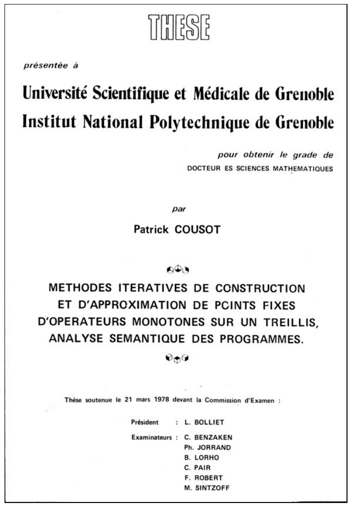
« Forward Relational Infinitary Static Analysis »

Patrick Cousot
Jerome C. Hunsaker Visiting Professor
Massachusetts Institute of Technology
Department of Aeronautics and Astronautics
cousot@mit.edu
www.mit.edu/~cousot

Course 16.399: "Abstract interpretation"
http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/


THESE

présentée à

Université Scientifique et Médicale de Grenoble
Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR ES SCIENCES MATHEMATIQUES

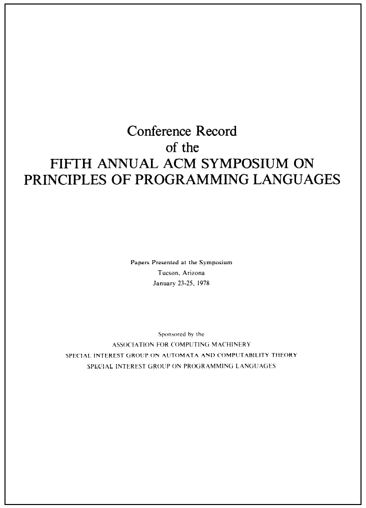
par

Patrick COUSOT

METHODES ITERATIVES DE CONSTRUCTION ET D'APPROXIMATION DE PCINTS FIXES D'OPERATEURS MONOTONES SUR UN TREILLIS, ANALYSE SEMANTIQUE DES PROGRAMMES.

Thèse soutenue le 21 mars 1978 devant la Commission d'Examen :

Président : L. BOLLIET
Examinateurs : C. BENZAKEN
Ph. JORRAND
B. LORHO
C. PAIR
F. ROBERT
M. SINTZOFF


Conference Record
of the
FIFTH ANNUAL ACM SYMPOSIUM ON
PRINCIPLES OF PROGRAMMING LANGUAGES

Papers Presented at the Symposium
Tucson, Arizona
January 23-25, 1978

Sponsored by the
ASSOCIATION FOR COMPUTING MACHINERY
SPECIAL INTEREST GROUP ON AUTOMATA AND COMPUTABILITY THEORY
SPECIAL INTEREST GROUP ON PROGRAMMING LANGUAGES

Back to closure operators

Motivations

- We want to combine abstract analyzes that are defined independently of one another
- Each analysis is defined on the collecting semantics by a closure operator
- Whence the combination of analyzes involves the combination of closure operators
- The reduced product corresponds to the lub of closure operators

- It is the most abstract/less precise analysis which is more precise than the component analyzes (since it is the smallest Moore family containing all abstract properties of the various components)
- The study of the lub of closure operators yields effective methods to approximate this ideal

The lub of closure operators (I)

THEOREM. If $\langle L, \sqsubseteq, \bot, \sqcup\rangle$ is a cpo and $f \in L \xrightarrow{\text{me}} L$ is monotone and extensive then $\mathsf{lfp}^{\dot\sqsubseteq}_f\ \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g$ is the $\dot\sqsubseteq$-least upper closure operator on L greater than or equal to f. ■

PROOF. – Because $\langle L, \sqsubseteq, \bot, \sqcup\rangle$ is a cpo, $(L \xrightarrow{\text{me}} L)$ is a cpo pointwise

- $\lambda g \cdot g \circ g$ is a function of $(L \xrightarrow{\text{me}} L)$ into $(L \xrightarrow{\text{me}} L)$ since the composition of monotonic and extensive functions is monotonic and extensive
- $\lambda g \cdot g \circ g \in (L \xrightarrow{\text{me}} L) \mapsto (L \xrightarrow{\text{me}} L)$ is monotonic. Indeed if $g_1 \dot\sqsubseteq g_2$ then by def. of a pointwise ordering $g_1 \circ g_2 \dot\sqsubseteq g_2 \circ g_2$ and by monotony of g_1, $g_1 \circ g_1 \dot\sqsubseteq g_1 \circ g_2$ so by transitivity $g_1 \circ g_1 \dot\sqsubseteq g_2 \circ g_2$ proving that $\lambda g \cdot g \circ g \in (L \xrightarrow{\text{me}} L) \xrightarrow{\text{m}} (L \xrightarrow{\text{me}} L)$

- If $f \in L \xrightarrow{\text{me}} L$ is monotone and extensive then $f \dot\sqsubseteq f \circ f$ so f is a prefixpoint of $\lambda g \cdot g \circ g$ considered as a function of $(L \xrightarrow{\text{me}} L) \xrightarrow{\text{m}} (L \xrightarrow{\text{me}} L)$
- It follows by Knaster-Tarski on cpos that $\mathsf{lfp}^{\dot\sqsubseteq}_f\ \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g$ does exists.
- If we consider the transfinite iterates $\langle g^\delta, \delta \in \mathbb{O}\rangle$ of $\lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g$ from f, the are all monotone and extensive since $g^0 = f \in L \xrightarrow{\text{me}} L$, if $g^\delta \in L \xrightarrow{\text{me}} L$ then $g^{\delta+1} = g^\delta \circ g^\delta \in L \xrightarrow{\text{me}} L$ as shown above and if $\forall \beta < \lambda : g^\beta \in\in L \xrightarrow{\text{me}} L$ implies $g^\lambda = \dot\bigsqcup_{\beta<\lambda} g^\beta$ from limit ordinal so in particular $\mathsf{lfp}^{\dot\sqsubseteq}_f\ \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g = g^\epsilon$ where ϵ is the rank of the iterates is certainly monotone and extensive
- Moreover, by the fixpoint property, $g^\epsilon = g^\epsilon \circ g^\epsilon$ proving $\mathsf{lfp}^{\dot\sqsubseteq}_f\ \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g$ idempotent whence a closure operator. Since the iterates are increasing it is also greater than of equal to f
- If ρ is another closure operator on L greater than of equal to f we have $f \dot\sqsubseteq \rho$ and $\rho = \rho \circ \rho$ so by Knaster-Tarski $\mathsf{lfp}^{\dot\sqsubseteq}_f\ \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g = \dot\bigsqcap\{g \in L \xrightarrow{\text{me}} L \mid f \dot\sqsubseteq g \wedge g = g \circ g\} \dot\sqsubseteq \rho$ by def. glbs □

COROLLARY. Let $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap\rangle$ be a complete lattice. The lub of a set F of upper closure operators in the complete lattice of closure operators on L is

$$\mathsf{lfp}^{\dot\sqsubseteq}_{\dot\bigsqcup F}\ \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g$$

■

PROOF. Let lub F be this lub. We have $\dot\bigsqcup F \dot\sqsubseteq$ lub F and, because $\dot\bigsqcup F$ is monotonic and extensive, lub F is the least closure operator $\dot\sqsubseteq$-greater than of equal to $\dot\bigsqcup F$, whence, by the previous theorem, $\mathsf{lfp}^{\dot\sqsubseteq}_{\dot\bigsqcup F}\ \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g$. □

The lub of closure operators (II)

THEOREM. If $\langle L, \sqsubseteq, \bot, \sqcup\rangle$ is a cpo and $f \in L \xrightarrow{\text{me}} L$ is monotone and extensive then $\mathsf{lfp}^{\dot{\sqsubseteq}}_f \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g = \lambda x \cdot \mathsf{lfp}^{\sqsubseteq}_x f$ ■

PROOF. – Define $g = \mathsf{lfp}^{\dot{\sqsubseteq}}_f \lambda g' \in L \xrightarrow{\text{me}} L \cdot g' \circ g'$. We just showed that g is the $\dot{\sqsubseteq}$-least closure operator which is greater than or equal to f.

– Given any $x \in L$, x is a prefixpoint of $f \in L \xrightarrow{\text{me}} L$ by extensivity. Since $\langle L, \sqsubseteq, \bot, \sqcup\rangle$ is a cpo and f is monotone, $\mathsf{lfp}^{\sqsubseteq}_x f$ does exists, whence $\lambda x \cdot \mathsf{lfp}^{\sqsubseteq}_x f$ is well-defined.

– Define $h \stackrel{\text{def}}{=} \mathsf{lfp}^{\sqsubseteq}_x f$. $h(x)$ is the limit of the transfinite iterates of f starting from the prefixpoint x, so we have shown h to be an upper closure operator (in the constructive proof of Tarski theorem).

– Since the iterates are increasing $\forall x \in L : f(x) \sqsubseteq h(x)$ so $f \mathrel{\dot{\sqsubseteq}} h$. It follows that h is a closure operator on L which is greater than or equal to f, proving that $g \mathrel{\dot{\sqsubseteq}} h$.

– Let $\langle h^\delta, \delta \in \mathbb{O}\rangle$ be the iterates of $\mathsf{lfp}^{\sqsubseteq}_x f$ and $\langle g^\delta, \delta \in \mathbb{O}\rangle$ be the iterates of $\mathsf{lfp}^{\dot{\sqsubseteq}}_f \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g$. Let us prove, by transfinite induction, that $\forall \delta \in \mathbb{O} : h^\delta \sqsubseteq g^\delta(x)$.

- $h^0 = x \sqsubseteq f(x) = g^0(x)$
- If $h^\delta \sqsubseteq g^\delta(x)$ then $g^\delta(h^\delta) \sqsubseteq g^\delta(g^\delta(x))$ since g^δ is monotone. But $f \mathrel{\dot{\sqsubseteq}} g^\delta$ since the iterates are increasing so $f(h^\delta) \sqsubseteq g^\delta(h^\delta)$. By transitivity and def. of the iterates $h^{\delta+1} = f(h^\delta) \sqsubseteq g^\delta(g^\delta(x)) = g^{\delta+1}(x)$.
- For a limit ordinal λ, if $\forall \beta < \lambda : h^\beta \sqsubseteq g^\beta(x)$ then $h^\lambda = \bigsqcup_{\beta<\lambda} h^\beta \sqsubseteq \bigsqcup_{\beta<\lambda} g^\beta(x) = (\dot{\bigsqcup\limits_{\beta<\lambda}} g^\beta)(x) = g^\lambda(x)$, by def. of the iterates, existence of the lubs in the cpo and and def. lubs.

– Let ϵ and ϵ' be the rank of the respective iterates. Then $h(x) = \mathsf{lfp}^{\sqsubseteq}_x f = h^\epsilon = h^{\max(\epsilon,\epsilon')} = g^{\max(\epsilon,\epsilon')}(x) = g^{\epsilon'}(x) = (\mathsf{lfp}^{\dot{\sqsubseteq}}_f \lambda g' \in L \xrightarrow{\text{me}} L \cdot g' \circ g')(x) = g(x)$ so that $h \mathrel{\dot{\sqsubseteq}} g$

– By antisymmetry, we conclude that $h = g$.

□

COROLLARY. Let $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap\rangle$ be a complete lattice. The lub of a set F of upper closure operators in the complete lattice of closure operators on L is $\lambda x \cdot \mathsf{lfp}^{\sqsubseteq}_x \dot{\bigsqcup} F$. ■

PROOF. The lub has been shown (on page 7) to be $\mathsf{lfp}^{\dot{\sqsubseteq}}_{\dot{\bigsqcup} F} \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g = \lambda x \cdot \mathsf{lfp}^{\sqsubseteq}_x \dot{\bigsqcup} F$. □

Iterative reduced product

Union of abstract domains

THEOREM. If $\langle L,\, \sqsubseteq,\, \bot,\, \top,\, \sqcup,\, \sqcap\rangle$ and $\langle M,\, \leq,\, 0,\, 1,\, \vee,\, \wedge\rangle$ are complete lattices and $\forall i \in \Delta : \langle L,\, \sqsubseteq\rangle \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} \langle M,\, \leq\rangle$ then $\langle L,\, \sqsubseteq\rangle \underset{\dot{\bigvee}_{i\in\Delta} \alpha_i}{\overset{\dot{\bigsqcap}_{i\in\Delta} \gamma_i}{\leftrightarrows}} \langle M,\, \leq\rangle$ ■

PROOF. For all $x \in L$ and $y \in M$, we have

$$
\begin{aligned}
& (\dot{\bigvee_{i\in\Delta}} \alpha_i)(x) \leq y \\
\Longleftrightarrow\ & (\bigvee_{i\in\Delta} \alpha_i(x)) \leq y && \text{≀pointwise def. } \dot{\vee}\text{≀} \\
\Longleftrightarrow\ & \forall i \in \Delta : \alpha_i(x) \leq y && \text{≀def. lub≀} \\
\Longleftrightarrow\ & \forall i \in \Delta : x \sqsubseteq \gamma_i(y) && \text{≀Galois connection≀} \\
\Longleftrightarrow\ & x \sqsubseteq \bigsqcap_{i\in\Delta} \gamma_i(y) && \text{≀def. glb≀} \\
\Longleftrightarrow\ & x \sqsubseteq (\dot{\bigsqcap_{i\in\Delta}} \gamma_i)(y) && \text{≀pointwise def. } \dot{\sqcap}\text{≀}
\end{aligned}
$$

□

– Will discover the information found by all component analyzes
– Usefull in theory, not much in practice

Cartesian product of abstract domains

THEOREM. If $\langle L,\, \sqsubseteq,\, \bot,\, \top,\, \sqcup,\, \sqcap\rangle$ is a complete lattice and $\langle M_i,\, \leq_i\rangle$ is a family of posets then $\forall i \in \Delta : \langle L,\, \sqsubseteq\rangle \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} \langle M,\, \leq\rangle$ implies that $\langle L,\, \sqsubseteq\rangle \underset{\lambda x \cdot \prod_{i\in\Delta} \alpha_i(x)}{\overset{\lambda \vec{a} \cdot \bigsqcap_{i\in\Delta} \gamma_i(\vec{a}_i)}{\leftrightarrows}} \langle \prod_{i\in\Delta} M_i,\, \leq\rangle$ where $\leq$ is the componentwise ordering for the $\leq_i$, $i \in \Delta$ ■

PROOF. For all $x \in L$ and $\vec{y} \in \prod_{i\in\Delta} M_i$, we have

$$
\begin{aligned}
& \prod_{i\in\Delta} \alpha_i(x) \leq \vec{y} \\
\Longleftrightarrow\ & \forall i \in \Delta : \alpha_i(x) \leq_i \vec{y}_i && \text{≀pointwise def. } \leq\text{≀} \\
\Longleftrightarrow\ & \forall i \in \Delta : x \sqsubseteq \gamma_i(\vec{y}_i) && \text{≀since } \langle L,\, \sqsubseteq\rangle \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} \langle M,\, \leq\rangle\text{≀} \\
\Longleftrightarrow\ & x \sqsubseteq \bigsqcap_{i\in\Delta} \gamma_i(\vec{y}_i) && \text{≀def. glb≀}
\end{aligned}
$$

□

– The cartesian product of abstractions discovers in one shot the information found separately by the component analyzes
– The problem is that we do not learn more by performing all analyzes simultaneously than by performing them one after another and finally taking their conjunctions

The reduction operator

Theorem. Let $\langle L, \sqsubseteq\rangle \xleftrightarrows[\alpha]{\gamma} \langle A, \leq\rangle$ where $\langle A, \leq, 0, 1, \vee, \wedge\rangle$ is a complete lattice. Define

$$\rho(a) \stackrel{\text{def}}{=} \bigwedge\{a' \in A \mid \gamma(a) \sqsubseteq \gamma(a')\}$$

then ρ is a lower closure operator and

$$\langle L, \sqsubseteq\rangle \xleftrightarrows[\rho\circ\alpha]{\gamma} \langle \rho(A), \leq\rangle$$

■

Proof. – ρ is reductive since $a \in \{a' \in A \mid \gamma(a) \sqsubseteq \gamma(a')\}$ by reflexivity and so $\rho(a) \sqsubseteq a$ by def. glb $\wedge$.

– If $a \sqsubseteq b$ then $\gamma(a) \sqsubseteq \gamma(b)$ so $\gamma(b) \sqsubseteq \gamma(b')$ implies $\gamma(a) \sqsubseteq \gamma(b')$ whence $\{b' \mid \gamma(b) \sqsubseteq \gamma(b')\} \subseteq \{a' \mid \gamma(a) \sqsubseteq \gamma(a')\}$ so $\rho(a) = \bigwedge\{a' \mid \gamma(a) \sqsubseteq \gamma(a')\} \sqsubseteq \bigwedge\{b' \mid \gamma(b) \sqsubseteq \gamma(b')\} = \rho(b)$.

– For idempotence, we have

$$\begin{aligned}
& \rho(\rho(a)) \\
= \; & \bigwedge\{a' \in A \mid \gamma(\rho(a)) \sqsubseteq \gamma(a')\} && \langle\!\langle \text{def. } \rho \rangle\!\rangle \\
= \; & \bigwedge\{a' \in A \mid \gamma(\bigwedge\{a'' \in A \mid \gamma(a) \sqsubseteq \gamma(a'')\}) \sqsubseteq \gamma(a')\} && \langle\!\langle \text{def. } \rho \rangle\!\rangle \\
= \; & \bigwedge\{a' \in A \mid \bigwedge\{\gamma(a'') \in A \mid \gamma(a) \sqsubseteq \gamma(a'')\} \sqsubseteq \gamma(a')\} && \langle\!\langle \gamma \text{ preserves meets} \rangle\!\rangle \\
= \; & \bigwedge\{a' \in A \mid \gamma(a) \sqsubseteq \gamma(a')\} && \langle\!\langle \text{since } \gamma(a) = \bigwedge\{\gamma(a'') \in A \mid \gamma(a) \sqsubseteq \gamma(a'')\} \text{ by reflexivity and def. glb} \rangle\!\rangle \\
& \rho(a) && \langle\!\langle \text{def. } \rho \rangle\!\rangle
\end{aligned}$$

– By the Galois connection, $x \sqsubseteq \gamma(y)$ implies $\alpha(x) \sqsubseteq y$ implies $\rho \circ \alpha(x) \sqsubseteq y$ since ρ is a closure operator and $y = \rho(y)$ is closed

– Inversely if $x \in L$ and $y \in \rho(A)$ then

$$\begin{aligned}
& \rho \circ \alpha(x) \sqsubseteq y \\
\Longrightarrow \; & \bigwedge\{a' \in A \mid \gamma(\alpha(x)) \sqsubseteq \gamma(a')\} \sqsubseteq y && \langle\!\langle \text{def. } \circ \text{ and } \rho \rangle\!\rangle
\end{aligned}$$

$$\begin{aligned}
\Longrightarrow \; & \gamma(\bigwedge\{a' \in A \mid \gamma(\alpha(x)) \sqsubseteq \gamma(a')\}) \sqsubseteq \gamma(y) && \langle\!\langle \gamma \text{ monotone} \rangle\!\rangle \\
\Longrightarrow \; & (\bigsqcap\{\gamma(a') \in A \mid \gamma(\alpha(x)) \sqsubseteq \gamma(a')\}) \sqsubseteq \gamma(y) && \langle\!\langle \gamma \text{ preserves existing glbs} \rangle\!\rangle \\
\Longrightarrow \; & \gamma \circ \alpha(x) \sqsubseteq \gamma(y) && \langle\!\langle \text{reflexivity for } a' = \alpha(x) \text{ and def. glb} \rangle\!\rangle \\
\Longrightarrow \; & x \sqsubseteq \gamma(y) && \langle\!\langle \gamma \circ \alpha \text{ extensive and transitivity} \rangle\!\rangle
\end{aligned}$$

□

– The reduction operator brings in the abstract the conjunction of properties we would have in the concrete.

