

Program verification

Data-flow analyses, Abstract Interpretation with Numerical
Domains

Laure Gonnord and David Monniaux

University of Lyon / LIP

October 2016



Context

Program Verification / Code generation:

- ▶ Variables : value range, scope, lifespan, constants, . . .
- ▶ Arrays : illicit accesses, alias discovery. . .
- ▶ Data Structures : memory leaks, null pointer dereferences. . .
- ▶ static analyses, of different kinds

Plan

Data Flow analysis

Available expressions - Recall from Compiler Course

Live Variable analysis

Toward a generalisation of these analyses

Abstract Interpretation

Transition systems and invariants

Computing Invariants (forward)

Non-relational vs relational analyses

Linear Relation Analysis

Classical Linear Relation Analysis

Some improvements

Diverse use of AI

Tools

Other abstract domains

Bonus: Additional Material



What for ?

Avoiding the computation of an (arithmetic) expression :

```
x:=a+b;  
y:=a*b;  
while(y>a+b) do  
    a:=a+a;  
    x:=a+b;  
done
```

Some defs

Definition

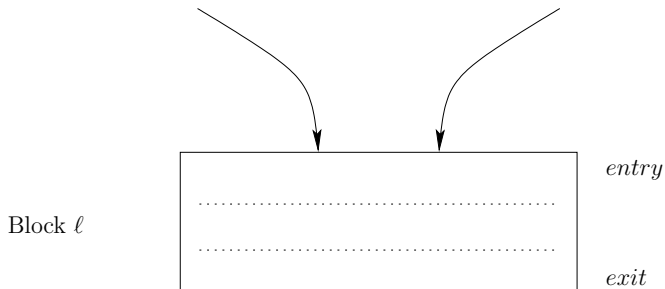
An expression is **killed** in a block if any of its variables is used in the block.

Definition

A **generated** expression is an expression evaluated in the block and none of its variables is killed in the block.

► Sets : $kill_{AE}(block)$ and $gen_{AE}(block)$

Data flow expressions



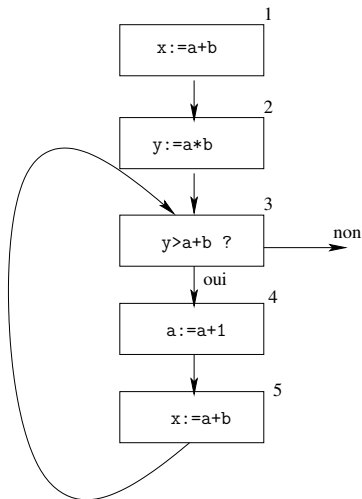
$$AE_{entry}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \textit{init} \\ \bigcap \{AE_{exit}(\ell') \mid (\ell', \ell) \in \textit{flow}(G)\} & \end{cases}$$

$$AE_{exit}(\ell) = (AE_{entry}(\ell) \setminus \textit{kill}_{AE}(\ell)) \cup \textit{gen}_{AE}(\ell)$$



On the example - equations

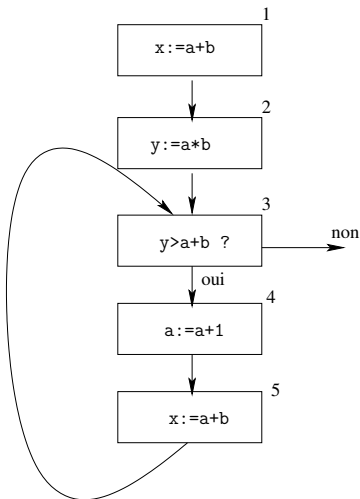
ℓ	$kill_{AE}(\ell)$	$gen_{AE}(\ell)$
1	\emptyset	$\{a+b\}$
2	\emptyset	$\{a*b\}$
3	\emptyset	$\{a+b\}$
4	$\{a+b, a*b, a+1\}$	\emptyset
5	\emptyset	$\{a+b\}$



On the example - final solution

l	$AE_{entry}(l)$	$AE_{exit}(l)$
1	\emptyset	$\{a+b\}$
2	$\{a+b\}$	$\{a*b, a*b\}$
3	$\{a+b\}$	$\{a+b\}$
4	$\{a+b\}$	\emptyset
5	\emptyset	$\{a+b\}$

- ▶ $a+b$ is available on entry to the loop, not $a*b$
- ▶ Improvement of code generation



Another example : live ranges

```
x:=2;  
y:=4;  
x:=1;  
if (y>x) then z:=y else z=y*y ;  
x:=z;
```

Definition

A variable is **live** at the exit of a block if there exists a path from the block to a use of the variable that does not redefine the variable.

Problem : determine the set of variables that *may be* live after each control point.



Data flow expressions

Definition

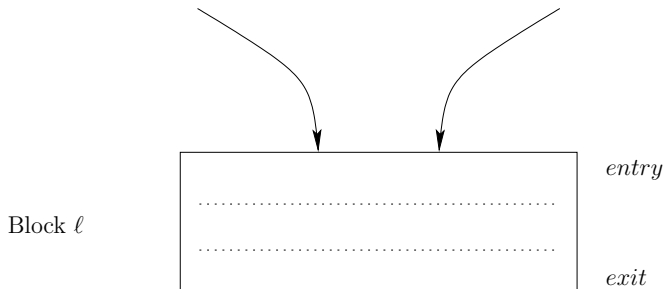
A variable that appears on the left hand side of an assignment is **killed** by the block. Tests do not kill variables.

