Local Bit-Precise Reasoning in Program Analysis

with applications to verification

Harald Søndergaard

Melbourne

ITAP, 28 June 2012

The context

Building tools for the analysis and verification of LLVM.

Planning to use whatever tools and tricks available, but starting from abstract interpretation.

Graeme Gange, Andy King (UK), Jorge Navas, Peter Schachte, Harald Søndergaard, Peter Stuckey.

The setting

Analysing control flow graphs.

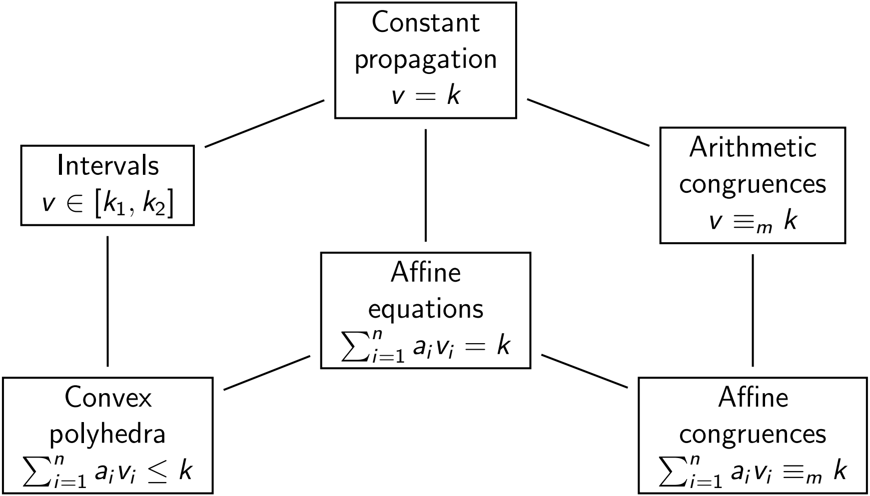
SSA.

Fixed-precision integers.

Bit-manipulating instructions.

Limited signedness information available.

Invariant finding by abstract interpretation



Constant propagation
$v = k$

Intervals
$v \in [k_1, k_2]$

Arithmetic congruences
$v \equiv_m k$

Affine equations
$\sum_{i=1}^{n} a_i v_i = k$

Convex polyhedra
$\sum_{i=1}^{n} a_i v_i \leq k$

Affine congruences
$\sum_{i=1}^{n} a_i v_i \equiv_m k$

Issue (precision loss): Non-linear assignments

Most analyses do not deal with non-linear constructs, such as multiplication and bit-string operations.

Relational analysis of

$$z := x \times y;$$

usually modelled as analysis of

$$z := *;$$

Issue (precision loss): Lack of compositionality

The usual trick for swapping x and y in situ:

$$x := x \oplus y;$$
$$y := x \oplus y;$$
$$x := x \oplus y;$$

If we do, say, interval analysis, processing statement-by-statement, we lose, compared to

$$x, y := y, x;$$

Issue: Respect the modular-arithmetic semantics

Relying on the classical analyses will not do.

```
for ($x$ := 42; $x < 9999999999$; $x$++)
    if ($x = 0$) error();
```

Issue (precision loss): No signedness information

Suppose we know that x is 0110 and also that y is in the interval $[0001, 0011]$.

$$z := x + y;$$

If the variables are unsigned, z must be in $[0111, 1001]$.

If we assume they are signed, we lose all information about z.

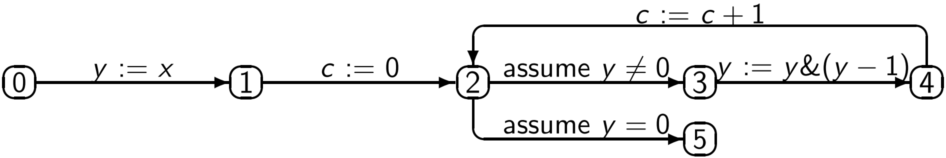
Moreover, we cannot simply adapt the usual interval-analysis rule for multiplication.