– So information can flow from any component analysis to all others

– Whence, this is more precise than the cartesian product

Theorem. $\gamma = \gamma \circ \rho$ ■

Proof. or all $x \in L$:

$$\begin{aligned}
& \gamma \circ \rho \circ \alpha(x) \\
= \; & \gamma(\dot\bigwedge\{\vec{a}' \mid \gamma(\alpha(x)) \sqsubseteq \gamma(\vec{a}')\}) && \langle\!\langle \text{def. } \rho \rangle\!\rangle \\
= \; & \bigsqcap\{\gamma(\vec{a}') \mid \gamma(\alpha(x)) \sqsubseteq \gamma(\vec{a}')\}) && \langle\!\langle \gamma \text{ preserves meets} \rangle\!\rangle \\
= \; & \gamma(\alpha(x)) && \langle\!\langle \text{choosing } \vec{a}' = \alpha(x) \text{ and def. glb} \rangle\!\rangle
\end{aligned}$$

and so

$$\begin{aligned}
\gamma \; & = \gamma \circ \alpha \circ \gamma && \langle\!\langle \text{Galois connection} \rangle\!\rangle \\
& = \gamma \circ \rho \circ \alpha \circ \gamma && \langle\!\langle \text{since } \gamma \circ \alpha = \gamma \circ \rho \circ \alpha \rangle\!\rangle \\
& \sqsubseteq \gamma \circ \rho && \langle\!\langle \alpha \circ \gamma \text{ is reductive and monotony} \rangle\!\rangle
\end{aligned}$$

Moreover ρ is a lower closure operator on $\langle \prod_{i\in\Delta} A_i, \sqsubseteq_\Delta\rangle$ so ρ is reductive ($\rho \mathrel{\dot\sqsubseteq_\Delta} 1$) whence by monotony $\gamma \circ \rho \mathrel{\dot\sqsubseteq} \gamma$. By antisymmetry, $\gamma \circ \rho = \gamma$. □

The reduced product

THEOREM. Assume that $\langle L, \sqsubseteq\rangle$ is a poset, $\langle A_i, \leq_i, 0_i, 1_i, \vee_i, \wedge_i\rangle$, $i \in \Delta$ are complete lattices such that $\forall i \in \Delta : \langle L, \sqsubseteq\rangle \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} \langle A_i, \leq_i\rangle$. Define $\leq_\Delta$ componentwise in terms of the $\leq_i$, $i \in \Delta$. Let $\gamma = \lambda\vec{a} \cdot \prod_{i\in\Delta} \gamma_i(\vec{a}_i)$ and $\alpha = \lambda x \cdot \prod_{i\in\Delta} \alpha_i(x)$ so that $\langle L, \sqsubseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \prod_{i\in\Delta} A_i, \leq_\Delta\rangle$ Now let $\rho = \lambda\vec{a} \cdot \sqcap\{\vec{a}' \mid \gamma(\vec{a}) \sqsubseteq \gamma(\vec{a}')\}$ so that $\langle L, \sqsubseteq\rangle \underset{\rho\circ\alpha}{\overset{\gamma}{\leftrightarrows}} \langle\rho(\prod_{i\in\Delta} A_i), \leq_\Delta\rangle$. Then we have:

$\langle\rho(\prod_{i\in\Delta} A_i), \leq_\Delta\rangle$ is the reduced product of the $\langle A_i, \leq_i\rangle$, $i \in \Delta$ ■

PROOF. – $\langle\rho(\prod_{i\in\Delta} A_i), \leq_\Delta\rangle$ is more precise that the $\langle A_i, \leq_i\rangle$ in that $\gamma \circ (\rho \circ \prod_{i\in\Delta} \alpha_i) \mathrel{\dot{\leq}} \gamma_i \circ \alpha_i$

$$
\begin{array}{ll}
\gamma \circ (\rho \circ \prod\limits_{i\in\Delta} \alpha_i)(x) & \\
= \gamma(\rho(\prod\limits_{i\in\Delta} \alpha_i(x))) & \wr\text{def. } \circ, \textstyle\prod\wr \\
= \bigwedge\{\gamma(\vec{a}') \mid \gamma(\prod\limits_{i\in\Delta} \alpha_i(x)) \sqsubseteq \gamma(\vec{a}')\} & \wr\gamma \text{ preserves existing meets}\wr \\
= \gamma(\prod\limits_{i\in\Delta} \alpha_i(x)) & \wr\text{choosing } \vec{a}' = \textstyle\prod_{i\in\Delta} \alpha_i(x) \text{ and def. glb}\wr \\
= \bigwedge\limits_{k\in\Delta} \gamma_k((\prod\limits_{i\in\Delta} \alpha_i(x))_k) & \wr\text{def. } \gamma\wr \\
= \bigwedge\limits_{k\in\Delta} \gamma_k(\alpha_k(x)) & \wr\text{def. index selection}\wr
\end{array}
$$

$\leq_\Delta \gamma_i \circ \alpha_i(x)$ $\wr$for any $i \in \Delta$, by def. glb$\wr$

– Let be given any other $\langle L, \sqsubseteq\rangle \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} \langle M, \leq_\Delta\rangle$ which is more precise than the $\langle A_i, \leq_i\rangle$, $i \in \Delta$ in that $\forall i \in \Delta : \gamma' \circ \alpha' \mathrel{\dot{\sqsubseteq}} \gamma_i \circ \alpha_i$. So $\gamma' \circ \alpha' \mathrel{\dot{\sqsubseteq}} \dot{\bigwedge}_{i\in\Delta} \gamma_i \circ \alpha_i = \gamma \circ (\rho \circ \prod_{i\in\Delta} \alpha_i)$ as just shown above, so $\langle\rho(\prod_{i\in\Delta} A_i), \leq_\Delta\rangle$ is less precise than $\langle M, \leq_\Delta\rangle$

– In conclusion, $\langle\rho(\prod_{i\in\Delta} A_i), \leq_\Delta\rangle$ is:
  - more precise than the $\langle A_i, \leq_i\rangle$, $i \in \Delta$
  - less precise than any other $\langle M, \leq_\Delta\rangle$ which is more precise than the $\langle A_i, \leq_i\rangle$, $i \in \Delta$

  whence it is the less precise abstraction of $\langle L, \sqsubseteq\rangle$ which is more precise than the $\langle A_i, \leq_i\rangle$, $i \in \Delta$ which was precisely defined as the reduced product of the $\langle A_i, \leq_i\rangle$, $i \in \Delta$.

□

– The advantage of the reduced product over the cartesian product of analyses is that each analysis in the abstract composition benefits from the information brought by the other analyses

– For example a sign analysis establishing $x = 0$ can be reduced by a parity analysis showing that x is odd to yield ff that is "unreachable program point"

– We must elaborate on the present non-constructive definition of the reduction operator to get algorithms for constructing reduced products of abstract domains

The reduction operator in fixpoint form

Definition. Let

- $\langle L,\ \sqsubseteq,\ \bot,\ \top,\ \sqcup,\ \sqcap\rangle$ be a complete lattice
- Let $\langle \Delta,\ \leq\rangle$ be a totally ordered set of indices [1]
- $\langle A_i,\ \leq_i,\ 0_i,\ 1_i,\ \vee_i,\ \wedge_i\rangle$, $i \in \Delta$ be complete lattices
- $\langle L,\ \sqsubseteq\rangle \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} \langle A_i,\ \sqsubseteq_i\rangle$ for all $i \in \Delta$

Define

- $\alpha \in L \mapsto \prod_{i\in\Delta} A_i$ as $\alpha(x) \stackrel{\text{def}}{=} \prod_{i\in\Delta} \alpha_i(x)$
- $\gamma \in \prod_{i\in\Delta} A_i \mapsto L$ by $\gamma(\vec{a}) \stackrel{\text{def}}{=} \prod_{i\in\Delta} \gamma_i(\vec{a}_i)$

[1] naming abstract domains, in practice Δ is finite.

- $\rho \in \prod_{i\in\Delta} A_i \mapsto \prod_{i\in\Delta} A_i$ by $\dot{\bigwedge}\{\vec{a}' \mid \gamma(\vec{a}) \sqsubseteq \gamma(\vec{a}')\}$
- $\rho_{ij} \in \langle A_i \times A_j,\ \leq_{ij}\rangle \mapsto \langle A_i \times A_j,\ \leq_{ij}\rangle$ be $\rho_{ij}(\langle x,\ y\rangle) \stackrel{\text{def}}{=}$ let $c = \gamma_i(x) \sqcap \gamma_j(y)$ in $\langle \alpha_i(c),\ \alpha_j(c)\rangle$ were $\leq_{ij}$ is defined pointwise, for all $i, j \in \Delta$, $i < j$
- $\rho_{\vec{ij}} \in \langle \prod_{i\in\Delta} A_i,\ \leq_\Delta\rangle \mapsto \langle \prod_{i\in\Delta} A_i,\ \leq_\Delta\rangle$ be $\rho_{\vec{ij}}(\vec{x}) \stackrel{\text{def}}{=}$ let $\langle x'_i,\ x'_j\rangle \stackrel{\text{def}}{=} \rho_{ij}(\langle x_i,\ x_j\rangle)$ in $\vec{x}[i := x'_i][j := x'_j]$ where $\leq_\Delta$ is defined pointwise and $\vec{x}[i := a]_i = a$ and $\vec{x}[i := a]_j = \vec{x}_j$ when $i \neq j$.
- $\rho^* \in \langle \prod_{i\in\Delta} A_i,\ \leq_\Delta\rangle \mapsto \langle \prod_{i\in\Delta} A_i,\ \leq_\Delta\rangle$ is
  $\rho^* \stackrel{\text{def}}{=} \lambda\vec{a}\,.\,\mathsf{gfp}_{\vec{a}}^{\leq_\Delta}\ \dot{\bigwedge}_{i,j\in\Delta; i<j}\, \rho_{\vec{ij}}$ ■

Theorem. ρ^* is the glb of the $\{\rho_{\vec{ij}} \mid i, j \in \Delta \wedge i < j\}$ in the complete lattice of lower closure operators ■

Proof. By dual of the definition of the lub of upper closures operators on a complete lattice (on page 7) and its equivalent definition (on page 11). □

Theorem. $\gamma = \gamma \circ \rho = \gamma \circ \rho^*$ ■

Proof. We have

$$\begin{aligned}
&\gamma(\vec{a})\\
=\ & \bigsqcap_{k\in\Delta\setminus\{i,j\}} \gamma_k(\vec{a}_k) \sqcap \gamma_i(\vec{a}_i) \sqcap \gamma_j(\vec{a}_j) && \wr\text{def. } \gamma\wr\\
\sqsubseteq\ & \bigsqcap_{k\in\Delta\setminus\{i,j\}} \gamma_k(\vec{a}_k) \sqcap \gamma_i \circ \alpha_i(\gamma_i(\vec{a}_i) \sqcap \gamma_j(\vec{a}_j)) \sqcap \gamma_j \circ \alpha_j(\gamma_i(\vec{a}_i) \sqcap \gamma_j(\vec{a}_j)) && \wr\text{since}\\
& \gamma_i \circ \alpha_i \text{ and } \gamma_j \circ \alpha_j \text{ are extensive}\wr
\end{aligned}$$

$$\begin{aligned}
=\ & \gamma(\vec{a}[i := \alpha_i(\gamma_i(\vec{a}_i) \sqcap \gamma_j(\vec{a}_j))][j := \alpha_i(\gamma_i(\vec{a}_i) \sqcap \gamma_j(\vec{a}_j))] && \wr\text{def. } \gamma\wr\\
=\ & \gamma(\rho_{\vec{ij}}(\vec{a})) && \wr\text{def. } \rho_{\vec{ij}}\wr
\end{aligned}$$

It immediately follows that for all $\vec{a} \in \prod_{i\in\Delta} A_i$, we have $\rho_{\vec{ij}}(\vec{a}) \in \{\vec{a}' \mid \gamma(\vec{a}) \sqsubseteq \gamma(\vec{a}')\}$ proving $\rho(\vec{a}) = \bigwedge\{\vec{a}' \mid \gamma(\vec{a}) \sqsubseteq \gamma(\vec{a}')\} \leq_\Delta \rho_{\vec{ij}}(\vec{a})$ by def. glb, whence $\rho \mathrel{\dot{\leq}_\Delta} \rho_{\vec{ij}}$, pointwise. If follows that ρ is a lower bound of the $\{\rho_{\vec{ij}} \mid i, j \in \Delta \wedge i < j\}$ so by the characterization of their glb on page 27, $\rho \mathrel{\dot{\leq}_\Delta} \rho^*$. By monotony, $\gamma \circ \rho \sqsubseteq \gamma \circ \rho^*$.

For all $\vec{a} \in \prod_{i\in\Delta} A_i$, we have $\rho^*(\vec{a}) = \mathsf{gfp}_{\vec{a}}^{\sqsubseteq_\Delta}\ \dot{\bigwedge}_{i,j\in\Delta; i<j}\, \rho_{\vec{ij}}$ whence $\rho^*(\vec{a}) \sqsubseteq_\Delta \vec{a}$. So by monotony and def. $\circ$, $\gamma \circ \rho^* \mathrel{\dot{\sqsubseteq}} \gamma$.

We conclude, using the theorem on page 20, that $\gamma = \gamma \circ \rho \mathrel{\dot{\sqsubseteq}} \gamma \circ \rho^* \mathrel{\dot{\sqsubseteq}} \gamma$ whence $\gamma = \gamma \circ \rho = \gamma \circ \rho^*$ by antisymmetry. □

The reduced product (iterative form)

THEOREM. $\langle \rho^\star(\prod_{i\in\Delta} A_i),\ \leq_\Delta\rangle$ is the reduced product of the $\langle A_i,\ \leq_i\rangle$, $i \in \Delta$ ■

PROOF. Let us first prove that we have $\langle L,\ \sqsubseteq\rangle \underset{\rho^\star\circ\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \rho(\prod_{i\in\Delta} A_i),\ \leq_\Delta\rangle$. Indeed for all $x \in L$ and $y \in \rho(\prod_{i\in\Delta} A_i)$, we have

$$
\begin{array}{ll}
\quad \rho^\star \circ \alpha(x) \leq_\Delta y & \\
\Longrightarrow \gamma \circ \rho^\star \circ \alpha(x) \sqsubseteq \gamma(y) & \wr \gamma \text{ monotone} \int \\
\Longrightarrow \gamma \circ \alpha(x) \sqsubseteq \gamma(y) & \wr \text{since } \gamma = \gamma \circ \rho^\star \int \\
\Longrightarrow x \sqsubseteq \gamma(y) & \wr \text{since } \gamma \circ \alpha \text{ is extensive} \int \\
\Longrightarrow \alpha(x) \leq_\Delta y & \wr \text{Galois connection} \int \\
\Longrightarrow \rho^\star \circ \alpha(x) \leq_\Delta y & \wr \rho^\star \circ \text{ reductive} \int
\end{array}
$$

We have $\langle L,\ \sqsubseteq\rangle \underset{\rho\circ\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \rho(\prod_{i\in\Delta} A_i),\ \leq_\Delta\rangle$ and $\langle L,\ \sqsubseteq\rangle \underset{\rho^\star\circ\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \rho(\prod_{i\in\Delta} A_i),\ \leq_\Delta\rangle$ whence $\rho \circ \alpha = \rho^\star \circ \alpha$ by unicity of the adjoint in Galois connections so that the reduced product of the $\langle A_i,\ \leq_i\rangle$, $i \in \Delta$ which has been shown to be $\langle \rho(\prod_{i\in\Delta} A_i),\ \leq_\Delta\rangle$ is also $\langle \rho^\star(\prod_{i\in\Delta} A_i),\ \leq_\Delta\rangle$. □

Implementing the reduced product of abstract domains

– Assume we have implemented several analyzes using abstract domains $\langle A_i,\ \leq_i\rangle$, $i \in \Delta$

– We can run them all simultaneously, by considering the cartesian product $\langle \prod_{i\in\Delta} A_i,\ \leq_\Delta\rangle$

– There is no advantage in doing so since the analyzes remain independent of one another

– However, if we use their reduced product, $\langle \rho(\prod_{i\in\Delta} A_i),\ \leq_\Delta\rangle$, each analysis can benefit from the information gathered by the others

– To do so we just have to implement ρ (or use any upper-approximation if this is too hard) and replace any abstract information $\vec{a} \in \prod_{i\in\Delta} A_i$ appearing during the analysis by $\rho(\vec{a})$

– This is sound since nothing is changed in the concrete (recall $\gamma = \gamma \circ \rho$)

– The design and implementation of ρ is a difficult task when $|\Delta|$ is large

– The design and implementation of ρ has to be entirely redone when a new abstract domain is added to the list Δ

The reduced product of three abstract domains or more

– We can consider instead the iterative reduction $\langle\rho^\star(\prod_{i\in\Delta} A_i),\ \leq_\Delta\rangle\rangle$ [2]

– We then consider the reductions, two by two:

$$\rho_{ij}, \quad i,j \in \Delta, i < j$$

– The computation of $\rho_{\vec{ij}}$ and the fixpoint computation of $\rho^\star$ can be implemented once for all

[2] Indeed this makes not difference when $|\Delta| \leq 2$.

– The advantage of this approach are that:
  - The pairwise reductions ρ_{ij}, $i, j \in \Delta, i < j$ are much simpler to design and implement than the global reduction ρ [3]
  - The iterative implementation [4] is equally precise (recall $\gamma \circ \rho = \gamma \circ \rho^\star$)
  - Termination of the fixpoint computation may have to be ensured by a narrowing [5]

[3] Which involves the evaluation of an hidden fixpoint anyway.
[4] replacing any abstract information $\vec{a} \in \prod_{i\in\Delta} A_i$ appearing during the analysis by $\rho^\star(\vec{a})$
[5] in which case the exact reduction ρ was certainly quite complex if not impossible to compute.

- The addition of a new abstract domain only requires
  · the design and implementation of its reduction with the existing ones,
  · without any modification of the existing reductions

The generic forward abstract interpreter with reduced product

Implementation of the ternary iterated reduction

```
(* red123.mli *)
open Avalues1
open Avalues2
open Avalues3
val reduce : (Avalues1.t * Avalues2.t * Avalues3.t)
                         -> (Avalues1.t * Avalues2.t * Avalues3.t)
```

```
(* red123.ml *)
open Red12
open Red23
open Red13
open Avalues1
open Avalues2
open Avalues3
```



```
open Trace
(* printing *)
let print (x,y,z) =
 (print_string "("; Avalues1.print x; print_string ",";
                    Avalues2.print y; print_string ",";
                    Avalues3.print z; print_string ")")
let reduce' (a, b, c) =
 let (a', b') = Red12.reduce (a, b) in
   let (b'', c') = Red23.reduce (b', c) in
     let (a'', c'') = Red13.reduce (a', c') in
       (a'', b'', c'')
let rec reduce t =
  if trace_red () then (print t; print_string " -> ");
  let t' = (reduce' t) in
    if (t = t') then
      (if trace_red () then (print_string "stable\n"); t)
    else (reduce t')
```


Note that we may have to include a narrowing to ensure termination of the iteration.