Definition

A **generated** variable is a variable that appears in the block.

► Sets : $kill_{LV}(block)$ and $gen_{LV}(block)$

Data flow expressions



$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = final \\ \bigcup \{LV_{entry}(\ell') \mid (\ell', \ell) \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = (LV_{exit}(\ell) \setminus kill_{LV}(\ell)) \cup gen_{LV}(\ell)$$



Final result and use

Backward analysis and we want the smallest sets, here is the final result : (we assume all vars are dead at the end).

ℓ	$LV_{entry}(\ell)$	$LV_{exit}(\ell)$
1	\emptyset	\emptyset
2	\emptyset	$\{y\}$
3	$\{y\}$	$\{x, y\}$
4	$\{x, y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
5	$\{y\}$	$\{z\}$
5	$\{z\}$	\emptyset

► Use : Dead code elimination ! Note : can be improved by computing the use-defs paths. (see Nielson/Nielson/Hankin)

Common points

- ▶ Computing growing sets from \emptyset via *fixpoint iterations*. (or the dual)
- ▶ Sets of equations of the form (collecting semantics) :

$$SS(\ell) = \bigcup_{(\ell', \ell) \in E} f(SS(\ell'))$$

where f is computed w.r.t. the *program statements*

- ▶ SS is an **abstract interpretation** of the program.

Plan

Data Flow analysis

Available expressions - Recall from Compiler Course

Live Variable analysis

Toward a generalisation of these analyses

Abstract Interpretation

Transition systems and invariants

Computing Invariants (forward)

Non-relational vs relational analyses

Linear Relation Analysis

Classical Linear Relation Analysis

Some improvements

Diverse use of AI

Tools

Other abstract domains

Bonus: Additional Material



Goal

Propagating **information** about program variables (numerical, arrays, ...) in order to get **invariants**.

▶ We focus on **numerical variables** here.

Initial states + transitions

Program or machine state = values of variables, registers, memories. . . within state space Σ .

Examples:

- ▶ if system state = 17-bit value, then $\Sigma = \{0, 1\}^{17}$;
- ▶ = 3 unbounded integers, $\Sigma = \mathbb{Z}^3$;
- ▶ if finite automaton, Σ is the set of states ;
- ▶ if stack automaton, complete state = pair (finite state, stack contents), thus $\Sigma = \Sigma_S \times \Sigma_P^*$.

Transition relation $\rightarrow x \rightarrow y =$ "if at x then can go to y at next time".



Reachable states

Let $\Sigma_0 \subseteq \Sigma$ the set of initial states of the program.

The **reachable** states are obtained by successively applying the transition relation, hence σ is reachable iff :

$$\exists \sigma_0 \in \Sigma_0 \sigma_0 \rightarrow^* \sigma$$

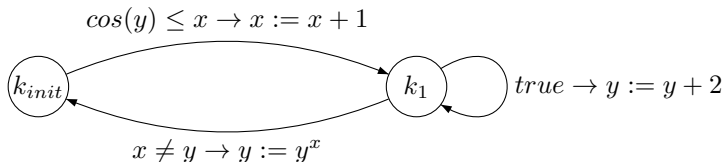
We also define X_n as the set of states reachable in at most n turns : $X_0 = \Sigma_0$, $X_1 = \Sigma_0 \cup R(\Sigma_0)$, $X_2 = \Sigma_0 \cup R(\Sigma_0) \cup R(R(\Sigma_0))$, etc.

with $R(X) = \{y \in \Sigma \mid \exists x \in X \ x \rightarrow y\}$.

The sequence X_k is ascending for \subseteq . Its limit (= the union of all iterates) is the **set of reachable states**.



Reachable states for programs



Semantics of the programs as **transition systems** :

- ▶ A **state** is a pair (pc, Val) :

$$\text{Val} : \text{Var} \rightarrow \mathcal{N}^d$$

- ▶ Var is $\llbracket 0, \dots, d - 1 \rrbracket$ (finite set, d vars)
- ▶ \mathcal{N} is $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$
- ▶ **Initial** states : $(pc_0, \text{all}v)$.



Iterative computation

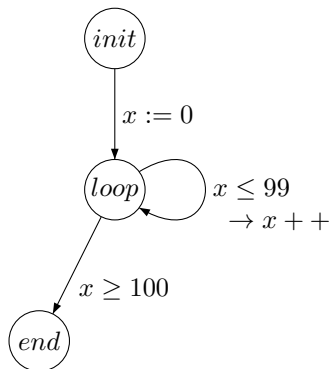
Remark $X_{n+1} = \phi(X_n)$ with $\phi(X) = \Sigma_0 \cup R(X)$.

How to **compute efficiently** the X_n ? And the limit?

- ▶ Explicit representations of X_n (list all states) : If Σ finite, X_n converges in at most $|\Sigma|$ iterations.
- ▶ else, we have to cope with two problems :
 - ▶ Representing the X_i s and computing $R(X_i)$.
 - ▶ Computing the limit ?
- ▶ $X_\infty = \bigcup \phi^n(X_0)$ is the strongest **invariant** of the program
- ▶ Looking for overapproximations : $X_\infty \subseteq X_{result}$ also called **invariant**.