Solution: Local bit-precise reasoning

Summarise each basic block bit-precisely.

Solution: Local bit-precise reasoning

Summarise each basic block bit-precisely.



(Wegner's pop count.)

Numeric/bit-twiddling assertions

Sometimes we wish to establish a bit-twiddling result that is best expressed numerically.

```
$\ell_0$:   $c := 0$; $y := x$;
$\ell_1$:   while $(y \neq 0)$
                $y := y \,\&\, (y - 1)$;
                $c := c + 1$;
$\ell_2$:   skip
```

At ℓ_2 we have $c = \sum x_i$. Or, assuming $w = 8$, still in conjunctive form:

$$\varphi : (\sum_{i=0}^{7} x_i \equiv_{256} c_0 + 2c_1 + 4c_2 + 8c_3) \wedge \bigwedge_{i=4}^{7} c_i \equiv_{256} 0$$

Using affine congruence equations

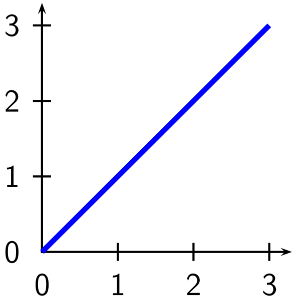
Affinity with arithmetic used in mainstream programming languages (overflow).

Affinity with bit-level analysis.

Computationally “manageable”.

The ascending chain property

Some abstract domains require special attention to guarantee termination of analysis.

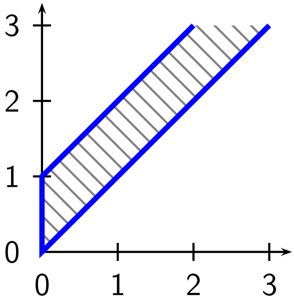


$0 \leq x \wedge x = y$

```
y = x;
while (*) y++;
```

The ascending chain property

Some abstract domains require special attention to guarantee termination of analysis.



$$0 \leq x \wedge x = y$$

$$\sqcup \qquad \text{(convex hull)}$$

$$0 \leq x \wedge y = x + 1$$

```
y = x;
while (*) y++;
```

The ascending chain property

Assume n program variables, word length w and $m = 2^w$.

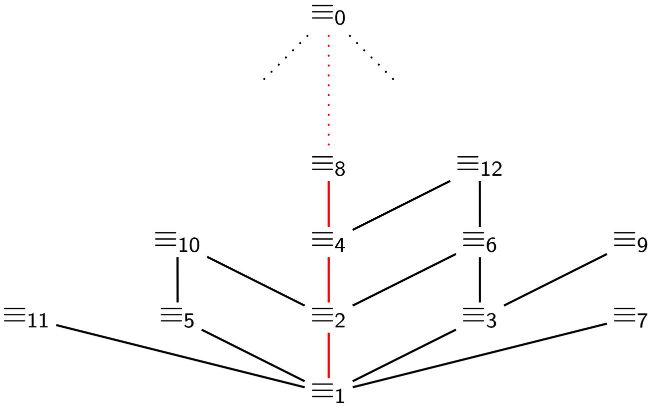
The congruence lattice Aff_m^n (defined later) is a subset of $\mathcal{P}(\mathbb{Z}_m^n)$, namely the affine sets of vectors.

The cardinality of such a set is always a power of 2.

Hence every strictly increasing chain in the congruence lattice has length at most wn (Müller-Olm and Seidl).

Modular arithmetic equivalences

$\equiv_m$ is the equivalence relation defined by $a \equiv_m b$ iff $b - a = km$ for some $k \in \mathbb{Z}$. We shall take $m = 2^w$.



Modulo m affine hull

For a given set S of vectors in $\mathbb{Z}_m^n$ we want the smallest affine superset of S.