The parity and initialization and simple sign reduction

– The main reduction is `ODD` $\wedge$ `EVEN` $\longrightarrow$ `BOT`

– The abstract values implementation is hidden, whence must be accessed through abstract primitives operations, such as:

- `(Avalues1.f_NAT "0")` = `EVEN`
- `(Avalues1.f_NAT "1")` = `ODD`
- `(Avalues2.f_NAT "0")` = `ZERO`
- `(Avalues2.f_NAT "2")` = `POS`
- ...

```
(* red-Parity-ISS12.ml *)
```

```
open Avalues1 (* avalues.ml of Parity *)
(* \gamma(BOT)  = {_O_(a)}                                     *)
(* \gamma(ODD)  = { 2n+1\in[min_int,max_int] | n\in Z } U {_O_(a)} *)
(* \gamma(EVEN) = { 2n\in[min_int,max_int] | n\in Z } U {_O_(a)}   *)
(* \gamma(TOP)  = [min_int,max_int] U {_O_(a),_O_(i)}          *)
open Avalues2 (* avalues.ml of initialization and simple sign *)
(* \gamma(BOT)  = {_O_(a)}                                   *)
(* \gamma(NEG)  = [min_int,-1] U {_O_(a)}                    *)
(* \gamma(POS)  = [1,max_int] U {_O_(a)}                     *)
(* \gamma(ZERO) = {0, _O_(a)}                                *)
(* \gamma(INI)  = [min_int,max_int] U {_O_(a)}               *)
(* \gamma(ERR)  = {_O_(a),_O_(i)}                            *)
(* \gamma(TOP)  = [min_int,max_int] U {_O_(a),_O_(i)}        *)
let reduce (p, i) =
if ((Avalues1.eq p (Avalues1.bot ()))
                              || (Avalues2.eq i (Avalues2.bot ())))
   then ((Avalues1.bot ()), (Avalues2.bot ()))
else if ((Avalues1.eq p (Avalues1.f_NAT "1"))
```



```
                              & (Avalues2.eq i (Avalues2.f_NAT "0")))
   then ((Avalues1.bot ()), (Avalues2.bot ()))
else if (Avalues2.eq i (Avalues2.f_NAT "0"))
   then ((Avalues1.f_NAT "0"), i)
else (p, i)
```


The parity and intervals reduction

```
(* red-Parity-Intervals12.ml *)
open Avalues1 (* avalues.ml of Parity *)
(* \gamma(BOT)  = {_O_(a)}                                     *)
(* \gamma(ODD)  = { 2n+1\in[min_int,max_int] | n\in Z } U {_O_(a)} *)
(* \gamma(EVEN) = { 2n\in[min_int,max_int] | n\in Z } U {_O_(a)}   *)
(* \gamma(TOP)  = [min_int,max_int] U {_O_(a),_O_(i)}          *)
open Avalues2 (* avalues.ml of Intervals *)
(* gamma (a,b) = [a,b] U {_O_(a), _O_(i)}                      *)
(*                                when min_int <= a <= b <= max_int *)
(*             = {_O_(a), _O_(i)}                              *)
(*                                when a = max_int > min_int = b    *)
(* reduction of parity and intervals *)
let reduce (p, i) =
if (Avalues1.eq p (Avalues1.bot ())) then
```



```
                                  ((Avalues1.bot ()), (Avalues2.bot ()))
else if (Avalues1.eq p (Avalues1.f_NAT "1")) then (p, (reduce_odd i))
else if (Avalues1.eq p (Avalues1.f_NAT "0")) then (p, (reduce_even i))
else if ((Avalues2.parity i) = 0) then ((Avalues1.f_NAT "0"), i)
else if ((Avalues2.parity i) = 1) then ((Avalues1.f_NAT "1"), i)
else (p, i)
```

In the interval abstract domain, the interval bounds can be reduced by the parity:

```
(* avalues.mli *)
...
(* reductions by parity *)
val reduce_even : t -> t
val reduce_odd  : t -> t
```

```
(* avalues.ml *)
(* reductions by parity *)
let reduce_even (a, b) = ((if ((a mod 2) = 0) then a
                           else if a = max_int then a else (a+1)),
                          (if ((b mod 2) = 0) then b
                           else if b = min_int then b else (b-1)))
let reduce_odd (a, b)  = ((if ((a mod 2) = 1) || ((a mod 2) = -1)
                           then a
                           else if a = max_int then a else (a+1)),
                          (if ((b mod 2) = 1) || ((b mod 2) = -1)
   then b
                           else if b = min_int then b else (b-1)))
```


In the parity abstract domain, the parity can be improved for constant intervals

```
(* reduction with parity *)
(* information on parity = 0:EVEN, 1:ODD, 2:TOP *)
let parity (a, b) = if (a = b) then (a mod 2) else 2
```

The parity information is sent in the form of a constant.

Notes on the naïve implementation

– In the reduction process, the information between modules is communicated in the form of constants.
– In general a more complex communication language, known by the two modules, is required to exchange the reduction information (e.g. in symbolic form)
– In the naïve implementation, the modules involved in the reduction are determined by using aliases of file names.
– In OCaml, modules can be parameterized, which would provide a more elegant solution

– Since the notation is positional (`Avalues1`, `Avalues2`, `...`) and module/abstract domain names cannot be changed, duplications of reductions may required (when a given reduction modules appear in different positions)
– This duplication, a simple macroexpansion, could have been automatized or better handled by the language module system

Reduction of intervals with initialization and simple sign

```
(* red-ISS-Intervals12.ml *)
open Avalues1 (* avalues.ml of Initialization-Simple-Sign *)
(* \gamma(BOT)  = {_O_(a)}                                     *)
(* \gamma(NEG)  = [min_int,-1] U {_O_(a)}                      *)
(* \gamma(POS)  = [1,max_int] U {_O_(a)}                       *)
(* \gamma(ZERO) = {0, _O_(a)}                                  *)
(* \gamma(INI)  = [min_int,max_int] U {_O_(a)}                 *)
(* \gamma(ERR)  = {_O_(a),_O_(i)}                              *)
(* \gamma(TOP)  = [min_int,max_int] U {_O_(a),_O_(i)}          *)
open Avalues2 (* avalues.ml of Intervals *)
(* gamma (a,b) = [a,b] U {_O_(a), _O_(i)}                      *)
(*                              when min_int <= a <= b <= max_int *)
(*              = {_O_(a), _O_(i)}                             *)
```



```
(*                              when a = max_int > min_int = b    *)
let gamma12 a =
   if (Avalues1.eq a (Avalues1.bot ()))
      then (Avalues2.bot ())
   else if (Avalues1.eq a (Avalues1.f_UMINUS (Avalues1.f_NAT "1")))
      then (Avalues2.neg ())
   else if (Avalues1.eq a (Avalues1.f_NAT "0"))
      then (Avalues2.f_NAT "0")
   else if (Avalues1.eq a (Avalues1.f_NAT "1"))
      then (Avalues2.pos ())
   else (Avalues2.top ())
let alpha21 i =
   if (Avalues2.eq i (Avalues2.bot ()))
      then (Avalues1.initerr ())
   else if ((sign i) = -1)
      then (Avalues1.f_UMINUS (Avalues1.f_NAT "1"))
   else if ((sign i) = 0)
      then (Avalues1.f_NAT "0")
```



```
   else if ((sign i) = 1)
      then (Avalues1.f_NAT "1")
   else (Avalues1.top ())
let reduce (a, b) = ((Avalues1.meet a (alpha21 b)),
                                    (Avalues2.meet b (gamma12 a)))
```

- Again the abstract values are communicated through abstract operations on constants
- The reduction is useful only in absence of thresholds in the widening (the initialization and simple sign amounts to restricting to the introduction of a 0 threshold)
- The reduction of intervals by initialization and simple sign uses primitives defined in the interval abstract domain:


```
(* avalues.mli *)
...
(* reduction with initialization and simple sign *)
(* information on simple sign = -1:<0, 0:=0, 1:>0, 2:TOP *)
val sign : t -> int
...

(* avalues.ml *)
...
(* reduction with initialization and simple sign *)
(* information on simple sign = -1:<0, 0:=0, 1:>0, 2:TOP *)
let sign (b1, b2) = (* b1 <= b2 *)
  if (b2 < 0) then -1
  else if (b1 = 0) & (b2 = 0) then 0
  else if (b1 > 0) then 1
  else 2
...
```

No error-intervals reduction

```
(* red-Errors-Intervals12.ml *)
open Avalues1 (* avalues.ml of Errors *)
(* gamma(NER) = [min_int,max_int]                    *)
(* gamma(AER) = [min_int,max_int] U {_O_(a)}         *)
(* gamma(IER) = [min_int,max_int] U {_O_(i)}         *)
(* gamma(ERR) = [min_int,max_int] U {_O_(a), _O_(i)} *)
open Avalues2 (* avalues.ml of Intervals *)
(* gamma (a,b) = [a,b] U {_O_(a), _O_(i)}            *)
(*                 when min_int <= a <= b <= max_int *)
(*             = {_O_(a), _O_(i)}                    *)
(*                 when a = max_int > min_int = b    *)
let reduce (a, b) = (a, b)
```

No reduction at all, which is always the case for independent information.

The ternary reduced product

```
(* avalues123.ml *)
open Avalues1
open Avalues2
open Avalues3
open Red123
(* reduced product *)
(*                 *)
(* ABSTRACT VALUES *)
(*                 *)
type t = Avalues1.t * Avalues2.t * Avalues3.t
(* gamma (a,b,c) =  Avalues1.gamma(a) /\ Avalues2.gamma(b) /\ *)
(*                                       Avalues3.gamma(c) *)
(* infimum: bot () = alpha({}) *)
let bot () = reduce ((Avalues1.bot ()), (Avalues2.bot ()),
                                         (Avalues3.bot ()))
```



```
(* isbotempty () = gamma(bot ()) = {} *)
let isbotempty () = (Avalues1.isbotempty ()) ||
            (Avalues2.isbotempty ()) || (Avalues3.isbotempty ())
(* uninitialization: initerr () = alpha({_O_i}) *)
let initerr () = reduce ((Avalues1.initerr ()), (Avalues2.initerr ()),
                                         (Avalues3.initerr ()))
(* supremum: top () = alpha({_O_i, _O_a} U [min_int,max_int]) *)
let top () =  reduce (Avalues1.top (), Avalues2.top (), Avalues3.top ())
(* least upper bound join: p q = alpha(gamma(p) U gamma(q)) *)
let join (v,w,t) (x,y,u) = reduce ((Avalues1.join v x),
                   (Avalues2.join w y), (Avalues3.join t u))
(* greatest lower bound meet p q = alpha(gamma(p) cap gamma(q)) *)
let meet (v,w,t) (x,y,u) = reduce ((Avalues1.meet v x),
                   (Avalues2.meet w y), (Avalues3.meet t u))
(* approximation ordering: leq p q = gamma(p) subseteq gamma(q) *)
let leq (v,w,t) (x,y, u) = (Avalues1.leq v x) & (Avalues2.leq w y)
                                              & (Avalues3.leq t u)
(* equality: eq p q = gamma(p) = gamma(q) *)
```



```
let eq (v,w,t) (x,y,u) = (Avalues1.eq v x) & (Avalues2.eq w y)
                                            & (Avalues3.eq t u)
(* included in errors?: in_errors p = gamma(p) subseteq {_O_i, _O_a} *)
let in_errors (x,y,z) = (Avalues1.in_errors x) ||
                  (Avalues2.in_errors y) || (Avalues3.in_errors z)
(* printing *)
let print (x,y,z) =
   (print_string "(";  Avalues1.print x;  print_string ",";
    Avalues2.print y; print_string ",";
    Avalues3.print z; print_string ")")
(*                      *)
(* ABSTRACT TRANSFORMERS *)
(*                      *)
(* forward abstract semantics of arithmetic expressions *)
(* f_NAT s = alpha({(machine_int_of_string s)})        *)
let f_NAT s = reduce (Avalues1.f_NAT s, Avalues2.f_NAT s,
                                         Avalues3.f_NAT s)
(* f_RANDOM () = alpha([min_int, max_int]) *)
```

```
let f_RANDOM () = reduce (Avalues1.f_RANDOM (),
                       Avalues2.f_RANDOM (), Avalues3.f_RANDOM ())
(* f_UMINUS a = alpha({ (machine_unary_minus x) | x \in gamma(a)} }) *)
let f_UMINUS (x, y, z) = reduce (Avalues1.f_UMINUS x,
                         Avalues2.f_UMINUS y, Avalues3.f_UMINUS z)
(* f_UPLUS a = alpha(gamma(a)) *)
let f_UPLUS (x, y, z) = reduce (x, y, z)
(* f_BINARITH a b = alpha({ (machine_binary_binarith i j) |     *)
(*                              i in gamma(a) /\ j \in gamma(b)} *)
let f_PLUS (a, b, c) (d, e, f) = reduce (Avalues1.f_PLUS a d,
                         Avalues2.f_PLUS b e, Avalues3.f_PLUS c f)
let f_MINUS (a, b, c) (d, e, f) = reduce (Avalues1.f_MINUS a d,
                       Avalues2.f_MINUS b e, Avalues3.f_MINUS c f)
let f_TIMES (a, b, c) (d, e, f) = reduce (Avalues1.f_TIMES a d,
                       Avalues2.f_TIMES b e, Avalues3.f_TIMES c f)
let f_DIV (a, b, c) (d, e, f) = reduce (Avalues1.f_DIV a d,
                           Avalues2.f_DIV b e, Avalues3.f_DIV c f)
let f_MOD (a, b, c) (d, e, f) = reduce (Avalues1.f_MOD a d,
```



```
                           Avalues2.f_MOD b e, Avalues3.f_MOD c f)
(* forward abstract semantics of boolean expressions              *)
(* Are there integer values in gamma(u) equal to values in gamma(v)? *)
(* f_LT p q = exists i in gamma(p) cap [min_int,max_int]:          *)
(*        exists j in gamma(q) cap [min_int,max_int]: machine_eq i j *)
let f_EQ (a, b, c) (d, e, f) = (Avalues1.f_EQ a d) &
                        (Avalues2.f_EQ b e) & (Avalues3.f_EQ  c f)
(* Are there integer values in gamma(u) strictly less than (<)  *)
(* integer values in gamma(v)?                                  *)
(* f_LT p q = exists i in gamma(p) cap [min_int,max_int]:       *)
(*   exists j in gamma(q) cap [min_int,max_int]: machine_lt i j *)
let f_LT (a, b, c) (d, e, f) = (Avalues1.f_LT a d) &
                         (Avalues2.f_LT b e) & (Avalues3.f_LT c f)
(* widening *)
let widen (a, b, c) (d, e, f) = reduce (Avalues1.widen a d,
                           Avalues2.widen b e, Avalues3.widen c f)
(* narrowing *)
let narrow (a, b, c) (d, e, f) = reduce (Avalues1.narrow a d,
```



```
                           Avalues2.narrow b e, Avalues3.narrow c f)
(* backward abstract semantics of arithmetic expressions        *)
(* b_NAT s v = (machine_int_of_string s) in gamma(v) cap        *)
(*                                       [min_int, max_int]? *)
let b_NAT s (a, b, c) = (Avalues1.b_NAT s a) &
                        (Avalues2.b_NAT s b) & (Avalues3.b_NAT s c)
(* b_RANDOM p = gamma(p) cap [min_int, max_int]  <> emptyset *)
let b_RANDOM (a, b, c) = (Avalues1.b_RANDOM a) &
                     (Avalues2.b_RANDOM b) & (Avalues3.b_RANDOM c)
(* b_UOP q p = alpha({i in gamma(q) |                           *)
(*                 UOP(i) \in gamma(p) cap [min_int, max_int]}) *)
let b_UMINUS (a, b, c) (d, e, f) = reduce (Avalues1.b_UMINUS a d,
                     Avalues2.b_UMINUS b e, Avalues3.b_UMINUS c f)
let b_UPLUS (a, b, c) (d, e, f) = reduce (Avalues1.b_UPLUS a d,
                       Avalues2.b_UPLUS b e, Avalues3.b_UPLUS c f)
(* b_BOP q1 q2 p = alpha2({<i1,i2> in gamma2(<q1,q2>) |         *)
(*            BOP(i1, i2) \in gamma(p) cap [min_int, max_int]}) *)
let b_PLUS (a, b, c) (d, e, f) (g, h, i) =
```



```
  let (a', d') = Avalues1.b_PLUS a d g in
   let (b', e') = Avalues2.b_PLUS b e h in
    let (c', f') = Avalues3.b_PLUS c f i in
       ((reduce (a', b', c')), (reduce (d', e', f')))
let b_MINUS (a, b, c) (d, e, f) (g, h, i) =
  let (a', d') = Avalues1.b_MINUS a d g in
   let (b', e') = Avalues2.b_MINUS b e h in
    let (c', f') = Avalues3.b_MINUS c f i in
      ((reduce (a', b', c')), (reduce (d', e', f')))
let b_TIMES (a, b, c) (d, e, f) (g, h, i) =
  let (a', d') = Avalues1.b_TIMES a d g in
   let (b', e') = Avalues2.b_TIMES b e h in
    let (c', f') = Avalues3.b_TIMES c f i in
      ((reduce (a', b', c')), (reduce (d', e', f')))
let b_DIV (a, b, c) (d, e, f) (g, h, i) =
  let (a', d') = Avalues1.b_DIV a d g in
   let (b', e') = Avalues2.b_DIV b e h in
    let (c', f') = Avalues3.b_DIV c f i in
```

```
      ((reduce (a', b', c')), (reduce (d', e', f')))
let b_MOD (a, b, c) (d, e, f) (g, h, i) =
  let (a', d') = Avalues1.b_MOD a d g in
   let (b', e') = Avalues2.b_MOD b e h in
    let (c', f') = Avalues3.b_MOD c f i in
      ((reduce (a', b', c')), (reduce (d', e', f')))
(* backward abstract interpretation of boolean expressions *)
(* a_EQ p1 p2 = let p = p1 cap p2 cap [min_int, max_int]I in <p, p> *)
let a_EQ p1 p2 = let p = meet p1 p2 in (p, p)
(* a_LT p1 p2 = alpha2({<i1, i2> |                            *)
(*                    i1 in gamma(p1) cap [min_int, max_int] /\ *)
(*                    i2 in gamma(p1) cap [min_int, max_int] /\ *)
(*                    i1 < i2})                                  *)
let a_LT (a, b, c) (d, e, f) =
  let (a', d') = Avalues1.a_LT a d in
   let (b', e') =  Avalues2.a_LT b e in
    let (c', f') =  Avalues3.a_LT c f in
     ((reduce (a', b', c')), (reduce (d', e', f')))
```


User manual of the generic abstract interpreter

- All abstract domains have the same interface
- The analyzer can be instanciated to a particular abstract domain by choosing which abstract domain to use
- This can be a basic domain, the reduction of 2 basic domains or the reduction of 3 basic domains
- Which abstract domains are used is chosen by aliasing to files implementing these domains
- the user manual is as follows:


```
Forward non-relational static analysis:
make help          : this help
(1) reset:
make reset         : erase all mode choices
(2) choose tracing mode:
make trace         : tracing all
make traceaexp     : tracing arithmetic expressions
make tracebexp     : tracing boolean expressions
make tracecom      : tracing commands
make tracered      : tracing ternary reductions
make notrace       : no tracing
(3) choose abstract interpreter mode:
(3a) relational/non-relational analysis:
make r             : relational abstract interpretor
make nr            : non-relational abstract interpretor
(3b) boolean expressions:
make fbool         : forward analysis
```



```
make fbbool        : forward/backward analysis
make fbrbool       : forward/backward reductive analysis
(3c) arithmetic expressions:
make fassign       : forward analysis
make fbassign      : forward/backward analysis
(4) choose static analysis and compile analyzer:
make err           : error analysis
make iss           : initialization and simple sign analysis
make int           : interval analysis
make par           : parity analysis
make err-int       : error x interval analysis
make iss-int       : initialization and simple sign x interval analysis
make par-int       : parity x interval analysis
make par-iss       : parity x initialization and simple sign analysis
make par-iss-int : parity x initialization and simple sign analysis x
                     interval
(5) analyze:
./a.out            : analyze (the standard input)
```

```
./a.out file.sil : analyze (the file "file.sil")
make examples    : analyze all examples
(6) clean:
make clean       : remove auxiliary files
```


Example reduced product of parity, initialization and simple sign and intervals

– Basic static analyses:
  - “Parity” static analysis:

```
% make reset
Remove instanciated files
% make notrace
Tracing mode off
% make nr
"Non-relational" static analysis
% make fbrbool
Forward/backward analysis of boolean expressions with reduction
% make fbassign
Forward/backward analysis of assignments
```



```
% make par
...
"Parity" static analysis
% ./a.out ../Examples/example09.sil
{ x:T; y:T; z:T; t:T }
0:
  x := (-536870912 * 2);
1:
  y := (536870912 * 2);
2:
  z := ((-1073741823 - 1) * 1);
3:
  t := ((-1073741823 - 1) * 1073741823)
4:
{ x:e; y:e; z:e; t:e }
%
```


Note that in the assignment to `y`, `(536870912 * 2) = 1073741824` $>$ `max_int` $=$ `1073741823`. So execution is stopped which is overapproximated by `y:e`.