Invariants for programs



- ▶ $\{x \in \mathbb{N}, 0 \leq x \leq 100\}$ is the most precise invariant in control point *loop*.

Back to our problem

Given a program (or an interpreted automaton), find inductive invariants for each control point : Recall : a **state** is a pair

(pc, Val) :

$$Val : Var \rightarrow \mathcal{N}^d$$

► We want to compute $lfp(\phi)$ with

$$\phi(X) = X_0 \cup \{y \in \Sigma \mid \exists x \in X \ x \rightarrow y\}$$

and \rightarrow entails the **actions** of the program.



Representing sets of valuations

First problem to cope with : **represent sets of** valuations

$$\text{Val} : \text{Var} \rightarrow \mathcal{N}^d$$

- ▶ Var is $\llbracket 0, \dots, d - 1 \rrbracket$ (finite set, d vars)
- ▶ \mathcal{N} is $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$
- ▶ Find a finite representation !

Computing R

Second problem to cope with : **computing** the transition relation

$$R(pc, X) = \{(pc', x') \mid \exists x \in X \text{ and } (pc, x) \rightarrow (pc', x')\}$$

- ▶ X is a (representation of a) set of valuations
- ▶ \rightarrow is the program transition function.
- ▶ Let's try **intervals** (easy storage, easy computation) !

A first example

Try to compute an **interval** for each variable at each program point using **interval arithmetic** :

```
assume(x >= 0 && x<= 1);
```

```
assume(y >= 2 && y= 3);
```

```
assume(z >= 3 && z= 4);
```

```
t = (x+y) * z;
```

Interval for z? [6, 16]

Loops?

Push intervals / polyhedra forward. . .

```
int x=0;
while (x<1000) {
    x=x+1;
}
```

Loop iterations $[0, 0]$, $[0, 1]$, $[0, 2]$, $[0, 3]$, . . .

How? $\phi(X) = \text{Initial state} \sqcup R(X)$, thus
 $\phi([a, b]) = \{0\} \sqcup [a + 1, \min(b, 999) + 1]$

► Stricly growing interval during 1000 iterations, then stabilizes : $[0, 1000]$ is an **invariant**.

Termination Problem

Third problem to cope with : **stopping the computation** :

- ▶ Too many computations
- ▶ unbounded loops

One solution...

Extrapolation!

$[0, 0], [0, 1], [0, 2], [0, 3] \rightarrow [0, +\infty)$

Push interval:

```
int x=0; /* [0, 0] */
while /* [0, +infty) */ (x<1000) {
    /* [0, 999] */
    x=x+1;
    /* [1, 1000] */
}
```

Yes! $[0, \infty[$ is stable!

Computing inductive invariants as intervals

- ▶ Representation : intervals. The union leads to an overapproximation.
- ▶ We don't know how to compute $R(P)$ with P interval (The statements may be too complex, ...)
 - ▶ Replace computation by simpler over-approximation $R(X) \subseteq R^\#(X)$.
 - ▶ The convergence is ensured by **extrapolation/widening**.
- ▶ We always compute $\phi^\#(X)$ with : $\phi(X) \subseteq \phi^\#(X)$
In the end, **over-approximation** of the least fixed point of ϕ .

Computing inductive invariants as intervals - 2

Interval operations :

- ▶ $+$, $-$, \times on intervals : interval arithmetic
- ▶ union : $[a, b] \cup [c, d]$: losing info !
- ▶ **widening** : $(l_1 \nabla l_2$ with $l_1 \subseteq l_2$)

$$\perp \nabla l = l$$

$$[a, b] \nabla [c, d] = [\text{if } c < a \text{ then } -\infty \text{ else } a, \\ \text{if } d > b \text{ then } +\infty \text{ else } b]$$

The idea is to infer the dynamic of the intervals thanks to the first terms.

Computing inductive invariants as intervals - 3

The widening operator being designed, we compute
 $(x \subseteq F(x))$

$$\Sigma_0, Y_1 = \Sigma_0 \nabla F(\Sigma_0), Y_2 = Y_1 \nabla F(Y_1) \dots$$

finite computation instead of :

$$\Sigma_0, F(\Sigma_0), F^2(\Sigma_0), \dots$$

which can be infinite.

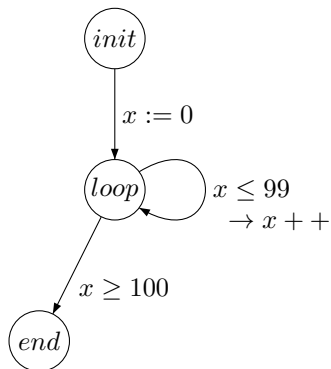
Theorem

*(Cousot/Cousot 77) Iteratively computing the reachable states from the entry point with the interval operators and applying widening at entry nodes of loops converges in a **finite** number of steps to a overapproximation of the least invariant (aka **postfixpoint**).*

► The widening operators must satisfy the non ascending chain condition (see Cousot/Cousot 1977).