The (modulo m) affine hull of $S \subseteq \mathbb{Z}_m^n$ is defined:

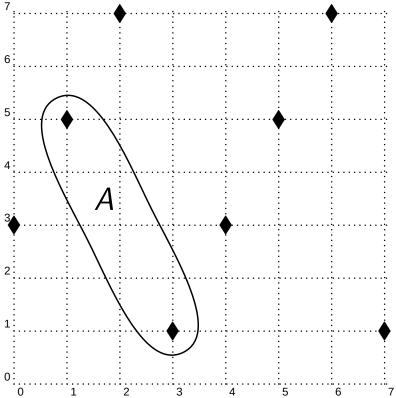
$$\mathrm{aff}_m^n(S) = \left\{ \vec{x} \in \mathbb{Z}_m^n \;\middle|\; \begin{array}{lcll} \vec{x}_1, \ldots, \vec{x}_\ell \in S & \wedge & \lambda_1, \ldots, \lambda_\ell \in \mathbb{Z} & \wedge \\ \sum_{i=1}^{\ell} \lambda_i \equiv_m 1 & \wedge & \vec{x} \equiv_m \sum_{i=1}^{\ell} \lambda_i \vec{x}_i & \end{array} \right\}$$

$$\mathrm{Aff}_m^n = \{ S \subseteq \mathbb{Z}_m^n \mid \mathrm{aff}_m^n(S) = S \}$$

(Aff_m^n is a Moore family, meet is just intersection.)

Affine sets modulo m

$A = \{\langle 0,3\rangle, \langle 1,5\rangle\}$



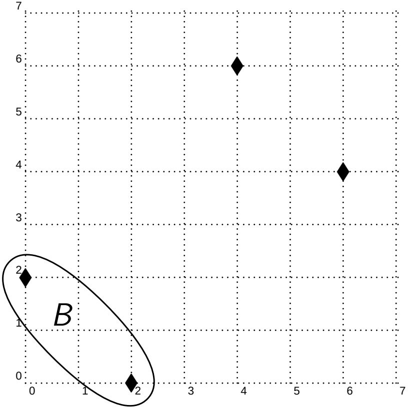
$$\begin{aligned}\mathrm{aff}_8^2(A) &= \{\vec{v} \in \mathbb{Z}_8^2 \mid \lambda_1 + \lambda_2 \equiv_8 1 \wedge \vec{v} \equiv_8 \lambda_1\langle 0,3\rangle + \lambda_2\langle 1,5\rangle\} \\ &= \{\vec{v} \in \mathbb{Z}_8^2 \mid \vec{v} \equiv_8 \langle k, 3+2k\rangle \wedge k \in \mathbb{Z}\}\end{aligned}$$

Or, in equational form: $6x + y + 5 \equiv_8 0$.

Affine sets modulo m

$$B = \{\langle 0, 2\rangle, \langle 2, 0\rangle\}$$

Note that coefficients λ are integers.



So $\mathrm{aff}^n_m(B)$ is not characterised by the equation $x + y + 6 \equiv_8 0$.

Rather we have $x + y + 6 \equiv_8 0 \wedge 4x + 4 \equiv_8 0$ (the second conjunct says x is odd).

Matrix manipulations

The equational form is a convenient way of representing the elements of the congruence lattice.

A system of equations can be captured in matrix form.

Since $\mathbb{Z}_m$ is not a field, Gaussian elimination needs to be adapted with some care.

Legal row operations include:

- Scale a row by an odd factor
- Add a multiple of one row to another row
- Extend the matrix with a multiple of some row

Matrix manipulations

Elder et al (SAS'11) identify Howell matrix form as the appropriate canonical form for systems of congruences modulo 2^w.

For $\mathrm{aff}^n_m(A)$ we had the equation $6x + y + 5 \equiv_8 0$, corresponding to the matrix $\begin{pmatrix} 6 & 1 & 5 \end{pmatrix}$. Howell form requires that leading entries are multiples of 2, so multiply by 3: $\begin{pmatrix} 2 & 3 & 7 \end{pmatrix}$.