- “Initialization and simple sign” static analysis:

```
{ x:ERR; y:ERR; z:ERR; t:ERR }
0:
  x := (-536870912 * 2);
1:
  y := (536870912 * 2);
2:
  z := ((-1073741823 - 1) * 1);
3:
  t := ((-1073741823 - 1) * 1073741823)
4:
{ x:NEG; y:POS; z:NEG; t:NEG }
```

- “Interval” static analysis:

```
{ x:[]; y:[]; z:[]; t:[] }
0:
  x := (-536870912 * 2);
1:
  y := (536870912 * 2);
2:
  z := ((-1073741823 - 1) * 1);
3:
  t := ((-1073741823 - 1) * 1073741823)
4:
{ x:[]; y:[]; z:[]; t:[min_int,min_int] }
```


- Binary/pairwise reductions:
  - Reduced product of “Parity” and “Initialization and simple sign” static analysis:

```
{ x:(T, ERR); y:(T, ERR); z:(T, ERR); t:(T, ERR) }
0:
  x := (-536870912 * 2);
1:
  y := (536870912 * 2);
2:
  z := ((-1073741823 - 1) * 1);
3:
  t := ((-1073741823 - 1) * 1073741823)
4:
{ x:(e, NEG); y:(e, POS); z:(e, NEG); t:(e, NEG) }
```


- Reduced product of “Parity” and “Interval” static analysis:

```
{ x:(T, []); y:(T, []); z:(T, []); t:(T, []) }
0:
  x := (-536870912 * 2);
1:
  y := (536870912 * 2);
2:
  z := ((-1073741823 - 1) * 1);
3:
  t := ((-1073741823 - 1) * 1073741823)
4:
{ x:(_|_, []); y:(_|_, []); z:(_|_, []); t:(e, [min_int,min_int]) }
```


- Reduced product of “Initialization and simple sign” and “Interval” static analysis:

```
{ x:(ERR, []); y:(ERR, []); z:(ERR, []); t:(ERR, []) }
0:
  x := (-536870912 * 2);
1:
  y := (536870912 * 2);
2:
  z := ((-1073741823 - 1) * 1);
3:
  t := ((-1073741823 - 1) * 1073741823)
4:
{ x:(BOT, []); y:(BOT, []); z:(BOT, []); t:(NEG, [min_int,min_int]) }
```

- Binary/pairwise reductions (reduced product of “Parity”, “Initialization and simple sign” and “Interval” static analysis):

```
{ x:(T,ERR,[]); y:(T,ERR,[]); z:(T,ERR,[]); t:(T,ERR,[]) }
0:
  x := (-536870912 * 2);
1:
  y := (536870912 * 2);
2:
  z := ((-1073741823 - 1) * 1);
3:
  t := ((-1073741823 - 1) * 1073741823)
4:
{ x:(_|_,BOT,[]); y:(_|_,BOT,[]); z:(_|_,BOT,[]);
t:(e,NEG,[min_int,min_int]) }
```


Reduction can cancel convergence enforcement by widening/narrowing

- With abstract domains not satisfying the ACC, the reduction can destroy the effect of the widenings in each of the abstract domains
- A post-reduction widening may have to be included
- If reduction is costly, it may be applied less often (e.g. only once in a loop)

Linearization

The linear abstraction

Many relational abstraction of sets of vectors of $\mathbb{I}$ ($\mathbb{I} = \mathbb{Z}, \mathbb{Q}$ or $\mathbb{R}$) involve linear expressions on the form

$$: a_{j0} x_0 + \cdots + a_{jn} x_n, \quad j = 1, \ldots, k$$

which can be encoded as a matrix product

$$\begin{pmatrix} a_{00} & \cdots & a_{0n} \\ \cdots & a_{ji} & \cdots \\ a_{k0} & & a_{kn} \end{pmatrix} \begin{pmatrix} x_0 \\ \vdots \\ x_n \end{pmatrix}$$

written AX where X is the column vector $(x_0, \ldots, x_n)$.

The affine abstraction

The affine case, involve expressions of the form

$a_{j0} x_0 + \dots + a_{jn} x_n + a_{jn+1}$, $j = 1, \dots k$

in matrix form $AX + B$ this is:

$$\begin{pmatrix} a_{00} & \cdots & \cdots & a_{0m} \\ \cdots & a_{ji} & \cdots & \\ a_{k0} & \cdots & \cdots & a_{km} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} a_{0m+1} \\ \vdots \\ a_{kn+1} \end{pmatrix}$$

Linear/affine expressions recognition

– Such linear or affine abstractions can only handled linear or affine expressions, the others being assimilated to a random assignment

– There are essentially two ways of recognizing linear/affine expressions:

- static (before the analysis), or
- dynamic (during the analysis)

Static linear/affine expressions recognition

– In the static view, the expressions which are recognized as linear/affine are those which directly appear in the program such ~~as~~ :

$2 * X + 1$

This will miss

$T := 2;$
$T * X + 1$

Dynamic linear/affine expressions recognition

– In the dynamic view (as used in ASTRÉE), the expression is partially evaluated using information presently available to transform it in linear form. If constant propagation is used above, then $T = 2$ in $T * X + 1$ yields the affine expression $2 * X + 1$. If on the other hand, $X = 3$ then we get $T * 3 + 1$ whereas the static view would yield the random assignment ?.

The syntax of linear arithmetic expressions

– The linear arithmetic expressions of SIL program P with finitely many variables $\text{Var}[\![P]\!] = \{\mathtt{X}_1, \ldots, \mathtt{X}_k\}$, $k \in \mathbb{N}$ are defined as

$$
\begin{array}{rl}
L ::= & ? \\
\mid & \sum_{i=1}^{k} n_i \times \mathtt{X}_i + n_{k+1}
\end{array}
$$

where the n_j, $j = 1, \ldots, k+1$ are numbers.

– This can be easily encoded as the vector $\langle n_1, \ldots, n_k, n_{k+1} \rangle$

The forward collecting semantics of linear arithmetic expressions

– The collecting semantics of linear arithmetic expressions is defined as:

$$
\boxed{
\begin{array}{c}
\text{Faexp}[\![?]\!]R \stackrel{\text{def}}{=} \mathbb{I} \\
\text{Faexp}[\![\sum_{i=1}^{k} n_i \times \mathtt{X}_i + n_{k+1}]\!]R \stackrel{\text{def}}{=} \{\sum_{i=1}^{k} n_i \times \rho(\mathtt{X}_i) + n_{k+1} \mid \rho \in R\}
\end{array}
}
$$

The linear abstraction of arithmetic expressions

We define a syntactic abstraction:

$$\alpha : A \to L$$

such that for all arithmetic expressions:

$$\text{Faexp}[\![A]\!] \stackrel{.}{\subseteq} \text{Faexp}[\![\alpha(A)]\!]$$

so that $\text{Faexp}[\![\alpha(A)]\!]$ is an upper-approximation of $\text{Faexp}[\![A]\!]$. This proves the soundness of the linearization α.

$$\alpha(n) = \sum_{i=1}^{n} 0 * X_i + n$$

$$\alpha(X_j) = \sum_{\substack{i=1 \\ i \neq j}}^{k} 0 * X_i + 1 * X_j + 0$$

$$\alpha(?) = ?$$

$\alpha(A_1 + A_2) =$

$\underline{\text{if}}\ \alpha(A_1) = ?\ \text{then}\ ?$

$\underline{\text{elsif}}\ \alpha(A_2) = ?\ \text{then}\ ?$

$\underline{\text{else}}\ \underline{\text{let}}\ \alpha(A_1) = \sum_{i=1}^{k} n_i^1 * X_i + n_{k+1}^1$

$\quad \underline{\text{and}}\ \alpha(A_2) = \sum_{i=1}^{k} n_i^2 * X_i + n_{k+1}^2$

$\underline{\text{in}}$

$$\sum_{i=1}^{k} (n_i^1 \oplus n_i^2) * X_i + (n_{k+1}^1 \oplus n_{k+1}^2)$$

(where $n \oplus m$ is ℓ such that $\underline{\ell} = \underline{n} + \underline{m}$ and in case of overflow the result is $?$).

$\alpha(-A) = \underline{\text{if}}\ \alpha(A) = ?\ \text{then}\ ?$

$\quad \underline{\text{else}}\ \underline{\text{let}}\ \alpha(A) = \sum_{i=1}^{k} n_i * X_i + n_{k+1}\ \underline{\text{in}}$

$$\sum_{i=1}^{k} (\ominus n_i) * X_i + \ominus n_{k+1}$$

(where $\ominus n$ is m such that $\underline{m} = -\underline{n}$. Morei
if $-\underline{n}$ overflows then the result is ?).

$\alpha(A_1 * A_2) =$

<u>if</u> $\alpha(A_1) = ?$ <u>or</u> $\alpha(A_2) = ?$ <u>then</u> ? <u>else</u>

<u>let</u> $\alpha(A_1) = \sum_{i=1}^{k} n_i^1 * X_i + n_{k+1}^1$

<u>and</u> $\alpha(A_2) = \sum_{i=1}^{k} n_i^2 * X_i + n_{k+1}^2$

<u>if</u> $\bigwedge_{i=1}^{k} n_i^1 = 0$ <u>then</u>

$$\sum_{i=1}^{k} n_{k+1}^1 \circledast n_i^2 * X_i + n_{k+1}^1 \circledast n_{k+1}^2$$

<u>elsif</u> $\bigwedge_{i=1}^{k} n_i^2 = 0$ <u>then</u>

$$\sum_{i=1}^{k} n_i^1 \circledast n_{k+1}^2 * X_i + n_{k+1}^1 \circledast n_{k+1}^2$$

<u>else</u> ?

(where $n \circledast m$ is ℓ such that $\underline{\ell} = \underline{n} \times \underline{m}$ and the result is ? when one of these products overflows).

Observe that the product of linear/affine arithmetic expressions is not, in general, linear/affine, so we consider the particular case of one constant expression only. The other operators are also abstracted to ?

Definition of the forward collecting semantics of arithmetic expressions

Recall the *forward/bottom-up collecting semantics* of an arithmetic expression from lecture 8:

$$\text{Faexp} \in \text{Aexp} \mapsto \wp(\text{Env}[\![P]\!]) \xrightarrow{\sqcup} \wp(\mathbb{I}_\Omega),$$

$$\text{Faexp}[\![A]\!]R \stackrel{\text{def}}{=} \{v \mid \exists \rho \in R : \rho \vdash A \Rrightarrow v\} . \qquad (1)$$

such that:

$$\text{Faexp}[\![A]\!]\Big(\bigcup_{k\in S} R_k\Big) = \bigcup_{k\in S} (\text{Faexp}[\![A]\!]R_k)$$

$$\text{Faexp}[\![A]\!]\emptyset = \emptyset .$$

Structural specification of the forward collecting semantics of arithmetic expressions

$$\text{Faexp}[\![\mathtt{n}]\!]R \stackrel{\text{def}}{=} \{\underline{\mathtt{n}}\}$$ [6]

$$\text{Faexp}[\![\mathtt{X}]\!]R \stackrel{\text{def}}{=} R(\mathtt{X})$$

where $R(\mathtt{X}) \stackrel{\text{def}}{=} \{\rho(\mathtt{X}) \mid \rho \in R\}$

$$\text{Faexp}[\![?]\!]R \stackrel{\text{def}}{=} \mathbb{I}$$

$$\text{Faexp}[\![\mathtt{u}\ A']\!]R \stackrel{\text{def}}{=} \underline{\mathtt{u}}^C(\text{Faexp}[\![A']\!]R)$$

where $\underline{\mathtt{u}}^C(V) \stackrel{\text{def}}{=} \{\underline{\mathtt{u}}(v) \mid v \in V\}$

$$\text{Faexp}[\![A_1\ \mathtt{b}\ A_2]\!]R \stackrel{\text{def}}{=} \underline{\mathtt{b}}^C(\text{Faexp}[\![A_1]\!], \text{Faexp}[\![A_2]\!])R$$

where $\underline{\mathtt{b}}^C(F_1, F_2)R \stackrel{\text{def}}{=} \{v_1\ \underline{\mathtt{b}}\ v_2 \mid \exists \rho \in R : v_1 \in F_1(\{\rho\}) \wedge v_2 \in F_2(\{\rho\})\}$

[6] For short, the case $\text{Faexp}[\![A]\!]\emptyset = \emptyset$ is not recalled.

Correctness of the linear abstraction

THEOREM. The syntactic transformation of an arithmetic expression A of a program P into its linear form $\alpha(A)$ yields an upper approximation of its forward collecting semantics

$$\forall R \in \wp(\mathrm{Env}[\![P]\!]) \setminus \{\emptyset\} : \mathrm{Faexp}[\![A]\!]R \subseteq \mathrm{Faexp}[\![\alpha(A)]\!]R$$

■

Note: since the analysis is defined by structural induction on the program syntax, a program transformation can be understood as an abstraction of the syntactic parameter of the analyzer: $\alpha(\mathrm{Faexp}[\![A]\!]) \stackrel{\text{def}}{=} \mathrm{Faexp}[\![\alpha(A)]\!]$

Proof of correctness of the linear abstraction

PROOF. By structural induction on A

- If A is n then:

$$\mathrm{Faexp}[\![\alpha(n)]\!]R$$
$$= \mathrm{Faexp}[\![\sum_{i=1}^{k} 0 * X_i + n]\!]R$$
$$= \{\sum_{i=1}^{k} 0 \times \rho(X_i) + n \mid \rho \in R\}$$
$$= \{n \mid \rho \in R\}$$
$$= \{n\} \quad \text{since } R \neq \emptyset$$
$$\mathrm{Faexp}[\![n]\!]R.$$

- If A is X_j then:

$$\mathrm{Faexp}[\![\alpha(X_j)]\!]R$$
$$= \mathrm{Faexp}[\![\sum_{\substack{i=1\\ i\neq j}}^{k} 0 * X_i + 1 * X_j + 0]\!]R$$
$$= \{\sum_{\substack{i=1\\ i\neq j}}^{k} 0 \times \rho(X_i) + 1 \times \rho(X_j) + 0 \mid \rho \in R\}$$
$$= \{\rho(X_j) \mid \rho \in R\} = R(X_j) = \mathrm{Faexp}[\![X_j]\!]R$$

- If A is $?$ then

$$\mathrm{Faexp}[\![\alpha(?)]\!]R = \mathbb{I}_{\Omega} \supseteq \mathbb{I} = \mathrm{Faexp}[\![?]\!]R^{(*)}$$

(*) Recall that $\mathrm{Faexp}[\![A]\!]R \in \mathbb{I}_{\Omega}$ since the evaluation of A might yield Ω_i or Ω_a, so that this possibility must be included in the abstraction $\alpha(A)$. Notice that to be more precise we could have distinguished $?$ and $?_{\Omega}$ in the abstract.

- If A is $(A_1 + A_2)$ then we proceed by cases on $\alpha(A_1)$ and $\alpha(A_2)$. If any one of them is $?$ then obviously:

$$
\begin{aligned}
& \mathrm{Caexp}[\![A]\!]R \\
\subseteq\ & \mathbb{I}_\Omega \\
=\ & \mathrm{Caexp}[\![?]\!]R \\
=\ & \mathrm{Caexp}[\![\alpha(A)]\!].
\end{aligned}
$$

Otherwise we have:

$$\alpha(A_1) = \sum_{i=1}^{k} n_i^1 . X_i + n_{k+1}^1$$

$$\alpha(A_2) = \sum_{i=1}^{k} n_i^2 . X_i + n_{k+1}^2$$

so that we have:

$$\mathrm{Caexp}[\![\alpha(A_1 + A_2)]\!]R$$

$$= \mathrm{Caexp}[\![\sum_{i=1}^{k} (n_i^1 \oplus n_i^2) * X_i + (n_{k+1}^1 \oplus n_{k+1}^2)]\!]R$$

By structural induction hypothesis, we have:

$$\mathrm{Caexp}[\![A_1]\!](\{\rho\}) \subseteq \mathrm{Caexp}[\![\alpha(A_1)]\!](\{\rho\})$$

$$= \mathrm{Caexp}[\![\sum_{i=1}^{k} n_i^1 \times \rho(X_i) + n_{k+1}^1]\!](\{\rho\})$$

and so:

$$\supseteq \{\mathrm{Caexp}[\![A_1]\!](\{\rho\}) + \mathrm{Caexp}[\![A_2]\!](\{\rho\}) \mid \rho \in R\}$$

$$= \mathrm{Caexp}[\![A_1 + A_2]\!]R$$

- For the product, the proof is similar.

□

- If some of the $n_i^1 \oplus n_i^2$ overflows, then we would have had $?$, a case already considered above.
- Otherwise,

$$= \{\sum_{i=1}^{k} (n_i^1 + n_i^2) \times \rho(X_i) + (n_{k+1}^1 + n_{k+1}^2) \mid \rho \in R\}$$

$$= \{(\sum_{i=1}^{k} n_i^1 \times \rho(X_i) + n_{k+1}^1) + (\sum_{i=1}^{k} n_i^2 \times \rho(X_i) + n_{k+1}^2) \mid \rho \in R\}$$

$$= \{\mathrm{Caexp}[\![\alpha(A_1)]\!](\{\rho\}) + \mathrm{Caexp}[\![\alpha(A_2)]\!](\{\rho\}) \mid \rho \in R\}$$

Syntax of linear boolean expressions

We define linear boolean expressions as::colorblue

$$
\begin{aligned}
BL ::=\ & BL_1 \mid BL_2 \\
\mid\ & BL_1 \,\&\, BL_2 \\
\mid\ & AL_1 = AL_2 \\
\mid\ & AL_1 < AL_2 \\
\mid\ & \mathtt{true} \\
\mid\ & \mathtt{false}
\end{aligned}
$$

The collecting semantics is essentially unchanged but for the use of linear arithmetic expressions.

The collecting semantics of linear boolean expressions

$$
\begin{aligned}
\mathrm{Cbexp}[\![\mathtt{true}]\!]R &\stackrel{\text{def}}{=} R\\
\mathrm{Cbexp}[\![\mathtt{false}]\!]R &\stackrel{\text{def}}{=} \emptyset\\
\mathrm{Cbexp}[\![AL_1 \;\mathsf{c}\; AL_2]\!] &\stackrel{\text{def}}{=} \underline{\mathsf{c}}^{\mathcal{C}}\,(\mathrm{Faexp}[\![AL_1]\!], \mathrm{Faexp}[\![AL_2]\!])R
\end{aligned}
$$

where $\underline{\mathsf{c}}^{\mathcal{C}}\,(F, G)R \stackrel{\text{def}}{=} \{\rho \in R \mid \exists v_1 \in F(\{\rho\}) \cap \mathbb{I} : \exists v_2 \in G(\{\rho\}) \cap \mathbb{I} : v_1 \;\underline{\mathsf{c}}\; v_2 = \mathtt{tt}\}$

$$
\begin{aligned}
\mathrm{Cbexp}[\![BL_1 \;\&\; BL_2]\!]R &\stackrel{\text{def}}{=} \mathrm{Cbexp}[\![BL_1]\!]R \cap \mathrm{Cbexp}[\![BL_2]\!]R\\
\mathrm{Cbexp}[\![BL_1 \mid BL_2]\!]R &\stackrel{\text{def}}{=} \mathrm{Cbexp}[\![BL_1]\!]R \cup \mathrm{Cbexp}[\![BL_2]\!]R
\end{aligned}
$$

Linearization of boolean expressions

The extension of linearization to boolean expressions is trivial since it essentially concerns the arithmetic expressions within the boolean expression:

$$
\begin{aligned}
\alpha(\mathtt{true}) &\stackrel{\text{def}}{=} \mathtt{true}\\
\alpha(\mathtt{false}) &\stackrel{\text{def}}{=} \mathtt{false}\\
\alpha(A_1 \;\mathsf{c}\; A_2) &\stackrel{\text{def}}{=} \alpha(A_1) \;\mathsf{c}\; \alpha(A_2)\\
\alpha(B_1 \;\&\; B_2) &\stackrel{\text{def}}{=} \alpha(B_1) \;\&\; \alpha(B_2)\\
\alpha(B_1 \mid B_2) &\stackrel{\text{def}}{=} \alpha(B_1) \mid \alpha(B_2)
\end{aligned}
$$

Soundness of the linearization of boolean expressions

THEOREM. The syntactic transformation of a boolean expression B of a program P into its linear form $\alpha(B)$ yields an upper approximation of its forward collecting semantics

$$\forall R \in \wp(\mathrm{Env}[\![P]\!]) \setminus \{\emptyset\} : \mathrm{Cbexp}[\![B]\!]R \subseteq \mathrm{Cbexp}[\![\alpha(B)]\!]R$$

■

Note: again the semantic abstraction $\alpha \in (\wp(\mathrm{Env}[\![P]\!]) \xrightarrow{\sqcup} \wp(\mathrm{Env}[\![P]\!])) \mapsto (\wp(\mathrm{Env}[\![P]\!]) \xrightarrow{\sqcup} \wp(\mathrm{Env}[\![P]\!]))$ is defined in term of a syntactic transformation of boolean expressions into linear boolean expressions: $\alpha(\mathrm{Cbexp}[\![B]\!]) \stackrel{\text{def}}{=} \mathrm{Cbexp}[\![\alpha(B)]\!]$

PROOF. We proceed by structural induction on B.