Invariants for programs - ex 1



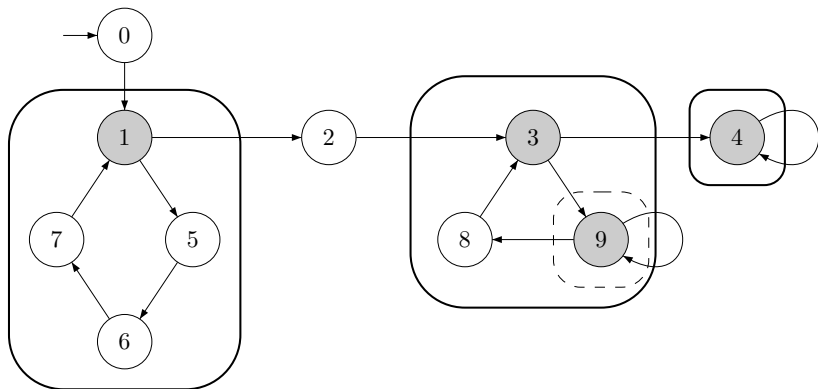
- ▶ $x \in [0, +\infty]$ in loop.

Computing inductive invariants as intervals - ex 2

```
x = random(0,7);  
y = cos(x)+x  
while (y<=100) {  
  if (x>2) x--;  
  else {  
    y = -4;  
    x--;  
  }  
}
```


Nested loops / Several loops

(Bourdoncle, 1992) Computing strongly connected subcomponents and iterate inside each :



Gray nodes are **widening nodes**

Improving precision after convergence - 1

```
int x=0; /* [0, 0] */
while /* [0, +infty) */ (x<1000) {
  /* [0, 999] */
  x=x+1;
  /* [1, 1000] */
}
```

we got $[0, +\infty)$ instead of $[0, 999]$. Run one more iteration of the loop: $\{0\} \sqcup [1, 1000] = [0, 1000]$. Check if $[0, 1000]$ is an inductive invariant ? **YES**

► This is called **narrowing** or descending sequence : ends when we have an inductive invariant or after k applications of the transition function.

Improving precision after convergence - 2

Let \hat{x} be the result of the computation

Result

The descending sequence always improves precision.

Proof : $lfp(F) \subseteq \hat{x}$, then $F(lfp(F)) = lfp(F) \subseteq F(\hat{x})$, and $F(\hat{x})$ is again a correct invariant. If \hat{x} is not a fixpoint, then $F(\hat{x}) \subset \hat{x}$, so is a strictly better invariant.

Best invariant in domain not computable

$P()$;

$x=0$;

Best invariant at end of program, as interval?

$[0, 0]$ iff $P()$ terminates

\emptyset iff $P()$ does not terminate

Entails solving the **halting problem**.

When intervals are not sufficient

```
assume(x >= 0 && x <= 1);
```

```
y = x;
```

```
z = x-y;
```

- ▶ The human (intelligent) sees $z = 0$ thus interval $[0, 0]$, taking into account $y = x$.
- ▶ Interval arithmetic does not see $z = 0$ because it does not take $y = x$ into account.

How to track relations

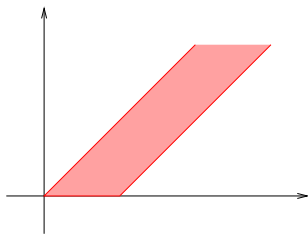
Using **relational domains**.

E.g.: keep

- ▶ for each variable an interval
- ▶ for each pair of variables (x, y) an information $x - y \leq C$.
- ▶ (One obtains $x = y$ by $x - y \leq 0$ and $y - x \leq 0$.)

How to **compute** on that?

Bounds on differences : practical example



Suppose $x - y \leq 4$, computation is $z = x + 3$, then we know $z - y \leq 7$.

Suppose $x - z \leq 20$, that $x - y \leq 4$ and that $y - z \leq 6$, then we know $x - z \leq 10$.

We know how to **compute** on these relations (transitive closure / shortest path).

On our example, obtain $z = 0$.



Why this is useful

Let $t(0..n)$ an array in the program.
The program writes $t(i)$.

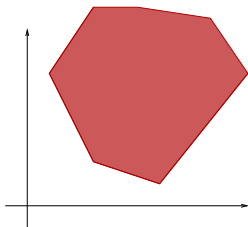
Need to know whether $0 \leq i \leq n$, otherwise said find bounds on i and on $n - i \dots$

Can we do better?

How about tracking relations such as $2x + 3y \leq 6$?

At a given program point, a set of **linear inequalities**.

In other words, a **convex polyhedron**.



Plan

Data Flow analysis

Available expressions - Recall from Compiler Course

Live Variable analysis

Toward a generalisation of these analyses

Abstract Interpretation

Transition systems and invariants

Computing Invariants (forward)

Non-relational vs relational analyses

Linear Relation Analysis

Classical Linear Relation Analysis

Some improvements

Diverse use of AI

Tools

Other abstract domains

Bonus: Additional Material



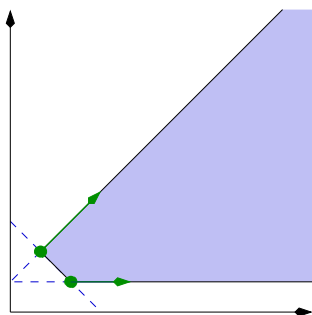
Intro

(Halbwachs/Cousot 1979)

- ▶ Abstract Interpretation in the Polyhedral domain
- ▶ Infinite Domain with many particularities
- ▶ Discover affine relations on variables
- ▶ Classically used in verification problems.