Certain consequences are made explicit. Multiplying by 4, we obtain additionally

$$\begin{pmatrix} 2 & 3 & 7 \\ 0 & 4 & 4 \end{pmatrix}$$

Propositional formulas vs congruence equations

The abstraction map $\alpha_m^n : \mathcal{P}(\mathbb{B}^n) \rightarrow \mathsf{Aff}_m^n$ is just aff_m^n.

Concretisation $\gamma_m^n : \mathsf{Aff}_m^n \rightarrow \mathcal{P}(\mathbb{B}^n)$ is defined: $\gamma_m^n(S) = S \cap \mathbb{B}^n$.

A Galois connection.

The abstraction of $\neg x_1 \wedge (x_2 \oplus x_3)$ is $x_1 \equiv 0 \wedge x_2 + x_3 \equiv 1$.

The abstraction of $x_1 \wedge (x_2 \vee x_3)$ is $x_1 \equiv 1$.

(The last shows loss of information, as $(x_1, x_2, x_3) = (1, 0, 0)$ is not a solution to the propositional formula.)

Flowchart programs: Syntax

$$\begin{array}{rl} \text{Expr} & ::= X \mid R \mid -\text{Expr} \mid \text{Expr bop Expr} \\ \text{Guard} & ::= \text{true} \mid \text{false} \mid \text{Expr rop Expr} \mid \text{Guard lop Guard} \\ \text{Stmt} & ::= \text{skip} \mid X := \text{Expr} \mid \text{Stmt}; \text{Stmt} \end{array}$$

with

$$\begin{array}{lcl} \text{rop} & = & \{=, \neq, <, \leq\} \\ \text{bop} & = & \{+, -, \ \& \ , \ | \ , \ll , \gg \} \quad \text{(C style)} \\ \text{lop} & = & \{\wedge, \vee\} \end{array}$$

Flowchart programs

A flowchart program is a tuple $\langle L, X, \ell_0, T \rangle$ where L is a set of labels, X a set of variables, ℓ_0 is the start label, and T is a set of transitions.

Each transition is of the form (ℓ_i, ℓ_j, g, s), with g a guard and s a statement.

Relational semantics, not functional

Usually we start from set of states $\Sigma = X \to R$ and define $\mathcal{E}$: Expr $\to \Sigma \to R$ as a function (expression evaluator) and $\mathcal{S}$: Stmt $\to \Sigma \to \Sigma$ as a function (a state transformer).

Instead we give a "relational" semantics over a double vocabulary.

Advantages:

- We consider programs to take input via program variables, so the semantics should say how, at different points, program states are related to initial states.
- The relational semantics can be bit-blasted in a natural way.
- No need for a so-called collecting semantics.

Relational semantics

More precisely, $\mathcal{S}[\![s]\!] : \Sigma \rightarrow \Sigma$ is replaced by a relation

$$r = \{\langle \sigma(x_1), \ldots, \sigma(x_k), \tau(x_1), \ldots, \tau(x_k) \rangle \mid \sigma \in \Sigma \wedge \tau = \mathcal{S}[\![s]\!](\sigma)\}$$

Henceforth $\mathcal{S}[\![s]\!]$ will denote a relation $\mathcal{S}[\![s]\!] \subseteq R^{2k}$.

Semantic machinery: Composition

If $\vec{a}, \vec{b} \in R^k$ then then $\vec{a} \cdot \vec{b} \in R^{2k}$ is the concatenation of $\vec{a}$ and $\vec{b}$.

The identity relation is $\mathrm{Id} = \{\vec{a} \cdot \vec{a} \mid \vec{a} \in R^k\}$.

If $r_1, r_2 \subseteq R^{2k}$, their composition is
$r_1 \circ r_2 = \{\vec{a} \cdot \vec{c} \mid \vec{b} \in R^k \wedge \vec{a} \cdot \vec{b} \in r_1 \wedge \vec{b} \cdot \vec{c} \in r_2\}$.