– trivial for true and false

– for the arithmetic comparison, we have shown that

$\forall R \in \wp(R) : \mathrm{Caexp}[\![A]\!]R \subseteq \mathrm{Caexp}[\![\alpha(A)]\!]R$

and so when $R \neq \emptyset$:

$$
\begin{aligned}
&\mathrm{Cbexp}[\![\alpha(A_1 \;\mathsf{c}\; A_2)]\!]R\\
= \;&\mathrm{Cbexp}[\![\alpha(A_1) \;\mathsf{c}\; \alpha(A_2)]\!]R\\
= \;&\underline{\mathsf{c}}^{\mathcal{C}}(\mathrm{Caexp}[\![\alpha(A_1)]\!], \mathrm{Caexp}[\![\alpha(A_2)]\!])\\
\supseteq \;&\{\underline{\mathsf{c}}^{\mathcal{C}} \text{ is pointwise extensive if } F \mathrel{\dot{\subseteq}} F' \text{ and } G \subseteq G' \text{ then } \underline{\mathsf{c}}^{\mathcal{C}}(F, G) \subseteq \underline{\mathsf{c}}^{\mathcal{C}}(F', G')\}\\
&\underline{\mathsf{c}}^{\mathcal{C}}(\mathrm{Caexp}[\![A_1]\!], \mathrm{Caexp}[\![A_2]\!])\\
= \;&\mathrm{Cbexp}[\![A_1 \;\mathsf{c}\; A_2]\!].
\end{aligned}
$$

- for the boolean operator, we have

$$
\begin{aligned}
& \text{Cbexp}[\![\alpha(B_1 \ \& \ B_2)]\!] R \\
= \ & \text{Cbexp}[\![\alpha(B_1) \ \& \ \alpha(B_2)]\!] R \\
= \ & \text{Cbexp}[\![\alpha(B_1)]\!] R \ \cap \ \text{Cbexp}[\![\alpha(B_2)]\!] R \\
\subseteq \ & \{\text{by induction hypothesis}\} \\
& \text{Cbex}[\![B_1]\!] R \ \cap \ \text{Cbexp}[\![B_2]\!] R \\
= \ & \text{Cbexp}[\![B_1 \ \& \ B_2]\!] R
\end{aligned}
$$

- The case of disjunction | is similar.

$\square$

Syntax of linear commands and programs

LC ::=		Linear commands
	skip	void
|	X := AL	linear assignment
|	if BL then LCL_0 else LCL_1 fi	test
|	while BL do LCL_0 od	iteration
LCL ::=		List of linear commands
	CL	
|	LC ; LCL_0	
PL ::=		Linear programs
	LCL;;	

The postcondition collecting semantics of linear programs

$$
\begin{aligned}
\text{Pcom}[\![\texttt{skip}]\!]R &= R \\
\text{Pcom}[\![\texttt{X} := AL]\!]R &= \{\rho[\texttt{X} := i] \mid \rho \in R \wedge i \in (\text{Faexp}[\![AL]\!]\{\rho\}) \cap \mathbb{I}\} \\
\text{Pcom}[\![\texttt{if } BL \texttt{ then } LCL_t \texttt{ else } LCL_f \texttt{ fi}]\!]R &= \\
& \text{Pcom}[\![LCL_t]\!](\text{Cbexp}[\![BL]\!]R) \cup \text{Pcom}[\![LCL_f]\!](\text{Cbexp}[\![T(\neg(BL))]\!]R) \\
\text{Pcom}[\![\texttt{while } BL \texttt{ do } LCL \texttt{ od}]\!]R &= \\
& \text{let } I = \mathsf{lfp}_{\emptyset}^{\subseteq} \lambda X \cdot R \cup \text{Pcom}[\![LCL]\!](\text{Cbexp}[\![BL]\!]X) \text{ in} \\
& \quad \text{Cbexp}[\![T(\neg(BL))]\!]I) \\
\text{Pcom}[\![BL \ ; \ LCL_0]\!]R &= (\text{Pcom}[\![LCL_0]\!] \circ \text{Pcom}[\![BL]\!])R \\
\text{Pcom}[\![LCL_0 \ ;;]\!]R &= \text{Pcom}[\![LCL_0]\!]
\end{aligned}
$$

Program linearization

The linearization abstraction is trivially extended to commands and programs as follows:

$$
\begin{aligned}
\alpha(\texttt{skip}) &\stackrel{\text{def}}{=} \texttt{skip} \\
\alpha(\texttt{X} := AL) &\stackrel{\text{def}}{=} \texttt{X} := \alpha(AL) \\
\alpha(\texttt{if } B \texttt{ then } LC_t \texttt{ else } LC_f \texttt{ fi}) &\stackrel{\text{def}}{=} \\
& \texttt{if } \alpha(B) \texttt{ then } \alpha(LC_t) \texttt{ else } \alpha(LC_f) \texttt{ fi} \\
\alpha(\texttt{while } B \texttt{ do } LC \texttt{ od}) &\stackrel{\text{def}}{=} \texttt{while } \alpha(B) \texttt{ do } \alpha(LC) \texttt{ od} \\
\alpha(B \ ; \ LC_0) &\stackrel{\text{def}}{=} \alpha(B) \ ; \ \alpha(LC_0) \\
\alpha(LC_0 \ ;;) &\stackrel{\text{def}}{=} \alpha(LC_0) \ ;;
\end{aligned}
$$

Soundness of program linearization

THEOREM. The syntactic transformation of a program P into its linear form $\alpha(P)$ yields an upper approximation of its postcondition collecting semantics

$$\forall R \in \wp(\mathrm{Env}[\![P]\!]) \setminus \{\emptyset\} : \mathrm{Pcom}[\![P]\!]R \subseteq \mathrm{Pcom}[\![\alpha(P)]\!]R$$

■

Note: again we leave implicit the fact that this is indeed a semantic abstraction defined as: $\alpha(\mathrm{Pcom}[\![P]\!]) \stackrel{\mathrm{def}}{=} \mathrm{Pcom}[\![\alpha(P)]\!]$

PROOF. We proceed by structural induction on P.

– trivial for <u>skip</u>.

– $\mathrm{Pcom}[\![X := \alpha(A)]\!]R$

$= \{\rho[X \leftarrow i] \mid \rho \in R \wedge i \in \mathrm{Caexp}[\![\alpha(A)]\!](\{\rho\}) \cap \mathbb{I}\}$

$\supseteq \{\rho[X \leftarrow i] \mid \rho \in R \wedge i \in \mathrm{Caexp}[\![A]\!](\{\rho\}) \cap \mathbb{I}\}$

$= \mathrm{Pcom}[\![X := A]\!]R$

– $\mathrm{Pcom}[\![\alpha(\underline{\mathtt{if}}\ B\ \underline{\mathtt{then}}\ LC_1\ \underline{\mathtt{else}}\ LC_2\ \underline{\mathtt{fi}})]\!]R$

$= \mathrm{Pcom}[\![\underline{\mathtt{if}}\ \alpha(B)\ \underline{\mathtt{then}}\ \alpha(LC_1)\ \underline{\mathtt{else}}\ \alpha(LC_2)\ \underline{\mathtt{fi}}]\!]R$

$= \mathrm{Pcom}[\![\alpha(LC_1)]\!](\mathrm{Cbexp}[\![\alpha(B)]\!]R) \cup$
$\quad \mathrm{Pcom}[\![\alpha(LC_2)]\!](\mathrm{Cbexp}[\![\alpha(T(\neg B))]\!]R$

$\supseteq \{\mathrm{Cbexp}[\![B]\!] \stackrel{.}{\subseteq} \mathrm{Cbexp}[\![\alpha(B)]\!],\ \alpha(T(\neg B)) = T(\alpha(\neg B))$ and $\mathrm{Pcom}[\![LC]\!]$ is monotone$\}$

$\mathrm{Pcom}[\![\alpha(LC_1)]\!](\mathrm{Cbexp}[\![B]\!]R) \cup$
$\mathrm{Pcom}[\![\alpha(LC_2)]\!](\mathrm{Cbexp}[\![T(\neg B)]\!]R)$

$\supseteq \{\mathrm{Pcom}[\![\alpha(LC)]\!]R \supseteq \mathrm{Pcom}[\![LC]\!]R\}$
$\mathrm{Pcom}[\![LC_1]\!](\mathrm{Cbexp}[\![B]\!]R) \cup$
$\mathrm{Pcom}[\![LC_2]\!](\mathrm{Cbexp}[\![T(\neg B)]\!]R)$

$= \mathrm{Pcom}[\![\underline{\mathtt{if}}\ B\ \underline{\mathtt{then}}\ LC_1\ \underline{\mathtt{else}}\ LC_2\ \underline{\mathtt{fi}}]\!]R$

– The proof for while is similar using the fact that $F \stackrel{.}{\sqsubseteq} G$ implies $\mathrm{lfp}_{\perp}^{\sqsubseteq} F \sqsubseteq \mathrm{lfp}_{\perp}^{\sqsubseteq} G$.

– The case $LC\ ;;$ is trivial.

□

Implementation of the syntactic linear abstraction

Linear relational representation of programs

The linear representation of

$$a_0 x_0 + \ldots + a_n x_n + a_{n+1}$$

is the vector

$$[a_0 ; a_1 ; \ldots ; a_n ; a_{n+1}]$$

while the representation of

$$a_0 x_0 + \cdots + a_n x_n \geqslant a_{n+1}$$

is:

$$[a_0 ; a_1 ; \cdots ; a_n ; a_{n+1}].$$

Non-linear expressions or tests are represented by a random assignment.

The non-initialization (Ω_{i}) and arithmetic errors (Ω_{a}) values of variables are simply ignored.

For the forthcoming backward analyzes, we need to know the label `after c` of a command `c` as well as a check `incom` ℓ `c` to test that the label ℓ does appear within command `c`.

```
(* linear_Syntax.mli *)
open Abstract_Syntax
(* A linear arithmetic expression a1.x1+...+an.xn+b, where n is the *)
(* number of program variables, is represented by a vector:         *)
(* LINEAR_AEXP a1 ... an b. A non-linear arithmetic expression is   *)
(* represented by RANDOM_AEXP.                                      *)
type laexp =
  | RANDOM_AEXP                (* random expression                 *)
  | LINEAR_AEXP of int array (* linear expression                 *)
```



```
and lbexp =
  | LTRUE | LFALSE     (* constant boolean expression                  *)
  | RANDOM_BEXP        (* random boolean expression                    *)
  | LAND of lbexp list (* boolean conjunction                          *)
  | LOR  of lbexp list (* boolean disjunction                          *)
  | LGE  of int array  (* LGE a1 ... an b is a1.x1+...+an.xn >= b      *)
  | LEQ  of int array  (* LGE a1 ... an b is a1.x1+...+an.xn = b       *)
and label = Abstract_Syntax.label
and lcom =
  | LSKIP of label * label
  | LASSIGN of label * variable * laexp * label
  | LSEQ of label * (lcom list) * label
  | LIF of label * lbexp * lbexp * lcom * lcom * label
  | LWHILE of label * lbexp * lbexp * lcom * label
val after : lcom -> label          (* command exit label                *)
val incom : label -> lcom -> bool (* label in command                  *)
```


The linear abstraction of programs

– The *linear abstraction* of programs consists in replacing all non-liear expressions by a random choice (which is safe). Moreover the linear expressions are transformed into the array form defined in `linear_Syntax.mli`.

```
(* abstract_To_Linear_Syntax.mli *)
open Abstract_Syntax
open Linear_Syntax
(* Linearization of commands *)
val linearize_com : com -> lcom
```

– The linearrization is by induction on the syntax of expressions by combination of the lineat forms of the subexpressions. For example:
  - For a constant v:

$$0.x_0 + 0.x_1 + \ldots + 0.x_n + v$$

  - For a variable x_i

$$0.x_0 + 0.x_1 + \ldots + 1.x_i + \ldots + 0.x_n + 0$$

  - For an addition:

$$\begin{aligned} & (a_0x_0 + \ldots a_nx_n + a_{n+1}) + (b_0x_0 + \ldots b_nx_n + b_{n+1}) \\ = & (a_0 + b_0)x_0 + \ldots (a_n + b_n)x_n + (a_{n+1} + b_{n+1}) \end{aligned}$$

    When a coefficient is not machine representable, the result is simply the random overapproximation (represented by the random assignment).
  - Finally, when the combination is not linear (e.g. product by non-constant), the result is also the random overapproximation.


```
(* abstract_To_Linear_Syntax.ml *)
open Abstract_Syntax
open Linear_Syntax
open Values
open Variables
(* Linearization of arithmetic operations *)
exception Not_constant
exception Not_linear
exception Abstract_To_Linear_Syntax_error
let rec linearize_aexp a =
 let n = (number_of_variables ()) in
  try
   match a with
   | (Abstract_Syntax.NAT i) ->
       (match (machine_int_of_string i) with
        | (ERROR_NAT _) -> RANDOM_AEXP
        | (NAT vi)      ->
```



```
            let l = Array.make (n+1) 0 in l.(n) <- vi;
              LINEAR_AEXP l)
   | (VAR v) -> (let l = Array.make (n+1) 0 in l.(v) <- 1;
        LINEAR_AEXP l)
   | RANDOM -> RANDOM_AEXP
   | (UPLUS a1) -> (linearize_aexp a1)
   | (UMINUS a1) -> (match linearize_aexp a1 with
        | RANDOM_AEXP -> RANDOM_AEXP
        | LINEAR_AEXP l1 ->
           let l = Array.make (n+1) 0 in
             (for i=0 to n do
                match machine_unary_minus (NAT l1.(i)) with
                | ERROR_NAT _ -> raise Abstract_To_Linear_Syntax_error
                | NAT v -> l.(i) <- v
              done;
              LINEAR_AEXP l))
   | (PLUS (a1, a2))  ->
      (match (linearize_aexp a1, linearize_aexp a2) with
```

```
       | (RANDOM_AEXP, _) | (_, RANDOM_AEXP) -> RANDOM_AEXP
       | (LINEAR_AEXP l1, LINEAR_AEXP l2) ->
          let l = Array.make (n+1) 0 in
           (for i=0 to n do
             match machine_binary_plus (NAT l1.(i)) (NAT l2.(i)) with
             | ERROR_NAT _ -> raise Abstract_To_Linear_Syntax_error
             | NAT v -> l.(i) <- v
             done;
             LINEAR_AEXP l))
   | (MINUS (a1, a2)) ->
       (match (linearize_aexp a1, linearize_aexp a2) with
        | (RANDOM_AEXP, _) | (_, RANDOM_AEXP) -> RANDOM_AEXP
        | (LINEAR_AEXP l1, LINEAR_AEXP l2) ->
           let l = Array.make (n+1) 0 in
            (for i=0 to n do
              match machine_binary_minus (NAT l1.(i)) (NAT l2.(i)) with
              | ERROR_NAT _ -> raise Abstract_To_Linear_Syntax_error
              | NAT v -> l.(i) <- v
```



```
             done;
             LINEAR_AEXP l))
   | (TIMES (a1, a2)) ->
      (match (linearize_aexp a1, linearize_aexp a2) with
       | (RANDOM_AEXP, _) | (_, RANDOM_AEXP) -> RANDOM_AEXP
       | (LINEAR_AEXP l1, LINEAR_AEXP l2) ->
          try
           for i=0 to n-1 do if l1.(i)<>0 then raise Not_constant done;
            let l = Array.make (n+1) 0 in
             for i=0 to n do
              match machine_binary_times (NAT l1.(n)) (NAT l2.(i)) with
              | ERROR_NAT _ -> raise Abstract_To_Linear_Syntax_error
              | NAT v -> l.(i) <- v
             done;
             LINEAR_AEXP l
          with Not_constant ->
           try
            for i=0 to n-1 do if l2.(i)<>0 then raise Not_linear done;
```



```
             let l = Array.make (n+1) 0 in
              for i=0 to n do
               match machine_binary_times (NAT l1.(i)) (NAT l2.(n)) with
               | ERROR_NAT _ -> raise Abstract_To_Linear_Syntax_error
               | NAT v -> l.(i) <- v
              done;
              LINEAR_AEXP l
           with Not_linear -> RANDOM_AEXP)
   | (DIV (a1, a2)) ->
      (match (linearize_aexp a1, linearize_aexp a2) with
       | (RANDOM_AEXP, _) | (_, RANDOM_AEXP) -> RANDOM_AEXP
       | (LINEAR_AEXP l1, LINEAR_AEXP l2) ->
          try
           for i=0 to n-1 do if l2.(i)<>0 then raise Not_constant done;
            if (l2.(n) = 0) then
               RANDOM_AEXP
            else
             let l = Array.make (n+1) 0 in
```



```
              for i=0 to n do
               match machine_binary_div (NAT l1.(i)) (NAT l2.(i)) with
               | ERROR_NAT _ -> raise Abstract_To_Linear_Syntax_error
               | NAT v -> l.(i) <- v
              done;
             LINEAR_AEXP l
          with Not_constant -> RANDOM_AEXP)
   | (MOD (a1, a2))   -> RANDOM_AEXP
    with Abstract_To_Linear_Syntax_error -> RANDOM_AEXP
(* Linearization of boolean operations *)
let rec linearize_bexp b =
  match b with
  | TRUE           -> LTRUE
  | FALSE          -> LFALSE
  | (EQ (a1, a2))  ->
      (match (linearize_aexp a1), (linearize_aexp a2) with
       | (RANDOM_AEXP, _) | (_, RANDOM_AEXP) -> RANDOM_BEXP
       | (LINEAR_AEXP l1, LINEAR_AEXP l2) ->
```

```
          let t = Array.make ((number_of_variables ())+1) 0 in
            for i=0 to (number_of_variables ()) do
          t.(i) <- l2.(i) - l1.(i)
        done;
             LEQ t)
| (LT (a1, a2))  ->
      (match (linearize_aexp a1),
       (linearize_aexp (MINUS (a2, (Abstract_Syntax.NAT "1")))) with
       | (RANDOM_AEXP, _) | (_, RANDOM_AEXP) -> RANDOM_BEXP
       | (LINEAR_AEXP l1, LINEAR_AEXP l2) ->
          let t = Array.make ((number_of_variables ())+1) 0 in
            for i=0 to (number_of_variables ()) do
          t.(i) <- l2.(i) - l1.(i)
        done;
             LGE t)
  | (AND (b1, b2)) ->
      (match (linearize_bexp b1), (linearize_bexp b2) with
       | (LFALSE, _) | (_, LFALSE) -> LFALSE
```



```
       | (LTRUE, b) -> b
       | (a, LTRUE) -> a
       | (RANDOM_BEXP, _) | (_, RANDOM_BEXP) -> RANDOM_BEXP
       | (LAND l1, LAND l2) -> LAND (l1@l2)
       | (LAND l, b) -> LAND (l@[b])
       | (b, LAND l) -> LAND (b::l)
       | (b1', b2') -> LAND [b1';b2'])
  | (OR (b1, b2))  ->
      (match (linearize_bexp b1), (linearize_bexp b2) with
       | (LTRUE, _) | (_, LTRUE) -> LTRUE
       | (LFALSE, b) -> b
       | (a, LFALSE) -> a
       | (RANDOM_BEXP, _) | (_, RANDOM_BEXP) -> RANDOM_BEXP
       | (LOR l1, LOR l2) -> LOR (l1@l2)
       | (LOR l, b) -> LOR (l@[b])
       | (b, LOR l) -> LOR (b::l)
       | (b1', b2') -> LOR [b1';b2'])
(* Linearization of commands *)
```



```
let rec linearize_com c =
  match c with
  | SKIP (l1, l2) -> (LSKIP (l1, l2))
  | ASSIGN (l1, v, a, l2) -> (LASSIGN (l1, v, (linearize_aexp a), l2))
  | SEQ (l1, cl, l2) -> (LSEQ (l1, (linearize_com_list cl), l2))
  | IF (l1, b, nb, ct, cf, l2) ->
      (LIF (l1, (linearize_bexp b), (linearize_bexp nb),
                 (linearize_com ct), (linearize_com cf), l2))
  | WHILE (l1, b, nb, c, l2) ->
      (LWHILE (l1, (linearize_bexp b), (linearize_bexp nb),
                                       (linearize_com c), l2))
and linearize_com_list cl =
  match cl with
  | [] -> []
  | c :: cl' -> (linearize_com c) :: (linearize_com_list cl')
```


Note: an alternative (as in ASTRÉE) is to use an abstract domain which keeps track of linear subexpressions (together with rounding errors) [1, 2]. The other numerical domains then use this symbolic domain to obtain a symbolic value of expressions which can then be evaluated in the abstract. In this pedagogical abstract interpreter, we use a much simpler hard-coding of linear expressions (with a static abstraction).