The polyhedral domain (1)

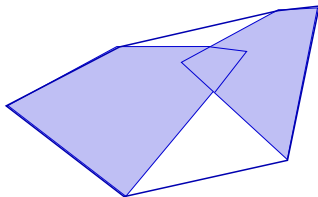
Convex polyhedra representation :



- Effective and efficient algorithmic (emptiness test, union, affine transformation ...)

The polyhedral domain(2)

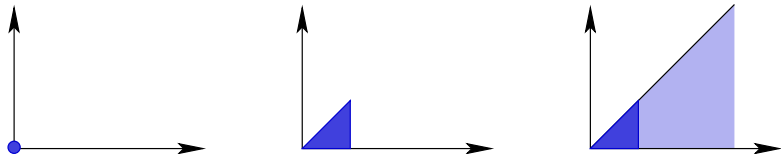
- ▶ Intersection, emptiness
- ▶ Affine Transformation : $a(P) = \{CX + D \mid X \in P\}$.
- ▶ Convex hull (loss of precision)



The Polyhedral domain (3)

Widening : $P \nabla Q$: limit extrapolation.

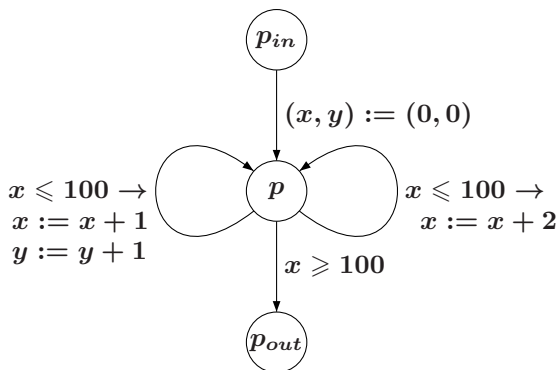
$P \nabla Q$ constraints : take Q constraints and remove those which are not saturated by P .



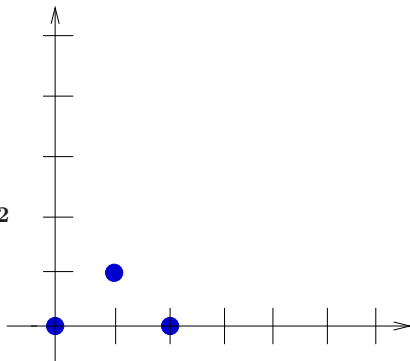
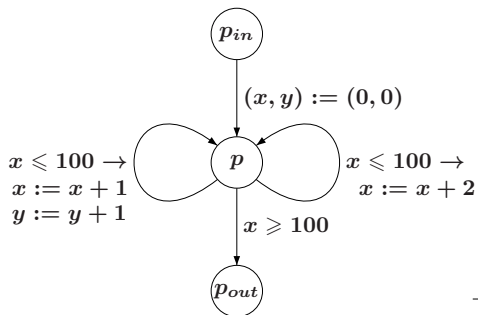
Trick (!) : $\{x = y = 0\} = \{0 \leq y \leq x \leq 0\}$

Analysis example - 1

```
x:=0;y:=0
while (x<=100) do
  read(b);
  if b then
    x:=x+2
  else begin
    x:=x+1;
    y:=y+1;
  end;
endif
endwhile
```



Example - 2



Linear Relation Analysis - Problems

Complexity increases with :

- ▶ number of control points
- ▶ number of numerical variables

Approximation is due to :

- ▶ Convex hulls
- ▶ Widening

(credits for these slides : Nicolas Halbwachs)

Complexity

(In general) The more precise we are, the higher the costs.

- ▶ Intervals: algorithms $O(n)$, n number of variables.
- ▶ Differences $x - y \leq C$: algorithms $O(n^3)$
- ▶ Octagons $\pm x \pm y \leq C$ (Miné) : algorithms $O(n^3)$
- ▶ Polyhedra (Cousot / Halbwachs): algorithms often $O(2^n)$.

Delaying widening - 1

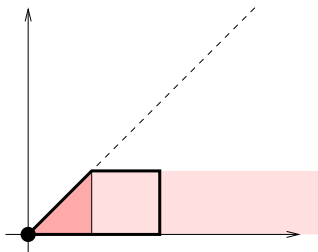
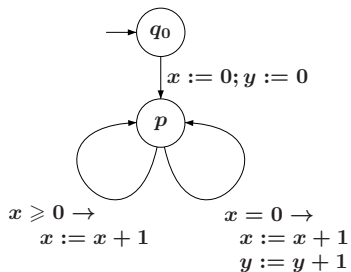
Halbwachs 1993 / Goubault 2001 / Blanchet et al. 2003

Fix k and compute :

$$X_n = \begin{cases} \perp & \text{if } n = 0 \\ F(X_{n-1}) & \text{if } n < k \\ X_{n-1} \nabla F(X_{n-1}) & \text{else.} \end{cases}$$

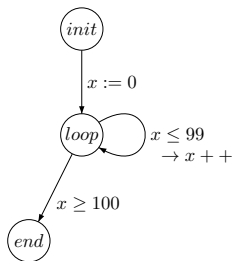
- ▶ Similar to unrolling loops, costly but useful (regular behaviour after a constant number of iterations).

