\forall i \in \{2, \dots, n\}, X_i = \perp$, and $X_1 \neq \perp$, then we can use the linear relation $x = x_1$ instead of the Φ function. In practical cases, this is a way to reduce the size of the abstract values.

4.2.3 Diseq comparisons

The set of elements x satisfying $x \neq N$ is not convex: it is the union of two intervals: $] -\infty, N[\cup]N, +\infty[$. Since we use convex polyhedra to represent the set of possible states of our program, we lose information when the analysis comes across a *diseq* operation. One classical approach would be to compute a disjunctive invariant, by attaching two different polyhedron to the BasicBlock. However, we preferred the following technique: we transform all the operations of the form $x \neq y$ into $x < y \vee x > y$, meaning that we separate the cases *less than* and *greater than* into two distinct paths.

```

if (x != y) {
    ...
}
                                     ⇒
if (x > y) {
} else if (x < y) {
} else {
    ...
}

```

To do so, we could implement an LLVM pass that transform the internal representation according to this principle. However, this pass has not been implemented in our analyser, and this transformation has to be done manually.

4.3 Unrolling loops

In some cases, the number of iterations inside a loop during execution may be equal to zero. This happens if the condition of the *while* statement is false at the first time the head of loop is reached.

In our analyser, this kind of loops gives imprecise results. In this section, we propose to unroll loops once in order to avoid it. This behaviour can happen in very simple programs, such as the one depicted in Figure 4.2. Indeed, in the case of $n \leq 0$, the condition $i < n$ is always false.

In this example, our program behaves as follows:

- At point p_1 , we first have the polyhedron $i = 0$, where n is unconstrained.
- The image of this polyhedron after the loop transition is $i \leq n \wedge i = 1$. After a convex hull, we obtain for p_1 $0 \leq i \leq 1$.

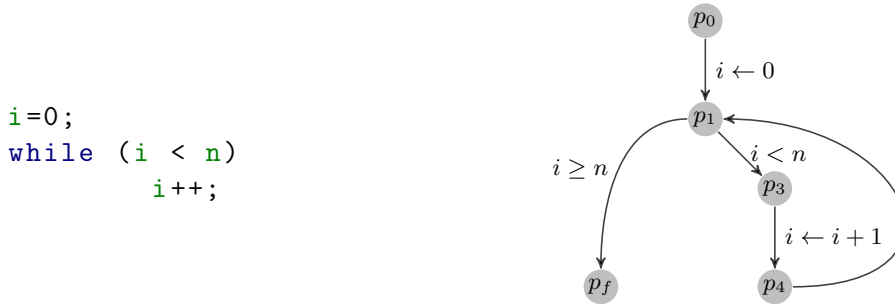


Figure 4.2: If $n \leq 0$, there is no loop iterations.

- Again, we compute the image of this polyhedron by the transition, $1 \leq i \leq 2 \wedge i \leq n$, and after widening we obtain at p_1 the polyhedron $0 \leq i$.
- Finally, at point p_f , we have the polyhedron $0 \leq i \wedge i \geq n$.

Now, after unrolling the loop once, the program is the one in Figure 4.3. The analysis gives:

- At point p_1 , we have $i = 0$, then we directly have at point p_f the polyhedron $i = 0 \wedge n \leq 0$.
- At point p_4 , we first have $i = 1 \wedge n \geq 1$.
- the image of the polyhedron after the loop transition is $2 \leq n \wedge i = 2$, which gives after a convex hull $1 \leq i \leq 2 \wedge i \leq n$.
- After one iteration and an application of widening, we find at p_4 the polyhedron $1 \leq i \leq n$.
- Then, at the exit point p'_f of the loop, we have the polyhedron $i = n \wedge i \geq 1$.