Reference

[1] Antoine Miné. "Relational abstract domains for the detection of floating-point run-time errors". In ESOP 2004 — European Symposium on Programming, D. Schmidt (editor), Mar. 27 — Apr. 4, 2004, Barcelona, Lecture Notes in Computer Science 2986, pp. 3—17, Springer.

[2] Antoine Miné. "Weakly relational numerical abstract domains". PhD, École polytechnique, 6 December 2004.

Pretty-printing linear programs

```
(* lpretty_Print.mli *)
open Linear_Syntax
val lpretty_print : lcom ->  unit

(* lpretty_Print.ml *)
open Linear_Syntax
open Variables
open Labels
(* print linearized arithmetic expressions *)
let rec print_Laexp a = match a with
| RANDOM_AEXP ->
    (print_string "?")
| LINEAR_AEXP l ->
    (let print_var v = (print_int l.(v); print_string ".";
                                           print_variable v)
```



```
     and print_plus v = (print_string " + ")
     in (map_variables print_var print_plus;
         print_string " + ";
         print_int l.(number_of_variables ())))
(* print linear boolean expressions *)
let rec print_Lbexp b = match b with
| LTRUE          -> print_string "true"
| LFALSE         -> print_string "false"
| RANDOM_BEXP    -> print_string "??"
| (LGE l)        -> (let print_var v = (print_int l.(v);
                               print_string "."; print_variable v)
                     and print_plus v = (print_string " + ")
                     in (map_variables print_var print_plus;
                         print_string " + ";
                         print_int l.(number_of_variables ());
                         print_string " >= 0"))
| (LEQ l)        -> (let print_var v = (print_int l.(v);
                               print_string "."; print_variable v)
```



```
                     and print_plus v = (print_string " + ")
                     in (map_variables print_var print_plus;
                         print_string " + ";
                         print_int l.(number_of_variables ());
                         print_string " = 0"))
| (LOR  bl)      -> print_string "("; (print_Lbexp_or bl);
                                         print_string ")"
| (LAND bl)      -> print_string "("; (print_Lbexp_and bl);
                                         print_string ")"
and print_Lbexp_or bl = match bl with
| []        -> ()
| b :: []   -> print_Lbexp b
| b :: bl' -> print_Lbexp b; print_string " | "; print_Lbexp_or bl'
and print_Lbexp_and bl = match bl with
| []        -> ()
| b :: []   -> print_Lbexp b
| b :: bl' -> print_Lbexp b; print_string " & "; print_Lbexp_or bl'
exception Error_lpretty_print of string
```



```
(* print linearized program *)
let lpretty_print c =
  let rec print_margin n =
    if n > 0 then (print_string "  "; print_margin (n-1))
    else ()
  and print_margin_label n l =
    (print_margin n;
     print_label l; print_string ": ";
     print_newline ())
  and print_seq n s =
    match s with
    | []      -> raise (Error_lpretty_print
                                    "empty sequence of commands")
    | [c']    -> print_com n c'
    | h :: s' -> (print_com n h;
                  print_string ";"; print_newline ();
                   print_seq n s')
  and print_com n c' =
```

```
    match c' with
    | (LSKIP (l,m)) ->
        print_margin_label n l; print_margin (n+1);
        print_string "skip"
    | (LASSIGN (l,v,a,m)) ->
       print_margin_label n l;
       print_margin (n+1); print_variable v; print_string " := ";
       print_Laexp a
    | (LSEQ (l,s,m)) ->
        print_seq n s
    | (LIF (l,b,nb,t,f,m)) ->
        print_margin_label n l; print_margin (n+1);
        print_string "if "; print_Lbexp b;
        print_string " then"; print_newline();
        print_com_line (n+2) t;
        print_margin (n+1); print_string "else";
        print_string " {"; print_Lbexp nb; print_string "}";
        print_newline ();
```



```
        print_com_line (n+2) f;
        print_margin (n+1); print_string "fi"
    | (LWHILE (l,b,nb,c'',m))  ->
        print_margin_label n l; print_margin (n+1);
        print_string "while "; print_Lbexp b;
        print_string " do"; print_newline();
        print_com_line (n+2) c'';
        print_margin (n+1); print_string "od";
        print_string " {"; print_Lbexp nb; print_string "}"
  and print_com_line n c' =
     print_com n c'; print_newline ();
     print_margin_label n (Linear_Syntax.after c')
  in
     print_com_line 0 c
```


– Example:

```
% cat ../Examples/example29.sil% example29.sil %
n := ?; i := n;
while (i <> 1) do
 j := 0;
 while(j <> i) do
   j := j + 1
 od;
 i := i - 1
od;;
```



```
% ./a.out ../Examples/example29.sil
** Program:
0:
  n := ?;
1:
  i := n;
2:
  while ((i < 1) | (1 < i)) do
    3:
      j := 0;
    4:
      while ((j < i) | (i < j)) do
        5:
          j := (j + 1)
        6:
      od {(j = i)};
    7:
      i := (i - 1)
    8:
  od {(i = 1)}
9:
```

```
** Linearized program:
0:
  n := ?;
1:
  i := 1.n + 0.i + 0.j + 0;
2:
  while (0.n + -1.i + 0.j + 0 >= 0 | 0.n + 1.i + 0.j + -2 >= 0) do
    3:
      j := 0.n + 0.i + 0.j + 0;
    4:
      while (0.n + 1.i + -1.j + -1 >= 0 | 0.n + -1.i + 1.j + -1 >= 0) do
        5:
          j := 0.n + 0.i + 1.j + 1
        6:
      od {0.n + 1.i + -1.j + 0 = 0};
    7:
      i := 0.n + 1.i + 0.j + -1
    8:
  od {0.n + -1.i + 0.j + 1 = 0}
9:
```


A generic linear relational abstract interpreter

Linear abstraction

- In relational analyzes, expressions must be analyzed as a whole (as opposed to the compositional analysis by structural induction in the case of non-relational analyzes)
- Most relational analyses consider only linear expressions of the form:

$$x_i := a_0 x_0 + \dots + a_n x_n$$

$$a_0 x_0 + \dots + a_n x_n \geqslant a_{n+1}$$

where the $x_0, \dots, x_n$ denote the values of the variables and $a_0 \dots a_n\, a_{n+1}$ are numeric coefficients (in $\mathbb{Z}$, $\mathbb{Q}$ or $\mathbb{R}$, see Miné [2004] for floats)

- So a generic relational analyzer may be specialized to linear expressions
  - keeping all linear expressions in the above standardized form
  - approximating the other non-linear expressions by a random choice (?).

Abstract syntax

- The basic files `lexer.mll` and `parser.mly` are unchanged.
- The variables are represented by a natural number, so the symbol table is essentially unchanged, but for the inclusion of functions `map_variables` and `string_of_variable`:

```
val map_variables       : (variable -> unit) -> (variable -> unit) ->
unit
val string_of_variable  : variable -> string
```

which are implemented as follows:


```
(* string of variable v in symbol table *)
exception Error_string_of_variable of string
let string_of_variable v =
 let p = ref !symb_table in
  for k = 0 to (v - 1) do
    if !p = [] then
      raise (Error_string_of_variable "too large")
    else
      p := tl !p
  done;
  if !p = [] then
    raise (Error_string_of_variable "not found")
  else
    hd !p
```



```
(* map_variables p q = (p v0); (q v1); (p v1); ... ;    *)
(*                     (p vn-2); (q vn-1); (p vn-1)     *)
(* where v0, ..., vn-1 are the n >= 0 program variables *)
let map_variables p q =
  if (number_of_variables ()) > 0 then
    (p 0;
     for v = 1 to ((number_of_variables ()) - 1) do
       q v;
       p v
     done)
  else
    ()
```

These functions are imported by the `Variables` modules (which no longer hides the internal implementation of variables by their natural rank, which is the representation considered in available libraries)


```
(* variables.mli *)
open Symbol_Table
type variable = Symbol_Table.variable
val number_of_variables : unit -> int
val for_all_variables : (variable -> 'a) -> unit
val print_variable : variable -> unit
val map_variables : (variable -> unit) -> (variable -> unit) -> unit
val string_of_variable : variable -> string
(* variables.ml *)
open Symbol_Table
type variable = Symbol_Table.variable
let number_of_variables = number_of_variables
let for_all_variables   = for_all_variables
let print_variable      = print_variable
let map_variables       = map_variables
let string_of_variable  = string_of_variable

```

- The program abstract syntax is unchanged, as well as the translation frm concrete to abstract syntax, as found in the files `abstract_Syntax.ml`, `concrete_To_Abstract_Syntax.mli`, `concrete_To_Abstract_Syntax.ml`, `labels.mli`, `labels.ml`, `program_To_Abstract_Syntax.mli`, `program_To_Abstract_Syntax.ml`, `pretty_Print.mli`, `pretty_Print.ml`
- The modules handling the concrete values are unchanged: `values.mli`, `values.ml`

Generic linear relational abstract domains

- The signature of the linear relational abstract domains is as follows:

```
(* aenv.mli *)
open Linear_Syntax
open Array
open Variables
(* set of environments *)
type t
(* relational library initialization *)
val init : unit -> unit
(* relational library exit *)
val quit : unit -> unit
(* infimum *)
val bot : unit -> t
(* check for infimum *)
```



```
val is_bot : t -> bool
(* uninitialization *)
val initerr : unit -> t
(* supremum *)
val top : unit -> t
(* least upper bound *)
val join : t -> t -> t
(* greatest lower bound *)
val meet : t -> t -> t
(* approximation ordering *)
val leq : t -> t -> bool
(* equality *)
val eq : t -> t -> bool
(* printing *)
val print : t -> unit
(* collecting semantics of assignment                                  *)
(* f_ASSIGN x f r =  {e[x <- i] | e in r /\ i in f({e}) cap I } *)
val f_ASSIGN : variable -> laexp -> t -> t
```



```
(* collecting semantics of boolean expressions                         *)
(* f_LGE a r = {e in r | a0.v0+...+an-1.vn-1 >= an                     *)
val f_LGE : (int array) -> t -> t
(* f_LEQ a r = {e in r | a0.v0+...+an-1.vn-1 = an                      *)
val f_LEQ : (int array) -> t -> t
(* convergence acceleration *)
(* widening *)
val widen : t -> t -> t
(* narrowing *)
val narrow : t -> t -> t
```

The abstract domains include:

- The initialization and finalization of the library used to implement the abstract domain
- The lattice structure
- The convergence acceleration operators (if necessary)
- The forward analysis of assignment

$$x := a_0 x_0 + \ldots + a_x x_n + a_{n+1}$$

by `f_ASSIGN x f r` where $\mathrm{f} = [a_0, a_1; \ldots; a_n; a_{n+1}]$ which an upper-approximation of

$$\alpha(\{\rho[\mathrm{x} := \mathrm{v}] \mid \rho \in \gamma(\mathrm{r}) \wedge \mathrm{v} \in [\![\mathrm{f}]\!]\rho \cap \mathbb{I}\})$$
$$\text{and } [\![\mathrm{f}]\!]\rho \stackrel{\text{def}}{=} a_0.\rho(x_0) + \ldots . a_n.\rho(x_n) + a_{n+1}$$

which is Ω_{a} is case of error in the machine computation of the expression $a_0.\rho(x_0) + \ldots . a_n.\rho(x_n) + a_{n+1}$.

- The analysis of boolean expressions

$$a_0 x_0 + \ldots + a_x x_n \geq a_{n+1}$$

by `f_LGE a r` where $\mathrm{a} = [a_0, a_1; \ldots; a_n; a_{n+1}]$ which an upper-approximation of

$$\alpha(\{\rho \in \gamma(\mathrm{r}) \mid a_0.\rho(x_0) + \ldots . a_n.\rho(x_n) \geq a_{n+1}\})$$

which is ff is case of error in the machine computation of the expression $a_0.\rho(x_0) + \ldots . a_n.\rho(x_n)$.

- It is sometimes useful to handle equality tests

$$a_0 x_0 + \ldots + a_x x_n = a_{n+1}$$

(which otherwise has to be handled by two opposite inequalities)

Generic linear relational analysis of boolean expressions

- The boolean expressions can be handled generically:

```
(* abexp.mli *)
open Linear_Syntax
open Aenv
(* abstract interpretation of boolean operations *)
val a_bexp : lbexp -> Aenv.t -> Aenv.t
```

- The implementation is as follows:

```
(* abexp.ml *)
open Linear_Syntax
open Aenv
(* abstract interpretation of linearized boolean operations *)
exception Error of string
let rec a_bexp b r =
```

```
  match b with
  | RANDOM_BEXP -> r
  | LTRUE       -> r
  | LFALSE      -> (Aenv.bot ())
  | (LGE a)     -> (Aenv.f_LGE a r)
  | (LEQ a)     -> (Aenv.f_LEQ a r)
  | (LAND l)    -> let rec andlist l = match l with
                  | []     -> (raise (Error "empty LAND incoherence"))
                  | b'::[] -> a_bexp b' r
                  | b'::l' -> Aenv.meet (a_bexp b' r) (andlist l')
                 in andlist l
  | (LOR l)     -> let rec orlist l = match l with
                  | []     -> (raise (Error "empty LOR incoherence"))
                  | b'::[] -> a_bexp b' r
                  | b'::l' -> Aenv.join (a_bexp b' r) (orlist l')
                 in orlist l

```


Generic linear relational analysis of commands

– The structure is quite similar to the non-relational case, but for the fact that the analysis operates on the linear abstraction of the program and non longer on the program abstract syntax):

```
(* acom.mli *)
open Linear_Syntax
open Aenv
(* forward abstract interpretation of commands *)
val acom : lcom -> Aenv.t -> label -> Aenv.t
```

– The implementation is as follows:

```
(* acom.ml *)
open Linear_Syntax
open Aenv
```



```
open Abexp
open Fixpoint
(* collecting semantics of commands *)

exception Error of string
let rec acom c r l =
  match c with
  | (LSKIP (l', l'')) ->
      if (l = l') then r
      else if (l = l'') then r
      else (raise (Error "SKIP incoherence"))
  | (LASSIGN (l',x,a,l'')) ->
      if (l = l') then r
      else if (l = l'') then
                f_ASSIGN x a r
      else (raise (Error "ASSIGN incoherence"))
  | (LSEQ (l', s, l'')) ->
      (acomseq s r l)
```



```
  | (LIF (l', b, nb, t, f, l'')) ->
      (if (l = l') then r
      else if (incom l t) then
         (acom t (a_bexp b r) l)
      else if (incom l f) then
         (acom f (a_bexp nb r) l)
      else if (l = l'') then
                (let rt = (acom t (a_bexp b r)  (after t))
                 and rf = (acom f (a_bexp nb r) (after f))
                 in join rt rf)
      else (raise (Error "IF incoherence")))
  | (LWHILE (l', b, nb, c', l'')) ->
     let f x = join r (acom c' (a_bexp b x) (after c'))
     in let i = lfp (bot ()) leq widen narrow f in
      (if (l = l') then i
       else if (incom l c') then (acom c' (a_bexp b i) l)
       else if (l = l'') then (a_bexp nb i)
       else (raise (Error "WHILE incoherence")))
```

```
and acomseq s r l = match s with
  | []   -> raise (Error "empty SEQ incoherence")
  | [c]  -> if (incom l c) then (acom c r l)
            else (raise (Error "SEQ incoherence"))
  | h::t -> if (incom l h) then (acom h r l)
            else (acomseq t (acom h r (after h)) l)
```


– In general an analyzer has both relational domains and non-relational domains, which must be combined through a reduced product

– In general languages have aliases and so the abstraction must map program variables to abstract variables of the abstract domain. It is then useful to have in the abstract domain variables with destructive assignment as well as variables with cumulative assignment

Fixpoint computation with widening/narrowing

– The fixpoint computation `fixpoint.mli` and `fixpoint-notrace.ml` is unchanged

– We add the ability to trace fixpoint computations to observe the iterates in `fixpoint-trace.ml`

– The choice of `fixpoint.ml` between `fixpoint-notrace.ml` or `fixpoint-trace.ml` is done in the `makefile` prior to starting the analysis

```
(* fixpoint.ml *)
open Aenv
(* iteration of f from prefixpoint x with ordering c and widening w *)
let rec luis x c w f =
```



```
  print_string "luis: x =\n";
  print x;
  let x' = (f x) in
    print_string "luis: f(x) =\n";
    print x';
    if (c x' x) then
      (print_string "luis: f(x) <= x, convergence\n";
       x)
    else
      (let x'' = (w x x') in
       print_string "luis: x \\/ f(x) =\n";
       print x'';
       luis x'' c w f)
(* iteration of f from postfixpoint x with ordering c and narrowing n *)
let rec llis x c n f =
  print_string "llis: x =\n";
  print x;
    let x' = (f x) in
```

```
      print_string "llis: f(x) =\n";
      print x';
      let x'' = (n x x') in
        print_string "llis: x /\\ f(x) =\n";
        print x'';
        if (c x x'') then
          (print_string "llis: x <= x /\\ f(x), convergence\n";
           x')
            else llis x'' c n f
(* lfp x c w n f : iterative computation of a c-postfixpoint of f *)
(* c-greater than or equal to the prefixpoint x (x <= f(x)) with  *)
(* widening w and narrowing n                                      *)
let lfp x c w n f = llis (luis x c w f) c n f
(* gfp x c n f : iterative computation of a c-postfixpoint of f   *)
(* c-less than or equal to the postfixpoint x (f(x) <= x) with    *)
(* narrowing n                                                     *)
let gfp x c n f = llis x c n f
```