Delaying widening - 2 - ex



Improving the widening operator

While applying $P \nabla Q$, intersect with constraints that are satisfied by both P and Q . The constraints must be precomputed.



Here, with “ $x \leq 100$ ” in the pool of constraints, it avoids narrowing.

► Warning **widening is not monotone**, so improving locally is not necessarily a good idea !



Local improvement with acceleration

(Gonnord/Halbwachs 2006, Schrammel 2012)

Idea : Sometimes, a fixpoint of a loop can be easily computed without any fixpoint iteration.

[More details here](#)

Good path heuristic

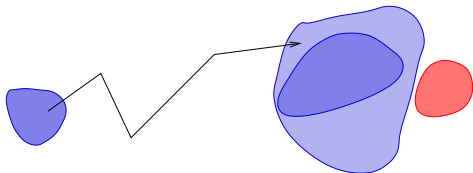
(Gonnord/Monniaux 2011)

Idea : find interesting paths by means of smt-queries

[More details here](#)

Applications

- ▶ Bounds on iterators of arrays (intervals, differences on bounds)
- ▶ Dead code elimination (all domains) - especially when the code has been automatically generated / asserts
- ▶ Vectorization : computations that can be permuted
- ▶ Memory optimisation : this `int` can be encoded in 16 bits ?
- ▶ Preconditions for code specialization (on going work with F. Rastello)
- ▶ Safety analysis



Plan

Data Flow analysis

Available expressions - Recall from Compiler Course

Live Variable analysis

Toward a generalisation of these analyses

Abstract Interpretation

Transition systems and invariants

Computing Invariants (forward)

Non-relational vs relational analyses

Linear Relation Analysis

Classical Linear Relation Analysis

Some improvements

Diverse use of AI

Tools

Other abstract domains

Bonus: Additional Material



Tools - Academic

ASPIC : Accelerated Symbolic Polyhedral Invariant
Computation

Aspic is **an invariant generator** :

- ▶ From counter automata with numerical variables.
- ▶ Invariants are **polyhedra**.

C2fsm is **a C parser** :

- ▶ From a source file in (a subset of) C into Aspic input language (fast).
 - ▶ **Safe** abstractions of non numerical variables, structures, behaviors.
- ▶ <http://laure.gonnord.org/pro/aspic/aspic.html>



Tools - More robust

- ▶ Frama-C : analysing/ proving correction of C programs (see <http://frama-c.com/>)
- ▶ Apron : numerical domain interface (<http://apron.cri.enscm.fr/library/>)
- ▶ Interproc : IA analyser connected to Apron (see <http://pop-art.inrialpes.fr/interproc/interprocweb.cgi>)
- ▶ Rose / LLVM : C (and more) parsers and API for manipulating C programs. Rose is more dedicated to program transformation, LLVM to compiler construction(<http://www.rosecompiler.org/> and <http://llvm.org/>).



Industrial succes stories

- ▶ Polyspace
- ▶ Astree
- ▶ See later for anecdotes.

Plan

Data Flow analysis

Available expressions - Recall from Compiler Course

Live Variable analysis

Toward a generalisation of these analyses

Abstract Interpretation

Transition systems and invariants

Computing Invariants (forward)

Non-relational vs relational analyses

Linear Relation Analysis

Classical Linear Relation Analysis

Some improvements

Diverse use of AI

Tools

Other abstract domains

Bonus: Additional Material



Octagons 1/2

```
while (1) {  
  R = A-Z;  
  L = A;  
  if (R>V)  
    { ★ L = Z+V; }  
  ★  
}
```

... Toward a cheaper abstract domain

- ▶ At ★, intervals give $L \leq \max(\max A, (\max Z) + (\max V))$.
- ▶ In fact, we have $L \leq A$
- ▶ To discover this, we must know at ★ that $R = A - Z$ and $R > V$.

Octagons 2/2

- ▶ **Octagons** a good cost / precision trade-off.
- ▶ Invariants of the form $\pm x \pm y \leq c$, with $\mathcal{O}(\mathbb{N}^2)$ memory and $\mathcal{O}(\mathbb{N}^3)$ time cost.
- ▶ Here, $R = A-Z$ cannot be discovered, but we get $L - Z \leq \max R$ which is sufficient.
- ▶ Use many octagons on small packs of variables instead of a large one using all variables to cut costs.

Other

filters, pointers, . . . later

Exercise : arrays

Construct two abstract domains:

- ▶ array smashing
- ▶ array with one FIXED distinguished cell

Give α, γ and abstract versions of the classical operations on arrays, and widening !

Plan

Data Flow analysis

Available expressions - Recall from Compiler Course

Live Variable analysis

Toward a generalisation of these analyses

Abstract Interpretation

Transition systems and invariants

Computing Invariants (forward)

Non-relational vs relational analyses

Linear Relation Analysis

Classical Linear Relation Analysis

Some improvements

Diverse use of AI

Tools

Other abstract domains

Bonus: Additional Material



LRA and acceleration

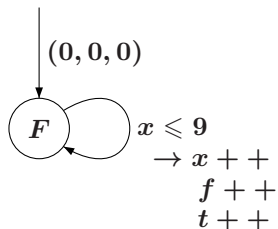
(Gonnord/Halbwachs 2006, Gonnord/Schrammel 2012, Schrammel/Jeannet 2014)

Combination LRA and acceleration techniques [Finkel/Sutre/Leroux/...]