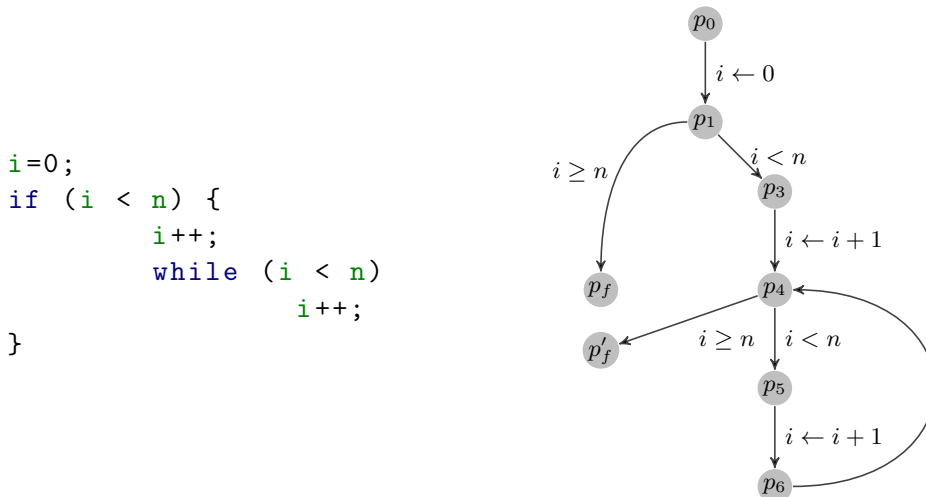


Figure 4.3: Program of Figure 4.2 after unrolling the loop once.

The convex hull of the polyhedron from p_f with the one from p'_f also gives $0 \leq i \wedge i \geq n$. Then, unrolling the loop does not give a better solution, since p_f and p'_f may be the same control point.

However, our technique does a convex hull only if this exit point is in P_R . If not, we consider independently the paths starting from p_4 and the ones starting from p_1 , hence the better precision. Indeed, this comes to computing a disjunctive invariant for the loop, by distinguishing the case where the number of iterations is zero.

In our analyser, we unroll every loops once, by using LLVM optimisation passes such as *loop-unroll*, *loop-rotate* or *loop-simplify*.

4.4 SMT-solving

Path focusing technique requires to decide the satisfiability of some SMT formula, and to extract a model when the formula is satisfiable, i.e an assignment of the variables so that the formula is *true*. For doing this, our implementation has an interface with Microsoft Z3 [dMB08], and Yices [DdM06]. One can easily switch from one to another with an argument we give as parameter to the analyser. In this way, we can compare the efficiency of both SMT-solvers.

4.5 Limitations

Our analyser could be extended in many ways. Indeed, it is intra-procedural, meaning that it does not consider that a variable may change after a function call. In addition, the return value of a function call is considered unconstrained. This limitation can be addressed by simply inlining the function call, provided the function is not recursive. There have been research to find efficient interprocedural analysis techniques [Bou90, Bou92, Jea09].

Additionally, our analyser has no memory model:

- *Store* instructions, that store a value in the memory, has no effect in the analysis.
- *Load* instructions, that get a value v in the memory and assign it to a variable x , does not provide properties about the value v . Then, our analyser considers x can take all values of its type.

In the case of local variables, the transformation of the internal representation into SSA form removes the *Load/Store* instructions. The use of global variables instead of local ones will induce huge loss of precision, since there will still be *Load/Store* instructions in the IA.

Additionally, some optimisation in the code could be proposed in order to reduce time and/or memory complexity. For instance, paths that have already been computed are until now stored in a tree. An optimisation could be to store them in a Binary Decision Diagram, in order to reduce significantly the memory consumption when there is a huge number of paths.

4.6 Example

In our running example (Figures 2.1, 3.3 and 3.5), the analyser outputs the following result for the basic block 1:

```
RESULT FOR BASICBLOCK: -----
; <label>:1 ; preds = %10, %0
  %x.0 = phi i32 [ 0, %0 ], [ %11, %10 ]
```

```

%y.0 = phi i32 [ 0, %0 ], [ %y.1, %10 ]
%2 = icmp sle i32 %x.0, 50
br i1 %2, label %3, label %5
-----
environment: dim = (2,0), count = 77
0: x.0
1: y.0
polyhedron of dim (2,0)
array of constraints of size 3
0: -x.0 - y.0 + 102 >= 0
1: y.0 >= 0
2: x.0 - y.0 >= 0

```

First, it shows the BasicBlock we work on, and its associated abstract value:

- The environment of the abstract value is its dimensions. Here, $x.0$ and $y.0$ are the dimensions of the polyhedron.
- The array of constraints defining the polyhedron. The three constraints here define the polyhedra seen in Figure 2.2.b.