The generic linear relational abstract interpreter

```
(* main.ml *)
open Program_To_Abstract_Syntax
open Labels
open Pretty_Print
open Lpretty_Print
open Abstract_To_Linear_Syntax
open Linear_Syntax
open Aenv
open Acom
let _ =
  let arg = if (Array.length  Sys.argv) = 1 then ""
            else Sys.argv.(1) in
     Random.self_init ();
```



```
    let p = (abstract_syntax_of_program arg) in
      (print_string "** Program:\n";
       pretty_print p;
       let p' = (linearize_com p) in
         print_string "** Linearized program:\n";
         lpretty_print p';
         init ();
         print_string "** Precondition:\n";
         print (initerr ());
         print_string "** Postcondition:\n";
         print (acom p' (initerr ()) (after p'));
         quit ())
```


makefile

```
EXAMPLES = ../Examples

SOURCES = \
symbol_Table.mli \
symbol_Table.ml \
variables.mli \
variables.ml \
abstract_Syntax.ml \
concrete_To_Abstract_Syntax.mli \
concrete_To_Abstract_Syntax.ml \
labels.mli \
labels.ml \
parser.mli \
parser.ml \
```

```
lexer.ml \
program_To_Abstract_Syntax.mli \
program_To_Abstract_Syntax.ml \
pretty_Print.mli \
pretty_Print.ml \
values.mli \
values.ml \
linear_Syntax.mli \
linear_Syntax.ml \
abstract_To_Linear_Syntax.mli \
abstract_To_Linear_Syntax.ml \
lpretty_Print.mli \
lpretty_Print.ml \
aenv.mli \
aenv.ml \
abexp.mli \
abexp.ml \
fixpoint.mli \
```



```
fixpoint.ml \
acom.mli \
acom.ml \
main.ml

.PHONY : help
help :
   @echo ""
   @echo "Forward relational static analysis:"
   @echo "make [help]      : this help"
   @echo "make pol         : polyhedral analysis"
   @echo "./a.out file.sil : analyze file.sil"
   @echo "make examples    : analyze all examples"
   @echo "make clean       : remove auxiliary files"
   @echo ""

.PHONY : pol
pol:
```



```
   @/bin/rm -f aenv.ml
   @ln -s ../Relational-FW/Polyhedra/aenv.ml aenv.ml
   @echo "Polyhedral analysis"
   ocamlyacc parser.mly
   ocamllex lexer.mll
   ocamlc -custom -I /usr/local/lib -I /usr/local/lib/ocaml \
     -cclib "-L/usr/local/lib -L/usr/local/lib/ocaml -lpolkag_caml \
     -lpolkag -lgmp -lcamlidl" \
     polka.cma ${SOURCES}

include ${EXAMPLES}/makefile

.PHONY : clean
clean :
   /bin/rm -f *.cmi *.cmo *~ a.out lexer.ml parser.mli parser.ml
```


Polyhedral relational static analysis

Polyhedral abstract domain

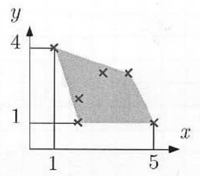
- We consider a vector space $\mathbb{V}$ over a field $\mathbb{F}$, that is a set closed under finite vector addition and mutiplication by a scalar in $\mathbb{F}$
- Typically $\mathbb{F} = \mathbb{Q}$ or $\mathbb{F} = \mathbb{R}$[7] and the vector space is the corresponding Euclidean space $\mathbb{V} = \mathbb{F}^n$
- The abstract predicates are affine inequalities $AX \leq B$ i.e. closed convex polyhedra over the field $\mathbb{F}$

$$\begin{cases} \sum_{i=1}^{n} a_{j,i}.x_i \leq a_{j,n+1} \\ j = 1, \ldots, m \end{cases}$$

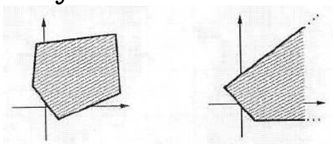
[7] which is an unsolved soundness problem whence implementing the algorithms in $\mathbb{R}$ with floats.

- Example:

$$\begin{aligned} & 3x + y \geq 7 \\ \wedge\ & 2x + y \leq 11 \\ \wedge\ & \quad\ \ y \geq 1 \\ \wedge\ & x + 3y \leq 13 \end{aligned}$$



- The polyhedra may be unbounded:



Example of polyhedral static analysis

```
program PL;
   var I, J : integer;
begin
   I := 2; J := 0;
   while ... do begin
      { 2J+2 ≤ I  ∧  0 ≤ J }
      if ... then begin
         I := I + 4;
         { 2J+6 ≤ I  ∧  0 ≤ J }
      end else begin
         I := I + 2; J := J + 1;
         { 2J+2 ≤ I  ∧  1 ≤ J }
      end;
      { 2J+2 ≤ I  ∧  6 ≤ I+2J  ∧  0 ≤ J }
   end;
end.
```


- A relation is discovered between I and J although they never appear in the same command (thus showing the limits of heuristic methods)

```
% example40.sil %
I:=2;J:=0;B:=?;
while B<>0 do
  if B<>1 then
     I:=I+1
  else
     I:=I+2;
     J:=J+1
  fi
od;;
** Program:
...
```

```
** Linearized program:
  I := 0.I + 0.J + 0.B + 2;
  J := 0.I + 0.J + 0.B + 0;
  B := ?;
  while (0.I + 0.J + -1.B + -1 >= 0 | 0.I + 0.J + 1.B + -1 >= 0) do
      if (0.I + 0.J + -1.B + 0 >= 0 | 0.I + 0.J + 1.B + -2 >= 0) then
          I := 1.I + 0.J + 0.B + 1
      else {0.I + 0.J + -1.B + 1 = 0}
          I := 1.I + 0.J + 0.B + 2;
          J := 0.I + 1.J + 0.B + 1
      fi
  od {0.I + 0.J + -1.B + 0 = 0}
** Precondition:
{1>=0}
** Postcondition:
{B=0,1>=0,J>=0,I>=2J+2}
%
```


Affine hull

– Given a set $V \in \mathbb{F}^{n \times p}$ representing a finite set of points $\{V_1, \ldots, V_p\}$, an *affine combination of points* in V is

$$\sum_{j=1}^{p} \lambda_j V_j \quad \text{where } \forall j \in [1,p] : \lambda_j \geq 0 \wedge \sum_{j=1}^{p} \lambda_j = 1$$

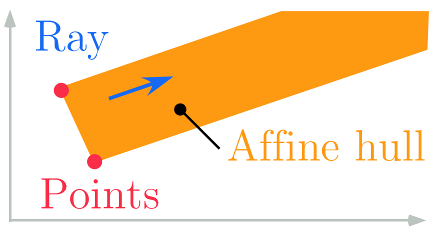
– To handle unbounded polyhedra, also consider a set $R \in \mathbb{F}^{n \times r}$ representing a set of rays $\{R_1, \ldots, R_r\}$ (i.e., intuitively, points at infinite). An *affine combination of rays* in R is

$$\sum_{k=1}^{r} \mu_k R_k \quad \text{where } \forall j \in [1,p] : \forall k \in [1,r] : \mu_k \geq 0$$

– The *affine hull* (also *convex hull*) of $\langle V, R\rangle$ is:

$$\{(\sum_{j=1}^{p} \lambda_j V_j) + (\sum_{k=1}^{r} \mu_k R_k) \mid \forall j \in [1,p] : \lambda_j \geq 0 \wedge \sum_{j=1}^{p} \lambda_j = 1 \wedge \forall k \in [1,r] : \mu_k \geq 0\}$$

– Example:



– The affine hull would be nice as an abstraction function — but — it is not defined for an infinite number of points/rays

– We use a concretization function only

Representation of polyhedra by constraints

We have two dual representations by constraints and systems of generators

- Representation by constraints: $\langle A,\ B\rangle$ where $A \in \mathbb{F}^{m\times n}$ and $B \in \mathbb{F}^{m}$ representing

$$\gamma(\langle A,\ B\rangle) \stackrel{\text{def}}{=} \{X \in \mathbb{F}^{n} \mid AX \geq B\}$$

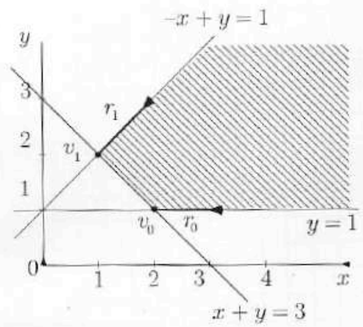
Representation of polyhedra by generators

- Representation by generators: $\langle V,\ R\rangle$ where $V \in \mathbb{F}^{n\times p}$ represents a set of vertices $\{V_1, \ldots, V_p\}$ while $R \in \mathbb{F}^{n\times r}$ represents a set of rays $\{R_1, \ldots, R_r\}$ [8] which encodes the concrete set

$$\gamma(\langle V,\ R\rangle) = \{(\sum_{j=1}^{p} \lambda_j V_j) + (\sum_{k=1}^{r} \mu_k R_k) \mid \forall j \in [1,p] : \lambda_j \geq 0 \wedge \sum_{j=1}^{p} \lambda_j = 1 \wedge \forall k \in [1,r] : \mu_k \geq 0\}$$

[8] It can be more efficient in the frame representation to use lines to represent rays in opposite directions

Example



System of linear inequalities:

$$P = \left\{\langle x,\ y\rangle \;\middle|\; \left(\begin{array}{rcl} y & \geq & 1 \\ x + y & \geq & 3 \\ -x + y & \leq & 1 \end{array}\right)\right\}$$

System of generators:

$$V = \left\{v_0\binom{2}{1}, v_1\binom{1}{2}\right\}$$

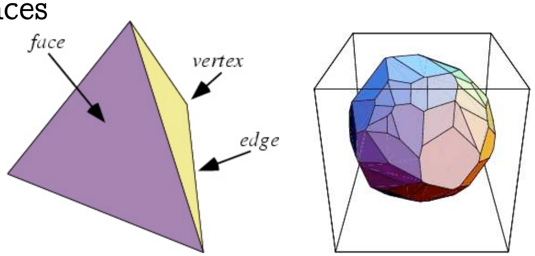
$$R = \left\{r_0\binom{1}{0}, r_1\binom{1}{1}\right\}$$

$$P = \left\{\sum_{v_i\in V} \lambda_i . v_i + \sum_{r_j \in R} \mu_j . r_j \;\middle|\; \lambda_i \geq 0,\ \sum_i \lambda_i = 1,\ \mu_j \geq 0\right\}$$

Minimal representations

- The representation by constraints $\langle A,\ B\rangle$ is minimal whenever no constraint can be eliminated without changing the polyhedron $\gamma(\langle A,\ B\rangle)$
- The representation by a system of generators $\langle V,\ R\rangle$ is minimal when no vertice of ray can be eliminated without changing the polyhedron $\gamma(\langle V,\ R\rangle)$

– There is no bound on the size of these (minimal) representations since polyhedra can have an arbitrary number of faces



Why two representations?

– Some operations are much more simple to define on one representation than on the other

– Some operations are much more efficient on one representation than on the other

– It is necessary to convert from one representation to the other

Conversion of constraints to generators by Chenikova algorithm

– Chernikova [3] algorithm computes iteratively the system of generators of a polyhedron P given by a system of linear inequalities $AX \geq B$ by successive intersections

References

[3] N.V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Comutational Mathematics and Mathematical Physics*, 8(6):282–293,1968.

Chenikova algorithm

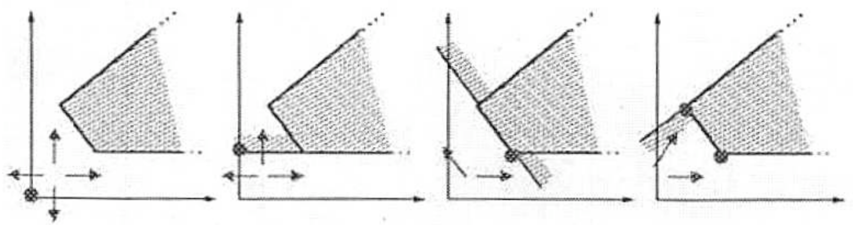
– Start with $P_0 = \mathbb{Q}^n$ given by the system of generators $V_0 = \{\vec{0}\}$ and $R_0 = \{\vec{i_1}, \ldots, \vec{i_n}, -\vec{i_1}, \ldots, -\vec{i_n}\}$ where $\{\vec{i_1}, \ldots, \vec{i_n}\}$ is a basis of $\mathbb{Q}^n$;

– At step k, intersect P_{k-1} with the k^{th} inequality $aX \geq b$ of $AX \geq B$, as follows:

1. any vertex $v \in V_{k-1}$ such that $av \geq b$ belongs to V_k;
2. any ray $r \in R_{k-1}$ such that $ar \geq 0$ belongs to R_k;
3. for any pair $\langle v,\, v'\rangle$ of vertices in V_{k-1} such that $av > b$ and $av' < b$, their convex combination $\frac{b-av'}{av-av'}.v - \frac{b-av}{av-av'}.v'$ belongs to V_k;

4. for any pair $\langle v, r\rangle$ of vertice and ray in $V_{k-1} \times R_{k-1}$ such that either $av > b$ and $ar < 0$ or $av < b$ and $ar > 0$, their positive combination $v + \frac{b-av}{ar}.r$ belongs to V_k;
5. for any pair $\langle r, r'\rangle$ of rays in R_{k-1} such that $ar > 0$ and $ar' < 0$, their positive combination $(ar').r - (ar).r'$ belongs to R_k.

Chenikova algorithm: example



Remarks on Chenikova algorithm

– In the worst case the algorithm can generate an exponential number of generators (an hypercube in dimmension n is described by $2n$ constraints but 2^n vertices)

– Moreover, the system of generators computed by Chernikova algorithm may not be minimal, redundant points and rays must be eliminated

– This can be done by Le Verge algorithm [1], to minimize the system of generators during its construction

– By duality, the algorithm can be used to convert a set of generators into a set of constraints

Reference

- N.V. Le Verge. A note on Chernikova's algorithm. *Research report 63*, IRISA, Rennes, February 1992.

Minimization of the system of generators by Le Verge algorithm

– A vertex v *saturates* an inequality $ax \geq b$ if $av = b$;
– A ray r *saturates* an inequality $ax \geq b$ if $ar = 0$;
– Let n_1 be the dimension of the least hyperplane containing P_k, and n_2 be the dimension of the greatest hyperplane contained in P_k,
  - a point v is an actual vertex of P_k if and only if it saturates $n_1 - n_2$ inequalities;
  - a vector r is an actual ray of P_k if and only if it saturates $n_1 - n_2 - 1$ inequalities.

The lattice structure of polyhedra

– Test for emptiness: P has no vertex;
– Test for inclusion: if P is defined by $AX \geq B$ and Q is defined by $\langle V, R \rangle$ then:

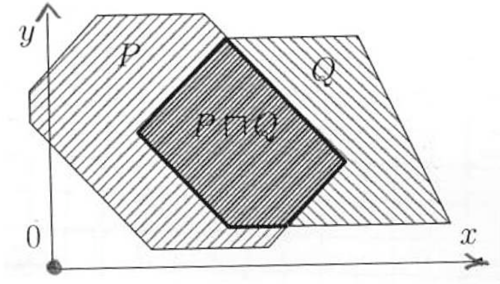
$$P \subseteq Q$$

if and only if:

$$\forall v \in V : Av \geq B \quad \wedge \quad \forall r \in R : Ar \geq 0$$

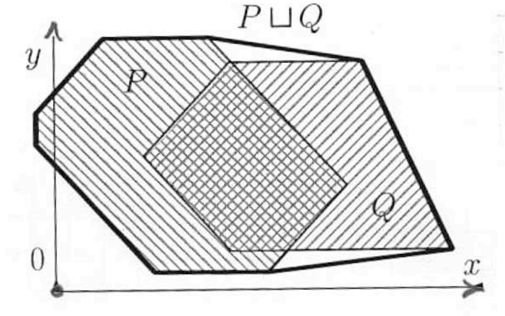
– Test for equality: $P = Q$ iff $P \subseteq Q \wedge P \supseteq Q$.

– Intersection $\sqcap$: conjunction of systems of linear inequalities;

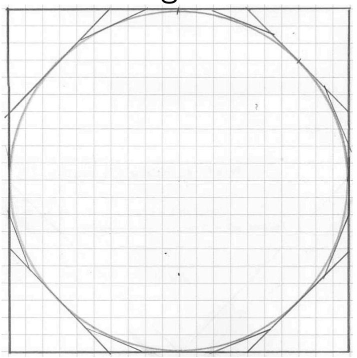


– These operations “$=\emptyset$?”, “$\subseteq$”, “$=$” and “$\sqcap$” are exact, i.e. same in concrete

– Union $\sqcup$: union of systems of generators
– This operation $\sqcup$ is the best possible, that is the convex hull of the concrete representations



- We get a lattice structure but not a complete lattice, as shown by the following counter-example:



The limit of the polyhedra is a disk (whence not a polyhedron)

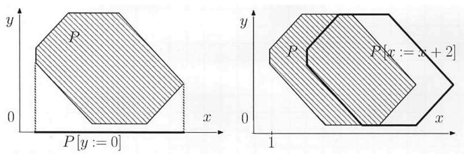
Abstract polyhedral transfer functions

- Linear transformation: If P defined by $\langle V,\ R\rangle$ then the image of P by $\lambda x \cdot Ax + B$ is:

$$P[x := Ax + B] \stackrel{\text{def}}{=} \{Ax + B \mid x \in P\} .$$

$P[x := Ax + B]$ is defined by $\langle V',\ R'\rangle$ where:

$$V' = \{Av + B \mid v \in V\}$$
$$R' = \{Ar \mid r \in R\}$$



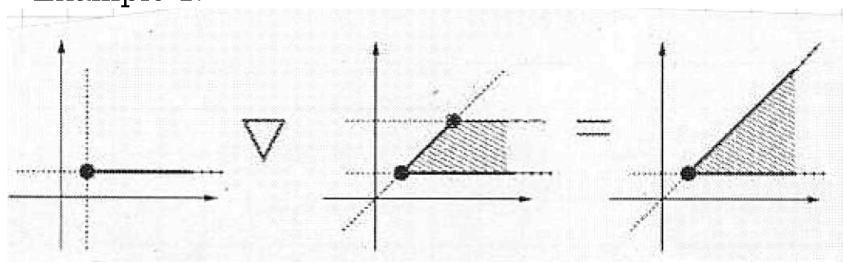
Widening of polyhedra

Informal definition:

- Polyhedron P is defined by $AX \geq B$ represented by the set of inequalities $I = \{\beta_1, \ldots, \beta_p\}$;
- Polyhedron Q is defined by $CX \geq D$ represented by the set of inequalities $J = \{\gamma_1, \ldots, \gamma_q\}$ and the generators $\langle V,\ R,\ L\rangle$;
- $P \triangledown Q$ is Q if P is empty;
- $P \triangledown Q$ is defined by the set of inequalities $I' \cup J'$ where:
  - I' is the set of inequalities $\beta_i \in I$ satisfied by all points (i.e. vertices V, rays R and lines L) of Q;
  - J' is the set of linear inequalities $\gamma_j \in J$ which can replace some $\beta_i \in I$ without changing polyhedron P.