- ▶ Abstract acceleration notion :
 - ▶ low-cost overapproximations;
 - ▶ inside LRA, combination with widening.
- ▶ Classification of accelerable loops.
- ▶ Prototype : ASPIC

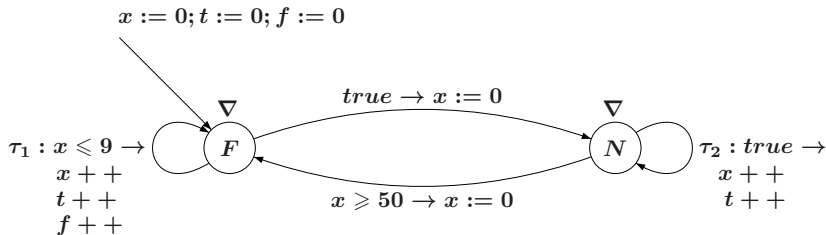
Acceleration - accelerable loops

An easy case



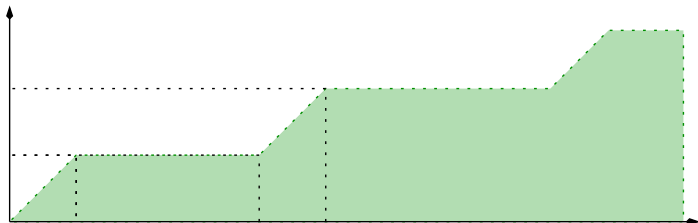
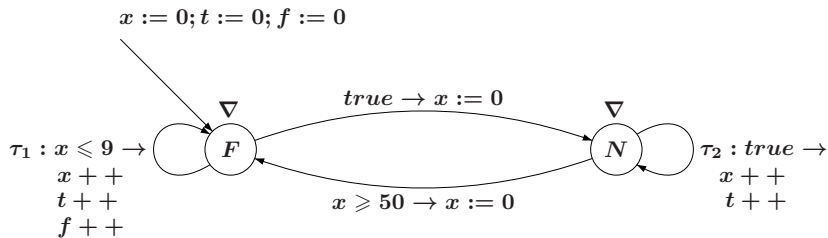
- ▶ **exact effect** : $\exists i \in \mathbb{N}, x = f = t = i, 0 \leq i \leq 10$
- ▶ **exact effect in the abstract domain** :
 $\{x = f = t, 0 \leq t \leq 10\}$

Gas Burner example - 1

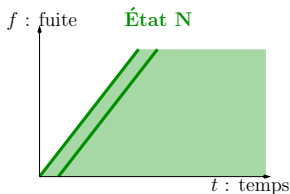
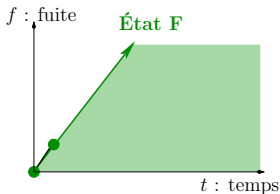
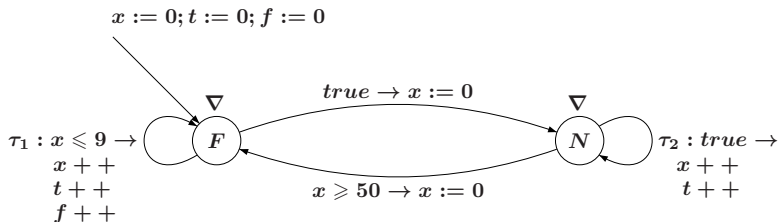


- ▶ f global leaking time
- ▶ t global time
- ▶ x local variable

Gas burner 2 - Real Behaviour

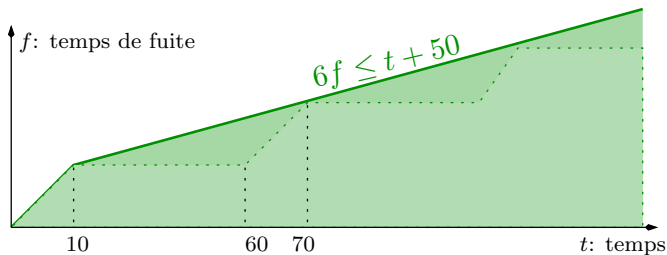
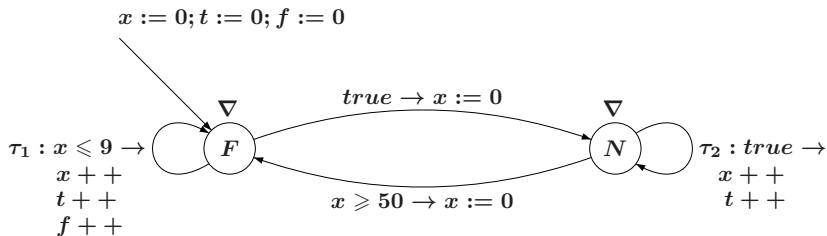


Gas burner 3 - with LRA



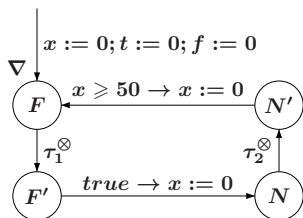
► Loss of precision

Gas burner - desired invariant



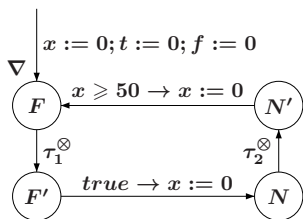
Accelerating the gas burner - 1

(mini-)loops are replaced ($\tau_i : g_i \rightarrow a_i$)