4.7 Experiments

In this section, we evaluate our *Lookahead Widening* and *Path Focusing* implementations. We compare their execution time in some practical cases, and the precision of their results. Table 4.2 gives the results of the analyser in various examples, some of them taken from the recent literature [CGG⁺05, GR06].

In all experiments, the set P_R is chosen equal to $P_W \cup P_{ret} \cup P_{init}$, where P_W is the set of widening points. P_{init} is the set of starting points of the program, and P_{ret} is the set of points that have no successors.

4.7.1 Benchmarks

Path Focusing appears to perform similarly or better than *Lookahead Widening*. Indeed, it is more precise in the case of programs containing control-flow merges inside loops, thanks to the distinction between all the paths of the loop. In simpler programs, it performs as well as *Lookahead Widening*.

ex6.c is a program containing non-linear arithmetic: there is an integer multiplication inside the loop. The Apron library handles multiplication using a linearization technique. However, multiplication is not allowed in LIA (Linear Integer Arithmetic) or LRA (Linear Real Arithmetic). Then, Yices SMT-solver errs with a message “not implemented” (which is the reason of the ∞ symbol in the table), when Z3 still works correctly.

We tried our technique on example with a huge number of paths: *ex8*, *ex8b* and *ex8c* (see Table 4.1). These benchmarks are constructed such that the loop contains only a few possible paths, but a lot of unfeasible paths. The idea is to show that our technique does not blow up, even with a huge expanded multigraph. Results show that in such benchmarks, our technique finds a much better invariant and is even faster than *Lookahead Widening*.

For small programs, our execution time may not be significant. Indeed, the cost for reading the program, initialising the SMT-solver, . . . , may have a non-negligible cost compared to the abstract interpretation analysis itself. The only interest of these times is to show our algorithm is pretty fast.

Benchmark	Gopan	Beyer	Boustr	ex9	test1	ex6	ex8	ex8c	ex8b
paths (total)	5	4	10	8	3	5	75626	$5.7 \cdot 10^9$	4.10^{14}
paths (computed)	5	4	4	4	3	4	4	4	4
iterations	6	5	5	4	3	4	4	4	4

Table 4.1: Number of path in the expanded multigraph, number of path the algorithm actually computes, and total number of iterations

The last remark is about the efficiency of Yices and Z3. Yices seems to always be slightly faster than Z3, but in some cases, Yices appeared to blow up, or to err with a message “not implemented”, for instance when adding non-linear relations into the formula. Then, it was a good choice to link our analyser with both SMT-solvers.

4.7.2 Analysis of real code

In this section, we show that *Path Focusing* technique performs better than *Lookahead widening* on some code taken from real open-source programs.

Analysis of cBench programs

Collective Benchmark (cBench) [cBe] is a collection of open-source sequential programs assembled by the community to enable realistic benchmarking and research on program.

We compare *Lookahead Widening* and *Path Focusing* techniques on these benchmarks, and we respectively count the number of BasicBlocks where the invariant is more precise with the first technique, or with the second one, or if it the invariant is the same (see Table 4.3).

These benchmarks often contain non-linear arithmetic, especially multiplications. This is not a problem to the Apron library since it uses linearization to handle them, but this sometimes implies an execution time blowup for the SMT-solver:

- *Yices* does not work at all when the formula contains non-linear arithmetic, so we could not use it. Indeed, Linear Real Arithmetic or Linear Integer Arithmetic theories do not treat multiplications.
- *Z3* implements some linearization techniques in order to deal with non-linear arithmetic, even if the used theory is LRA or LIA. In lots of cases, it is sufficient for deciding the satisfiability of our formulae, but it seems sometimes to have a huge execution time blowup. When running our benchmarks, we fixed a time limit (5 minutes) for the analysis, and ignore in our results the functions that make the SMT-solver blow.