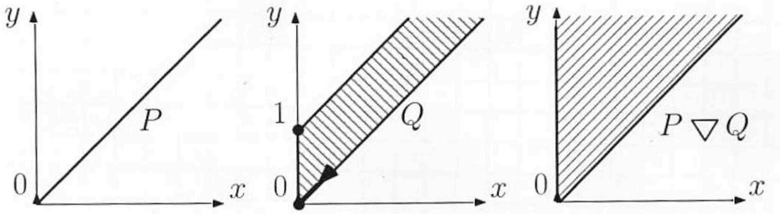
Examples of polyhedral widening

- Example 1:



– Example 2:

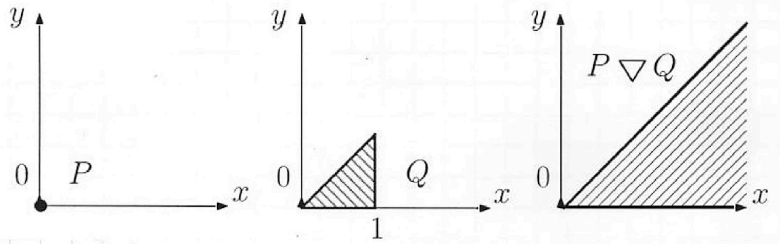
$$P = \{\langle x,\, y\rangle \mid 0 \leq x \wedge x \leq y \wedge y \leq x\};$$
$$Q = \{\langle x,\, y\rangle \mid 0 \leq x \leq y \leq x + 1\};$$
$$P \nabla Q = \{\langle x,\, y\rangle \mid 0 \leq x \leq y\};$$



– Example 3:

$$P = \{\langle x,\, y\rangle | x \leq 0 \wedge x \geq 0 \wedge y \leq 0 \wedge y \geq 0\};$$
$$Q = \{\langle x,\, y\rangle | 0 \leq y \leq x \leq 1\};$$
$$P \nabla Q = \{\langle x,\, y\rangle | 0 \leq y \leq x\},$$

instead of $\{\langle x,\, y\rangle | 0 \leq y \wedge 0 \leq x\}$;



Polyhedral widening improvements

Possible improvements:

- Thresholds: given a finite number of constraints T, we had to $X \nabla Y$ the constraints of T satisfied by X and Y
- Delay: $X \nabla Y$ can be replaced by $X \sqcup Y$ finitely many times
- Various heuristics have been proposed by [4] to improve the delay technique

References

[4] Roberto Bagnara, Patricia M. Hill, Elisa Ricci & Enea Zaffanella. "Precise Widening Operators for Convex Polyhedra" Proceedings of the 10th International Symposium on Static Analysis (SAS'03) (San Diego, California, USA, June 2003), vol. 2694 of Lecture Notes in Computer Science, R. Cousot, Ed., pp. 337-354.

Example 1 of polyhedral analysis

```
Generic-FW-REL-Abstract-Interpreter % ./a.out
../Examples/example41.sil
** Program:
  X := ?; Y := X;
  while (((0 < X) | (X = 0)) & ((0 < Y) | (Y = 0))) do
      Y := (Y + 1)
  od {((X < 0) | (Y < 0))}
** Linearized program:
  X := ?; Y := 1.X + 0.Y + 0;
  while ((1.X + 0.Y + -1 >= 0 | -1.X + 0.Y + 0 = 0) & (0.X + 1.Y + -1 >= 0
         | 0.X + -1.Y + 0 = 0)) do
      Y := 0.X + 1.Y + 1
  od {(-1.X + 0.Y + -1 >= 0 | 0.X + -1.Y + -1 >= 0)}
** Precondition:
{1>=0}
** Postcondition:
{1>=0,X<=Y,X+1<=0}
```

The loop invariant:

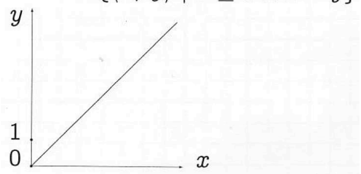
$$\{\langle x,\ y\rangle \mid x \geq 0 \wedge y \geq 0 \wedge x \leq y\}$$

is the least fixpoint of:

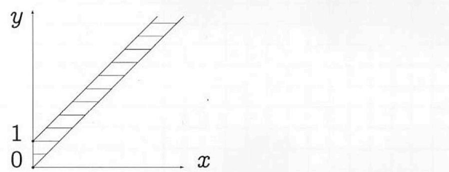
$$F(X) = \{\langle x,\ y\rangle \mid x \geq 0 \wedge y \geq 0 \wedge ((x = y) \vee (\langle x,\ y-1\rangle \in X))$$

The iterates are as follows:

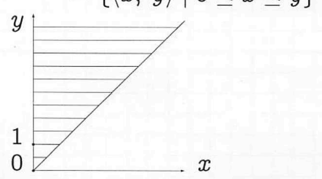
- $\hat{X}^0 = \emptyset$
- $\hat{X}^1 = F(\hat{X}^0)$
  $= \{\langle x,\ y\rangle \mid x \geq 0 \wedge x = y\}$



- $F(\hat{X}^1) = \{\langle x,\ y\rangle \mid 0 \leq x \leq y \leq x+1\}$



- $\hat{X}^2 = \hat{X}^1 \triangledown F(\hat{X}^1)$
  $= \{\langle x,\ y\rangle \mid 0 \leq x \leq y\}$



- $\hat{X}^3 = F(\hat{X}^2) = \hat{X}^2$

Example 2 of polyhedral analysis

```
X=2; I=0;
while ● (I<10) {
  if (?) X=X+2; else X=X-3;
  I=I+1;
} ♦
```

– The narrowing is simply a finite number of intersections.

– The iterations with widening/narrowing are as follows (the result is given at program point ●:

$\mathcal{X}^{\sharp 1}_{\bullet}$ $\{X = 2, I = 0\}$

$\mathcal{X}^{\sharp 2}_{\bullet}$ $\{X = 2, I = 0\} \triangledown \{X \in [-1; 4],\ I = 1\} =$
$\{I \geq 0,\ X \in [2 - 3I; 2I + 2]\}$

$\mathcal{X}^{\sharp 3}_{\bullet}$ $\{I \geq 0,\ X \in [2 - 3I; 2I + 2]\} \sqcap^{\sharp} \{I \in [0; 10],\ X \in [2 - 3I; 2I + 2]\} =$
$\{I \in [0; 10],\ X \in [2 - 3I; 2I + 2]\}$

– The final result at program point ♦ is:

$$\{I = 10,\ X \in [-28; 22]\}$$

– The example is handled by the polyhedral analyzer as follows:

```
Generic-FW-REL-Abstract-Interpreter % make pol
Polyhedral analysis
...
Generic-FW-REL-Abstract-Interpreter % cat ../Examples/example42.sil
% example42.sil %
B:=?; X:=2; I:=0;
while (I<10) do
  if B <> 0 then
    X := X+2
  else
    X:=X-3
  fi;
  I:=I+1
od;;
```



```
Generic-FW-REL-Abstract-Interpreter % ./a.out
../Examples/example42.sil
** Linearized program:
  B := ?;
  X := 0.B + 0.X + 0.I + 2; I := 0.B + 0.X + 0.I + 0;
  while 0.B + 0.X + -1.I + 9 >= 0 do
      if (-1.B + 0.X + 0.I + -1 >= 0 | 1.B + 0.X + 0.I + -1 >= 0) then
          X := 0.B + 1.X + 0.I + 2
      else {-1.B + 0.X + 0.I + 0 = 0}
          X := 0.B + 1.X + 0.I + -3
      fi;
      I := 0.B + 0.X + 1.I + 1
  od {(0.B + 0.X + 1.I + -11 >= 0 | 0.B + 0.X + -1.I + 10 = 0)}
** Precondition:
{1>=0}
** Postcondition:
{I=10,X<=22,X+28>=0}
```


Strict inequalities

– Polyhedral analysis can ve extended to include strict constraints:

$$\{X \mid AX \geq B \wedge A'X > B'\}$$

– A non-closed polyhedron on $\{X_1, \ldots, X_n\}$ is represented by a closed polyhedron on $X' = \{X_1, \ldots, X_n\} \cup \{X_\epsilon\}$ where X_ϵ is a fresh variable which value is assumed to be arbitrarily small

– $a_1X_1 + \ldots + a_nX_n \geq 0$ is encoded as $a_1X_1 + \ldots + a_nX_n + 0.X_\epsilon \geq 0$

– $a_1X_1 + \ldots + a_nX_n > 0$ is encoded as $a_1X_1 + \ldots + a_nX_n + e.X_\epsilon \geq 0$ where $e > 0$

– The concretization is

$$\begin{array}{c}\gamma_\epsilon(P) \stackrel{\text{def}}{=} \{\langle X_1, \ldots, X_n\rangle \mid \exists X_\epsilon > 0 : \\ \langle X_1, \ldots, X_n, X_\epsilon\rangle \in \gamma(P)\end{array}$$

– Minimal representations must be adapted to X_ϵ

– The algorithms for the closed case can be adapted easily to the open case [5]

References

[5] Roberto Bagnara, Elisa Ricci, Enea Zaffanella, Patricia M. Hill. "Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library". SAS 2002: 213–229

Polyhedral libraries

The *Parma* library for polyhedral static analysis

- The most recent library is PPL, The Parma Polyhedral Library, Roberto Bagnara, University of Parma, Italy
- `http://www.cs.unipr.it/ppl/`
- The Parma Polyhedra Library is (to cite the authors):
  - “user friendly (you write x + 2*y + 5*z <= 7 when you mean it);
  - “fully dynamic (available virtual memory is the only limitation to the dimension of anything);

  - “portable (written in standard C++ and following all other available standards);
  - “exception-safe (never leaks resources or leaves invalid object fragments around);
  - “efficient (and we hope to make it even more so);
  - “thoroughly documented (perhaps not literate programming but close enough);
  - “free software (distributed under the terms of the GNU General Public License).”

The *New Polka* library for polyhedral static analysis

- **Polka**, Nicolas Halbwachs, Verimag, Grenoble, France (first available library)
- **New Polka**, Bertrand Jeannet, Irisa, Rennes, France (its successor)
  `http://www.irisa.fr/prive/bjeannet/newpolka.html`
- Programmed in NASI C (whence usable in C, C++ and OCaml)
- 64 bits and multiprecision integers

- The implemented operations are
  - creation of polyhedra from constraints or generators, including strict inequalities
  - intersection
  - convex hull
  - image and pre-image by a linear transformation
  - widening, . . .
- The OCaml interface offers input and output of constraints, matrices and polyhedra as well as a polyhedra desk calculator usable at OCaml top level
- The library is available at:
  http://www.irisa.fr/prive/bjeannet/archives/polka/

Implementation of the polyhedral analysis

Interface with the New Polka Library

- Data types:

dimsup Datatype

```
type dimsup = {
  pos: int;
  nbdims: int;
}
```

Data-type for insertion and deletion of columns in vectors, matrices, and polyhedra.

- Initialization and finalization functions:

initialize : bool -> int -> int -> unit Function

`initialize strict maxdims maxrows` initializes internal data-structures and global variables of the library:

- `strict` indicates wether strict inequalities are enabled or not;
- `maxdims` is the maximum number of dimensions allowed in polyhedra; the maximum number of columns allowed in vectors and matrices is thus equal to this number plus `polka_dec` (see below);
- `maxrows` is the maximum number of rows or vectors allowed in matrices.

Set variables `strict` and `dec` (see below).

finalize : unit -> unit Function

Free internal data-structure used in the libary.

– Vector:

`t` Datatype

Abstract datatype for vectors.

`make : int -> t` Function

`make size`

Return a vector of size `size` with all coefficient initialized to 0. `size=0` is accepted.

`set : t -> int -> int -> unit` Function

`set vec index val`

Store the value `val` in the corresponding coefficient of the vector.

– Polyhedra:

t : the type of polyhedra Datatype

- Constructors for OCaml polyhedra:

`empty : int -> t` Function

`universe : int -> t` Function

Return respectively the empty and the universe polyhedron of the given dimension.

`minimize : t -> unit` Function

Minimizes in place the polyhedron.

- Predicates on OCaml polyhedra:

`is_included_in : t -> t -> bool` Function

`is_equal : t -> t -> bool` Function

Test of inclusion and equality of polyhedra.

- Change of dimension of OCaml polyhedra:

`add_dims_and_embed_multi : t -> Polka.dimsup array -> t` Function

add_dims_and_embed_multi (po, dimsup)

Adds new dimensions in the polyhedron *po*, according to the array *tab* of size *size*, and embed it in the new space. Preserves the minimality of the polyhedron.

`del_dims_multi : t -> Polka.dimsup array -> t` Function

del_dims_multi (po, dimsup)

This function projects the given polyhedron onto the $po->dim-dimsup$ first dimensions, and eliminates the last coefficients. Minimality is lost. The parameter may be minimzed in order to get its generators.

- Intersection and convex hull of OCaml polyhedra:

`inter : t -> t -> t` Function

`add_constraint : t -> Vector.t -> t` Function

`union : t -> t -> t` Function

- Linear transformation on OCaml polyhedra:

`assign_var : t -> int -> Vector.t -> t` Function

Same as C function `poly_assign_variable`.

`substitute_var : t -> int -> Vector.t -> t` Function

Same as C function `poly_substitute_variable`.

- Widening operator on OCaml polyhedra:

widening : t -> t -> t — Function

- Input and output of OCaml polyhedra:

This functions use the pretty input/output facilities described in module Polka.

print_constraints : Format.formatter -> t -> unit — Function

Print '"empty"' if the polyhedron is empty, the constraints of the polyhedron if they are available, '"constraints not available"' otherwise.

- Compilation of the New Polka library:

The library can be implemented with various representations of integer (32, 64 bits, GMP*...).

New Polka should be compiled with GMP in order to avoid overflows as in the example:
y := 0; if (y > (1073741823 - z)) then y := y - z else y := y + z fi;;

GMP: GNU arbitrary precision arithmetic for signed integers, rational numbers and floating point numbers. See http://www.swox.com/gmp/

The polyhedral abstract environment domain

```
(* aenv.ml *)
open Linear_Syntax
open Variables
type lattice = BOT | TOP
type t = NULL of lattice (* if no variable, dimension = 0 *)
  | POLY of Poly.t   (* must be of dimension > 0          *)
exception PolyError of string
(* relational library initialization *)
let init () = (Polka.initialize false 10000 100; Polka.strict := false)
(* relational library exit (* print statistics *) *)
let quit () = Polka.finalize ()
(* infimum *)
let bot () = match (number_of_variables ()) with
  | 0 -> NULL BOT
```



```
  | n -> POLY (Poly.empty n) (* 1 <= n <= polka_maxcolumns-polka_dec *)
(* check for infimum *)
let is_bot r = match r with
  | NULL BOT -> true
  | NULL TOP -> false
  | POLY p -> (Poly.is_equal p (Poly.empty (number_of_variables ())))
(* uninitialization *)
let initerr () = match (number_of_variables ()) with
  | 0 -> NULL TOP
  | n -> POLY (Poly.universe n)
(* supremum *)
let top () =  match (number_of_variables ()) with
  | 0 -> NULL TOP
  | n -> POLY (Poly.universe n)
(* least upper bound *)
let ljoin l1 l2 = match (l1, l2) with
  | BOT, _ -> l2
  | _, BOT -> l1
```

```
  | _, _   -> TOP
let join r1 r2 = match (r1, r2) with
  | NULL l1, NULL l2 -> (NULL (ljoin l1 l2))
  | POLY p1, POLY p2 -> (POLY (Poly.union p1 p2))
  | _, _ -> raise (PolyError "join")
(* greatest lower bound *)
let lmeet l1 l2 = match (l1, l2) with
  | TOP, _ -> l2
  | _, TOP -> l1
  | _, _   -> BOT
let meet r1 r2 = match (r1, r2) with
  | NULL l1, NULL l2 -> (NULL (lmeet l1 l2))
  | POLY p1, POLY p2 -> (POLY (Poly.inter p1 p2))
  | _, _ -> raise (PolyError "meet")
(* approximation ordering *)
let lleq l1 l2 = match (l1, l2) with
  | BOT, _   -> true
  | _, TOP   -> true
```



```
  | TOP, BOT -> false
let leq r1 r2 = match (r1, r2) with
  | NULL l1, NULL l2 -> (lleq l1 l2)
  | POLY p1, POLY p2 -> (Poly.is_included_in p1 p2)
  | _, _ -> raise (PolyError "leq")
(* equality *)
let eq r1 r2 = match (r1, r2) with
  | NULL l1, NULL l2 -> (l1 = l2)
  | POLY p1, POLY p2 -> (Poly.is_equal p1 p2)
  | _, _ -> raise (PolyError "eq")
(* printing *)
let print r = match r with
  | NULL BOT -> (print_string "{ _|_ }\n")
  | NULL TOP -> (print_string "{ T }\n")
  | POLY p ->
    (Poly.minimize p; (* to get the constraints and generators of p *)
     Poly.print_constraints string_of_variable Format.std_formatter p;
     Format.pp_print_newline Format.std_formatter ())
```



```
(* convert a0.v0+...+an-1.vn-1+an where n = (number_of_variables ()) *)
(* into vector [1,an,a0,...,an-1].                                   *)
let vector_of_lin_expr a =
  let v = Vector.make ((number_of_variables ()) + 2) in
    (Vector.set v 0 1;
     Vector.set v 1 (a.(number_of_variables ()));
     for i = 0 to ((number_of_variables ()) - 1) do
       Vector.set v (i+2) a.(i)
     done;
     (*
     Vector._print v;
     Vector.print_constraint string_of_variable Format.std_formatter v;
     Format.pp_print_newline Format.std_formatter ();
     *)
     v)
(* f_ASSIGN x f r =  {e[x <- i] | e in r /\ i in f({e}) cap I } *)
let f_ASSIGN x f r =
  match r with
```



```
  | NULL _ -> r
  | POLY p -> (match f with
  | RANDOM_AEXP ->
      let d = [|{ Polka.pos = x; Polka.nbdims = 1 }|] in
       (POLY (Poly.add_dims_and_embed_multi (Poly.del_dims_multi p d) d))
  | LINEAR_AEXP a ->
         (POLY (Poly.assign_var p x (vector_of_lin_expr a))))
(* f_LGE a r = {e in r | a0.v0+...+an-1.vn-1+an >= 0} *)
let f_LGE a r =
  match r with
  | NULL _ -> r
  | POLY p -> POLY (Poly.add_constraint p (vector_of_lin_expr a))
(* f_LEQ a r = {e in r | a0.v0+...+an-1.vn-1+an = 0} *)
let minus = Array.map (fun x -> (- x))
let f_LEQ a r = meet (f_LGE a r) (f_LGE (minus a) r)
(* widening *)
let widen r1 r2 = match (r1, r2) with
  | NULL l1, NULL l2 -> (NULL (ljoin l1 l2))
```

```
  | POLY p1, POLY p2 -> (POLY (Poly.widening p1 p2))
  | _, _ -> raise (PolyError "widen")
(* narrowing *)
(* let narrow a b = a (* does not ensure termination *) *)
let narrow a b = b (* less precise but does ensure termination *)
```


Typescript examples of affine inequality analyses

Top element of the lattice:

```
Generic-FW-REL-Abstract-Interpreter % ./a.out
skip;;
** Precondition:
{ T }
** Postcondition:
{ T }
```


Bottom element of the lattice:

```
Generic-FW-REL-Abstract-Interpreter % ./a.out
x := 1;
while (x > 0) do
   x := x + 1000000
od;;
** Linearized program:
  x := 0.x + 1;
  while 1.x + -1 >= 0 do
      x := 1.x + 1000000
  od {(-1.x + -1 >= 0 | -1.x + 0 = 0)}
** Precondition:
{1>=0}
** Postcondition:
empty(1)
```



```
Generic-FW-REL-Abstract-Interpreter % ./a.out
../Examples/example36.sil
** Program:
  x := ?; y := ?;
  if (x = y) then
      x := 0; y := 200
  else {((x < y) | (y < x))}
      x := 20; y := 0
  fi
** Linearized program:
  x := ?; y := ?;
  if -1.x + 1.y + 0 = 0 then
      x := 0.x + 0.y + 0;
      y := 0.x + 0.y + 200
  else {(-1.x + 1.y + -1 >= 0 | 1.x + -1.y + -1 >= 0)}
      x := 0.x + 0.y + 20;
      y := 0.x + 0.y + 0
  fi
```

```
** Precondition:
{1>=0}
** Postcondition:
{10x+y=200,y<=200,y>=0}
```


Bibliography

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY, U.S.A.

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY, U.S.A.

THE END

My MIT web site is http://www.mit.edu/~cousot/

The course web site is http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/.