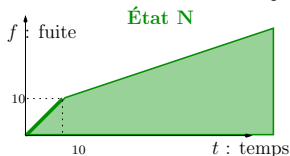
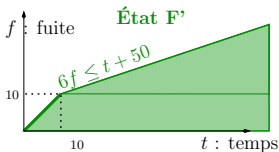
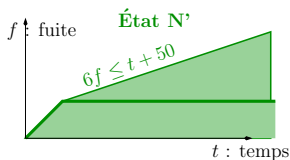
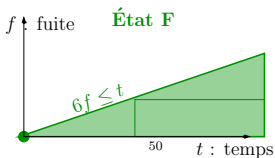
- ▶ τ_i^\otimes summarizes the effect of any application of τ_i (unfixed number of iterations).
- ▶ Outer loop is **widened**.

Accelerated Gas Burner - 2 back



► $\tau_1^\otimes =$ “add ray (1, 1, 1) while $x \leq 10$ ”

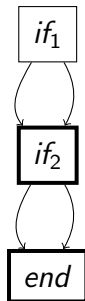
► $\tau_2^\otimes =$ “add ray (1, 0, 1)”



SMT+LRA, Motivation : example 1

Some properties cannot be expressed in convex abstract domains:

```
if (x >= 0) { xabs = x; }
  else { xabs = -x; }
if (xabs >= 0.01) {
  y = (sin(x) / x) - 1;
} else {
  xsq = x*x;
y = xsq*(-1/6. + xsq/120.);
}
```



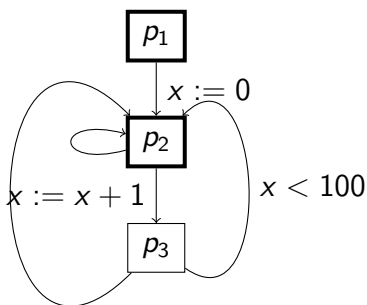
- Store the fact that $x = xabs \vee x = -xabs$ at node *if*₂

SMT + LRA, Motivation : example 2

The widening operator can be too coarse:

```
int x = 0;
while (true) {
  if (nondet()) {
    x = x+1;
    if (x >= 100) x = 0;
  }
}
```

$x \geq 100$
 $x := 0$



- The analysis (interval domain) gives $[0, +\infty)$, not improved by narrowing !

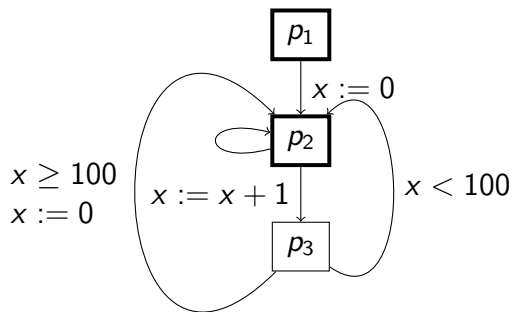
Two ideas

- ▶ First idea : do not compute the convex hull on “diamonds” .
- ▶ Second idea : consider all paths and analyse them **separately**.

- ▶ Advantage : precision
- ▶ Drawback : combinatorial explosion

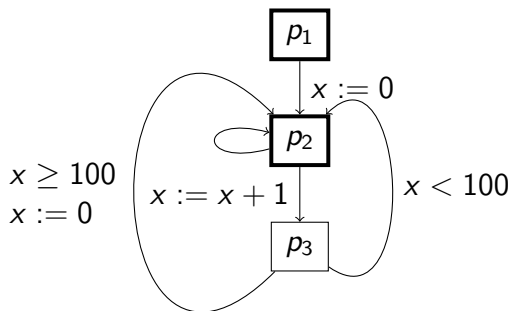
- ▶ Our method will implement these two ideas without computing all the program paths explicitly.

Invariants



Is $x = 0$ an invariant in p_2 ?

Invariants

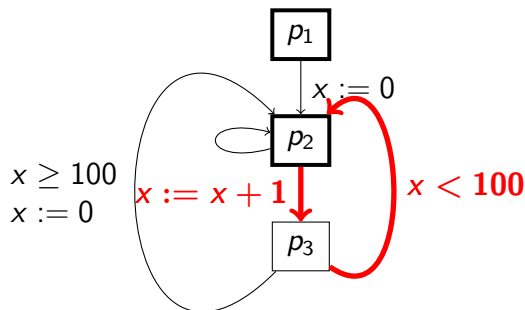


Is $x = 0$ an invariant in p_2 ?

► No ! Because it's not stable.

Our method on example 2

Focus on the red path !



- Its least inductive invariant (for p_2) is $x \in [0, 99]$, which is also an invariant while considering the whole graph.

How to detect paths ? [back](#)

We delegate the search for new paths to an SMT solver.
The problem is encoded into an SMT-problem thanks to the use of an internal **structure**:

- ▶ compact; (complexity reasons)
 - ▶ acyclic; (to reason about loops as for paths)
 - ▶ all variables are assigned once (to reason about unique variable values).
- ▶ Preprocessing : computing this structure.
(SAS 2011 for technical details)