This drawback could be solved in the future, by linearizing parts of the SMT-formula that make the computation too costly, or make it fail (in some cases, the SMT-solver is unable to decide the satisfiability of the formula, and answers “unknown”).

Path Focusing technique finds more precise invariants in some cases, whereas *Lookahead Widening* performs better in other cases. This is partly due to the widening operator: even

Benchmark	Path Focusing			cmp.	Lookahead Widening	
	time		result		time	result
	Z3	Yices				
Gopan.c	0.076s	0.076s	-x.0 - y.0 + 102 >= 0 y.0 >= 0 x.0 - y.0 >= 0	=	0.032	-x.0 - y.0 + 102 >= 0 y.0 >= 0 x.0 - y.0 >= 0
Beyer.c	0.084s	0.076s	-x.0 + 100 >= 0 y.0 - 50 >= 0 -x.0 + y.0 >= 0	=	0.032	-x.0 + 100 >= 0 y.0 - 50 >= 0 -x.0 + y.0 >= 0
Boustr.c	0.100s	0.088s	-2x.0 + d.0 + 1999 >= 0 -2x.0 + 3d.0 + 2001 >= 0 -d.0 + 1 >= 0	⊆	0.064	-1996x.0 + 4991d.0 + 1998995 >= 0 -2x.0 + d.0 + 1999 >= 0 -d.0 + 1 >= 0
ex9.cpp	0.084s	0.082s	r.0 = 0 -i.0 + 100 >= 0 i.0 >= 0	⊆	0.052	-i.0 + 100 >= 0 r.0 >= 0 i.0 - r.0 >= 0 24i.0 - 25r.0 + 75 >= 0
test1.c	0.064s	0.064s	2j.0 + i.0 - 21 = 0 -j.0 + 10 >= 0 j.0 - 6 >= 0	=	0.016	2j.0 + i.0 - 21 = 0 -j.0 + 10 >= 0 j.0 - 6 >= 0
ex6.c	0.104s	∞	-y.0 + 51 >= 0 y.0 >= 0 x.0 - 51y.0 + 2550 >= 0 x.0 >= 0	⊆	0.032	y.0 >= 0
ex8.c	0.516s	0.408s	-y.0 - x.0 + 1 >= 0 -y.0 + x.0 + 1 >= 0 y.0 - x.0 + 1 >= 0 y.0 + x.0 + 1 >= 0	⊆	0.208	-x.0 + 1 >= 0 x.0 + 1 >= 0
ex8c.c	0.656s	0.548s	c.0 + a.0 = 0 b.0 + d.0 = 0 -b.0 - c.0 + 1 >= 0 -b.0 + c.0 + 1 >= 0 b.0 - c.0 + 1 >= 0 b.0 + c.0 + 1 >= 0	⊆	0.916s	-b.0 + 1 >= 0 -b.0 + a.0 + 1 >= 0 -c.0 + 1 >= 0 -c.0 + d.0 + 1 >= 0 -a.0 + 1 >= 0 -d.0 + 1 >= 0 a.0 + 1 >= 0 c.0 - d.0 + 1 >= 0 c.0 + d.0 + 1 >= 0 b.0 - a.0 + 1 >= 0 b.0 + 1 >= 0
ex8b.c	1.016s	0.816s	c.0 + y.0 = 0 c.0 + a.0 = 0 b.0 + d.0 = 0 b.0 + x.0 = 0 -b.0 - c.0 + 1 >= 0 -b.0 + c.0 + 1 >= 0 b.0 - c.0 + 1 >= 0 b.0 + c.0 + 1 >= 0	⊆	36.30s	-b.0 + 1 >= 0 -b.0 + a.0 + 1 >= 0 -c.0 + 1 >= 0 -c.0 + d.0 + 1 >= 0 -a.0 + 1 >= 0 -y.0 - x.0 + 1 >= 0 -y.0 + x.0 + 1 >= 0 -d.0 + 1 >= 0 x.0 + 1 >= 0 y.0 - x.0 + 1 >= 0 y.0 + 1 >= 0 a.0 + 1 >= 0 c.0 - d.0 + 1 >= 0 c.0 + d.0 + 1 >= 0 b.0 - a.0 + 1 >= 0 b.0 + 1 >= 0