If $r_1 \subseteq R^k$ and $r_2 \subseteq R^{2k}$ then let $r_1 \circ r_2 = \{\vec{b} \mid \vec{a} \in r_1 \wedge \vec{a} \cdot \vec{b} \in r_2\}$.

If $\vec{a} = \langle a_1, \ldots, a_k \rangle \in R^k$ $\vec{a}[i] = a_i$.

If $b \in R$ let $\vec{a}[i \mapsto b] = \langle a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_k \rangle$.

Semantic equations

The effect of a guard $g \in \text{Guard}$ is described by

$$\mathcal{S}[\![g]\!] = \{\vec{a} \cdot \vec{a} \mid \vec{a} \in R^k \wedge \mathcal{G}[\![g]\!]\vec{a}\}$$

The effect of a statement $s \in \text{Stmt}$ is described by

$$\begin{array}{rcl} \mathcal{S}[\![\mathsf{skip}]\!] & = & \mathsf{Id} \\ \mathcal{S}[\![x_i := e]\!] & = & \{\vec{a} \cdot \vec{a}[i \mapsto \mathcal{E}[\![e]\!]\vec{a}] \mid \vec{a} \in R^k\} \\ \mathcal{S}[\![s_1 ; s_2]\!] & = & \mathcal{S}[\![s_1]\!] \circ \mathcal{S}[\![s_2]\!] \end{array}$$

$\mathcal{E}$ and $\mathcal{G}$

$$\begin{array}{rcll}
\mathcal{E}[\![x_i]\!]\vec{a} & = & \vec{a}[i] & \\
\mathcal{E}[\![n]\!]\vec{a} & = & n & \\
\mathcal{E}[\![e_1 \odot e_2]\!]\vec{a} & = & (\mathcal{E}[\![e_1]\!]\vec{a}) \odot (\mathcal{E}[\![e_2]\!]\vec{a}) & \text{where } \odot \in \text{bop} \\
& & & \\
\mathcal{G}[\![\text{true}]\!]\vec{a} & = & 1 & \\
\mathcal{G}[\![\text{false}]\!]\vec{a} & = & 0 & \\
\mathcal{G}[\![g_1 \ominus g_2]\!]\vec{a} & = & (\mathcal{G}[\![g_1]\!]\vec{a}) \ominus (\mathcal{G}[\![g_2]\!]\vec{a}) & \text{where } \ominus \in \text{lop} \\
\mathcal{G}[\![e_1 \otimes e_2]\!]\vec{a} & = & (\mathcal{E}[\![e_1]\!]\vec{a}) \otimes (\mathcal{E}[\![e_2]\!]\vec{a}) & \text{where } \otimes \in \text{rop}
\end{array}$$

Semantics of programs

Finally, the semantics of $P = \langle L, X, \ell_0, T\rangle$ can be defined as the set $\{r_\ell \in R^{2k} \mid \ell \in L\}$ of smallest relations r_ℓ such that

1. $\mathrm{Id} \subseteq r_{\ell_0}$
2. $r_{\ell_i} \circ \mathcal{S}[\![g]\!] \circ \mathcal{S}[\![s]\!] \subseteq r_{\ell_j}$ for all $\langle \ell_i, \ell_j, g, s\rangle \in T$.

Each relation r_ℓ is finite and relates states at ℓ_0 to states at ℓ.

The set of reachable states at ℓ is given by the composition $R^k \circ r_\ell$.

From Boolean transfer to congruence/interval/... transfer

King and Søndergaard, VMCAI 2010.

Elder et al, SAS 2011.

Brauer and King, SAS 2011: synthesis of interval transfer.

Synthesis of congruence/interval/... transfer relations makes use of a SAT/SMT solver, using the idea of Reps, Sagiv and Yorsh, VMCAI 2004.

Thank you

... and questions ...