Table 4.2: Results of *Path Focusing* and *Lookahead Widening* techniques applied to various benchmarks. The invariant we give is the one at the head of the loop.

if the analysis is more subtle, the result at the end may be worse. This is a well-known result in abstract interpretation: when using widening operations, the final result is not monotonic with the quality of the abstraction.

Benchmark	Path Focusing	Lookahead Widening	Same invariant	Total
automotive_bitcount	1	0	39	40
automotive_qsort1	9	0	13	22
automotive_susan_c	0	2	59	61
automotive_susan_e	0	2	59	61
automotive_susan_s	0	2	59	61
bzip2d	70	3	81	154
bzip2e	70	3	81	154
consumer_jpeg_c	22	8	711	741
consumer_jpeg_d	13	8	667	688
consumer_lame	13	15	366	394
consumer_mad	12	11	609	632
consumer_tiff2bw	19	3	593	615
consumer_tiff2rgba	21	6	661	688
consumer_tiffdither	21	3	662	686
consumer_tiffmedian	25	5	707	737
network_dijkstra	0	0	19	19
network_patricia	2	0	21	23
office_rsynth	6	0	158	164
office_stringsearch1	5	1	44	50
security_blowfish_d	4	0	23	27
security_blowfish_e	4	0	23	27
security_rijndael_d	1	0	31	32
security_rijndael_e	1	0	31	32
security_sha	1	0	26	27
telecom_CRC32	0	0	13	13
telecom_adpcm_c	0	0	10	10
telecom_adpcm_d	0	0	10	10
telecom_gsm	5	0	153	158

Table 4.3: Number of basicblocks where the computed invariant is more precise by *Path Focusing*, *Lookahead Widening*, or the same with both techniques.

Analysis of Open Source projects

We compare *Lookahead Widening* and *Path Focusing* on various GNU projects (see Table 4.4), so that we can see their precision on really used code.

The results we obtain are very promising. Most of the cases, our technique performs as well as *Lookahead Widening*, and discovers regularly more precise invariants.

However, *Lookahead Widening* seems to have better results in some cases. We already have some ideas for explaining this lack of precision in our technique:

- We use the domain of convex polyhedra as the abstract domain. Since this domain requires a widening operator, a more subtle analysis technique may sometimes discover invariants that are less precise than a more simplified technique.
- Our technique performs well in the case of self loops in the multigraph, thanks to the

Benchmark	Path Focusing	Lookahead Widening	Same invariant	Total
Bc	0	0	177	177
Gawk	4	3	284	291
Gnuchess	22	33	1506	1561
Gnugo	105	35	1303	1443
Grep	4	3	323	330
Gzip	9	3	189	201
Make	7	11	457	475
Tar	16	5	555	576
Wget	8	12	715	735

Table 4.4: Result of the analysis of various open-source projects.

widening/narrowing phases (or acceleration). Yet, in the case of paths that do not constitute a self loop in the multigraph, one performs a widening operation, but no narrowing iteration. Apparently, lots of the cases where *Lookahead Widening* performs better than *Path Focusing* are due to this drawback. An example of such loss of precision is depicted in appendix B. An idea that could solve this problem is to try to combine the two methods, and get the best of both worlds (see 5.3).

Although these project are not safety-critical and do not necessarily need static analysis, they give a good idea of the quality of our technique and the scalability of our implementation.

Chapter 5

Future Work

Path Focusing technique allows to work on control flow graphs with a huge number of paths, since they are computed only when needed. With this in mind, one could think of several improvements using path focusing, by encoding into the control flow graph new informations as paths, such as arithmetic overflows, alias analysis, ... In this part, we explain how to encode such kind of informations into implicit paths of the CFG.

5.1 Arithmetic Overflows

Some operations such as additions, subtractions... may overflow if their result is not in the type's definition interval. For instance, an *int* variable can only take values in $[-2^{31}, 2^{31} - 1]$. Then, there are two kind of overflows. Suppose $N = 2^{31}$:

- The result r of the operation may be greater than $N - 1$. In this case, the result we obtain is actually $r - 2N$.
- r may be lesser than $-N$. Then, the result is $r + 2N$.

The idea would be to distinguish the correct behaviour and the two kind of overflow in the control flow graph, each behaviour corresponding to a specific path (see Figure 5.1).

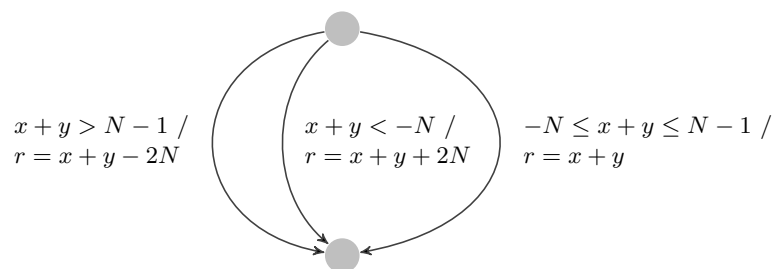


Figure 5.1: How to treat overflows of the addition $x + y$

This results in a huge blowup in the number of paths, but hopefully a lot of them will never be possible, so they will never be computed by our algorithm.

5.2 Alias analysis

One could use the same idea as for overflows, in order to treat aliases. Assume a pointer p that possibly points to variables x, y, z or t . For each operation involving $*p$, one distinguishes these different cases by creating a specific path for each of them. If the pointers are manipulated in a tricky way, one considers they may point to each variable of the program. Again, there is an exponential blowup in the number of paths, but these are treated implicitly.

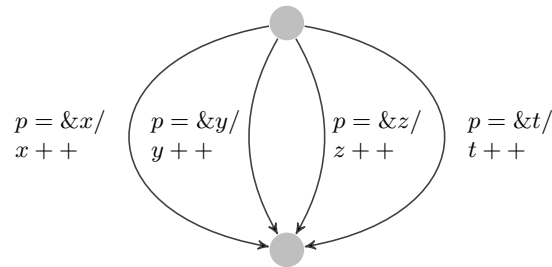


Figure 5.2: How to treat pointers: $(*p) ++$

5.3 Combining Lookahead Widening and Path Focusing

Path Focusing technique works on the expanded multigraph. Its limitation is that it does narrowing sequences during the iterations only for self-loops of this multigraph.

Lookahead Widening works on the classical control flow graph, and stabilises temporarily the fixpoint analysis on a part of this graph, forgetting paths that are not feasible at the first iteration.

One could try to combine these two techniques, and apply Lookahead Widening on the expanded multigraph, where the different paths are maintained implicit thanks to Path Focusing technique. The advantage of this approach compared to the classical Path Focusing is that narrowing iterations could be computed on loops that are not self-loops.

Chapter 6

Conclusion

Validation of software has become an essential domain in industry, especially those dealing with embedded systems, where a bug in the program can endanger the safety of people, or induce huge costs to fix. These domains constitute a privileged application field for formal verification techniques. Abstract Interpretation is a commonly used technique to do static analysis of programs. It is aimed at computing iteratively an over-approximation of the set of possible states of the program, and allows to prove properties such as “In this program, x is always between 1 and 10”, “The program never divides by zero”, etc. The objective is to obtain a result as precise as possible, by limiting the loss of precision during the analysis, induced for instance by the widening operator, or control-flow merges.

In this report, we proposed a new approach to guide the iterations of the analyser, in order to distinguish the different paths of the program and to treat them independently. The loss of precision due to widening and control-flow merges is then reduced. This technique allows us, for instance, to do narrowing steps right after widening, or to use acceleration techniques when a path actually is a simple loop.

Since distinguishing all the paths leads to an exponential blowup in time and memory, we do compute a path only when needed, otherwise the path is kept implicit. In this way, memory consumption is preserved, and the computation time is reduced, but still potentially exponential. In order to find which path we should compute, we use SMT-solving to find paths that makes the abstract interpretation analysis progress.

We implemented this technique into a small analyser dealing with C and C++ programs, and run it on several benchmarks. It appears that in lots of cases, the precision of the result is improved compared to the classical abstract interpretation technique. We compared it with *Lookahead Widening*, and our technique seems to perform better in the case of loops containing control-flow merges.

Our analyser is able to deal with small- and middle-sized test-cases, as well as real-life programs. It loses precision in the case of programs with pointers or several functions. It has been tested with program having loops containing several hundreds lines of code, and gave results after a time in the order of the second. It could be improved in many ways, in order to support for instance interprocedural analysis. A memory model could be added to handle programs with pointers and *load/store* instructions. One could also try to compute disjunctive invariants using the technique described in this report. Since our technique combines well with acceleration techniques, another direction is to implement this last technique to improve our results obtained with classical widening/narrowing.

Path Focusing technique performs better thanks to the distinction between paths. Then, encoding some properties in the control flow graph, such as arithmetic overflows or aliases, is one direction to explore. One could also try to combine *Path Focusing* with other techniques, such as *Lookahead Widening*.

Bibliography

- [BCC⁺03] 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 *PLDI*, pages 196–207. ACM, 2003. [2.3.1](#)
- [BHZ05] Roberto Bagnara, Patricia M. Hill, Elisa Ricci 0002, and Enea Zaffanella. Precise widening operators for convex polyhedra. *Sci. Comput. Program.*, 58(1-2):28–56, 2005. [2.3.1](#)
- [BJ06] Thomas Ball and Robert B. Jones, editors. *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*. Springer, 2006. [6](#)
- [Boi10] B. Boissinot. *Towards an SSA based compiler back-end: some interesting properties of SSA and its extensions*. PhD thesis, École Normale Supérieure de Lyon, 2010. [4.2.2](#)
- [Bou90] François Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In Pierre Deransart and Jan Maluszynski, editors, *PLILP*, volume 456 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 1990. [4.5](#)
- [Bou92] François Bourdoncle. *Sémantiques des Langages Impératifs d’Ordre Supérieur et Interprétation Abstraite*. PhD thesis, Ecole Polytechnique, 1992. Ph.D. dissertation. [2.3.3](#), [3.2.1](#), [4.5](#)
- [BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2010. [6](#)
- [cBe] cBench. Collective benchmarks. <http://cTuning.org/cbench>. [4.7.2](#)
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977. [2.2](#), [2.2.2](#)
- [CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see <http://www.di.ens.fr/~cousot>.) [2.2](#)

- [CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992. [2.2.2](#)
- [CGG⁺05] Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 462–475. Springer, 2005. [4.7](#)
- [CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978. [2.3](#), [2.3.1](#)
- [DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for `dppl(t)`. In Ball and Jones [[BJ06](#)], pages 81–94. [3.2.2](#), [4.4](#)
- [dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. [3.2.2](#), [4.4](#)
- [GH06] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In *13th International Static Analysis Symposium, SAS'06*, Seoul, Korea, aug 2006. [2.3.3](#), [3.2.4](#)
- [Gon07] Laure Gonnord. *Acceleration abstraite pour l'amélioration de la précision en analyse des relations linéaires*. PhD thesis, Université Joseph Fourier, Grenoble, France, 2007. [2.3.3](#), [3.2.4](#)
- [GR06] Denis Gopan and Thomas W. Reps. Lookahead widening. In Ball and Jones [[BJ06](#)], pages 452–466. [1](#), [2.1](#), [3.1](#), [3.2.4](#), [4.7](#)
- [GZ10] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 292–304. ACM, 2010. [1](#), [3.2.6](#), [3.2.6](#)
- [Hal79] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Université Scientifique et Médicale de Grenoble, March 1979. These de 3e cycle. [2.3.1](#)
- [HPR97] Nicolas Halbwachs, Yann-Eric Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, aug 1997. [2.3.1](#)
- [Jea09] B. Jeannet. Relational interprocedural verification of concurrent programs. In *Software Engineering and Formal Methods, SEFM'09*. IEEE, November 2009. [4.5](#)
- [JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009. [4.2](#)

- [KS08] D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008. [3.2.2](#)
- [LA04] 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. [4.1](#)
- [MG11] David Monniaux and Laure Gonnord. Using bounded model checking to focus fixpoint iterations. In *SAS*, 2011. [3.2](#)
- [Mon09] David Monniaux. A minimalistic look at widening operators. *Higher order and symbolic computation*, 22(2):145–154, December 2009. [2.2.2](#)

Appendices

A Generated ρ formula

```

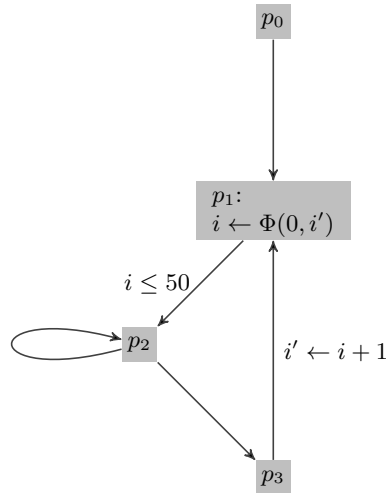
(and (= bd_0 false)
      (= t_0_1 bs_0)
      (or (not bd_0) true)
      (= b_5 t_1_5)
      (= x_%6_ (+ x_y.0_ -1))
      (= t_5_7 b_5)
      (or (not b_5) true)
      (= b_3 t_1_3)
      (= x_%4_ (+ x_y.0_ 1))
      (= t_3_7 b_3)
      (or (not b_3) true)
      (= b_7
         (or t_5_7 t_3_7))
      (= t_7_9 (and b_7 (< x_y.1_ 0)))
      (= t_7_10
         (and b_7 (not (< x_y.1_ 0))))
      (or (not b_7)
          (= x_y.1_ (ite t_3_7 x_%4_ x_%6_)))
      (= b_10 t_7_10)
      (= x_%11_ (+ x_x.0_ 1))
      (= t_10_1 b_10)
      (or (not b_10) true)
      (= bd_1
         (or t_10_1 t_0_1))
      (= t_1_3 (and bs_1 (<= x_x.0_ 50)))
      (= t_1_5
         (and bs_1 (not (<= x_x.0_ 50))))
      (or (not bd_1)
          (and (= x'_x.0_ (ite t_0_1 0 x_%11_))
                (= x'_y.0_ (ite t_0_1 0 x_y.1_))))
      (= b_9 t_7_9)
      (= t_9_12 b_9)
      (or (not b_9) true)
      (= bd_12 t_9_12)
      (or (not bd_12) true))

```

Figure 1: Expression of ρ associated to the program in Figures 2.1, 3.3 and 3.5, in the SMT-lib format.

B Example of loss of precision in Path Focusing

The following graph is a simplified example taken from one of our benchmarks, where *Path Focusing* technique obtains bad results.



p_1 and p_2 are both loop headers. We have $P_R = \{p_1, p_2\}$.

- We start with $p_1 : i = 0$. The SMT-solver finds the path $p_1 \rightarrow p_2$.
- At point $p_2 : i = 0$. Then, we focus on the path $p_2 \rightarrow p_3 \rightarrow p_1$. The obtained invariant in p_1 becomes $i \in [0, 1]$. Since this is a loop header, widening is applied: $i \geq 0$.
- The new focus path is $p_1 \rightarrow p_2$. The image of the polyhedron $i \geq 0$ by this transition is $0 \leq i \leq 50$. Again, p_2 is a loop header, and widening gives $i \geq 0$.

In this example, we do not have narrowing iterations since we never focus on self loops.