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
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 \( \text{lfp}_f \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g \) is the \( \sqsubseteq \)-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

- \( \lambda g \cdot g \circ g \in (L \xrightarrow{\text{me}} L) \Rightarrow (L \xrightarrow{\text{me}} L) \) is monotonic. Indeed if \( g_1 \sqsubseteq g_2 \) then by def. of a pointwise ordering \( g_1 \circ g_2 \sqsubseteq g_1 \circ g_2 \) and by monotony of \( g_1, g_1 \circ g_1 \sqsubseteq g_1 \circ g_2 \) so by transitivity \( g_1 \circ g_1 \sqsubseteq g_2 \circ g_2 \) proving that \( \lambda g \cdot g \circ g \in (L \xrightarrow{\text{me}} L) \xrightarrow{\text{me}} (L \xrightarrow{\text{me}} L) \)

If \( f \in L \xrightarrow{\text{me}} L \) is monotone and extensive then \( f \sqsubseteq f \circ f \) so \( f \) is a prefixpoint of \( \lambda g \cdot g \circ g \) considered as a function of \( (L \xrightarrow{\text{me}} L) \xrightarrow{\text{me}} (L \xrightarrow{\text{me}} L) \)

It follows by Knaster-Tarski on cpos that \( \text{lfp}_f \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g \) does exists.

If we consider the transfinite iterates \( \langle g^\beta, \beta \in \Omega \rangle \) of \( g \in L \xrightarrow{\text{me}} L \cdot g \circ g \) from \( f \), the are all monotone and extensive since \( g^\beta = f \in L \xrightarrow{\text{me}} L \), if \( g^\beta \in L \xrightarrow{\text{me}} L \) then \( g^{\beta+1} = g^\beta \circ g^\beta \in L \xrightarrow{\text{me}} L \) as shown above and if \( \forall \beta < \lambda : g^\beta \in L \xrightarrow{\text{me}} L \) implies \( g^\lambda = \bigcup_{\beta < \lambda} g^\beta \) from limit ordinal so in particular \( \text{lfp}_f \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g = g^\epsilon \) where \( \epsilon \) is the rank of the iterates is certainly monotone and extensive

Moreover, by the fixpoint property, \( g^\epsilon = g^\epsilon \circ g^\epsilon \) proving \( \text{lfp}_f \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g \) idempotent whence a closure operator. Since the iterates are increasing it is also greater than or equal to \( f \)

If \( \rho \) is another closure operator on \( L \) greater than or equal to \( f \) we have - \( f \sqsubseteq \rho \) and \( \rho = \rho \circ \rho \) so by Knaster-Tarski \( \text{lfp}_f \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g = \bigcup \{ g \in L \xrightarrow{\text{me}} L \mid f \sqsubseteq g \circ g \} \sqsubseteq \rho \) by def. glbs

**Corollary.** Let \( \langle L, \sqsubseteq, \bot, \sqcup, \sqcap \rangle \) be a complete lattice. The lub of a set \( F \) of upper closure operators in the complete lattice of closure operators on \( L \) is

\[ \text{lub}_F \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g \]

**Proof.** Let \( \text{lub} F \) be this lub. We have \( \bigcup F \sqsubseteq \text{lub} F \) and, because \( \bigcup F \) is monotonic and extensive, \( \text{lub} F \) is the least closure operator \( \sqsubseteq \)–greater than of equal to \( \bigcup F \), whence, by the previous theorem, \( \text{lfp}_F \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g \)
The lub of closure operators (II)

**Theorem.** If \( \langle L, \subseteq, \perp, \sqcup \rangle \) is a cpo and \( f \in L \xrightarrow{\text{me}} L \) is monotone and extensive then \( \text{lfp}_f \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g = \lambda x : \text{lfp}_f g \)

**Proof.** Define \( g = \text{lfp}_f \lambda g' \in L \xrightarrow{\text{me}} L \cdot g' \circ g' \). We just showed that \( g \) is the \( f \)-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 \( \langle L, \subseteq, \perp, \sqcup \rangle \) is a cpo and \( f \) is monotone, \( \text{lfp}_f f \) does exists, whence \( \lambda x : \text{lfp}_f f \) is well-defined.

- Define \( h \overset{\text{def}}{=} \text{lfp}_f f \). \( h(x) \) 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 \( f \sqsubseteq h \). It follows that \( h \) is a closure operator on \( L \) which is greater than or equal to \( f \), proving that \( g \sqsubseteq h \).

- Let \( \langle h^\delta, \delta \in \mathbb{O} \rangle \) be the iterates of \( \text{lfp}_f f \) and \( \langle g^\delta, \delta \in \mathbb{O} \rangle \) be the iterates of \( \text{lfp}_f \lambda g' \in L \xrightarrow{\text{me}} L \cdot g \circ g \). Let us prove, by transfinite induction, that \( \forall \delta \in \mathbb{O} : h^\delta \sqsubseteq g^\delta(x) \).
  - \( h^\delta = x \sqsubseteq f(x) = g^0(x) \)
  - If \( h^\delta \sqsubseteq g^\delta(x) \) then \( g^\delta(h^\delta) \sqsubseteq g^\delta(g^\delta(x)) \) since \( g^\delta \) is monotone. But \( f \sqsubseteq g^\delta \) 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 \( \lambda \), if \( \forall \beta < \lambda : h^\beta \sqsubseteq g^\beta(x) \) then \( h^\lambda = \bigcup_{\beta < \lambda} h^\beta \sqsubseteq \bigcup_{\beta < \lambda} g^\beta(x) = (\bigcup_{\beta < \lambda} g^\beta)(x) = g^\lambda(x) \), by def. of the iterates, existence of the lubs in the cpo and def. lubs.

- Let \( \epsilon \) and \( \epsilon' \) be the rank of the respective iterates. Then \( h(x) = \text{lfp}_f g = h^\epsilon = h^{\max(\epsilon, \epsilon')}(x) = g^\epsilon(x) = (\text{lfp}_f \lambda g' \in L \xrightarrow{\text{me}} L \cdot g' \circ g')(x) = g(x) \) so that \( h \sqsubseteq g \).
- By antisymmetry, we conclude that \( h = g \).

**Corollary.** Let \( \langle L, \subseteq, \perp, \sqcup, \sqcap \rangle \) be a complete lattice. The lub of a set \( F \) of upper closure operators in the complete lattice of closure operators on \( L \) is \( \lambda x : \text{lfp}_x \bigcup F \).

**Proof.** The lub has been shown (on page 7) to be \( \text{lfp}_x \lambda g \in L \xrightarrow{\text{me}} L \cdot g \circ g = \lambda x : \text{lfp}_x \bigcup F \).

Iterative reduced product
Union of abstract domains

**Theorem.** If \( \langle L, \subseteq, \bot, \top, \sqcup, \sqcap \rangle \) and \( \langle M_i, \leq \rangle \) are complete lattices and \( \forall i \in \Delta : \langle L, \subseteq \rangle \xrightarrow{\gamma_i} \langle M_i, \leq \rangle \)

\[
\bigwedge \gamma_i \quad \text{then} \quad \langle L, \subseteq \rangle \xrightarrow{T} \langle M, \leq \rangle
\]

**Proof.** For all \( x \in L \) and \( y \in M \), we have

\[
\left( \bigvee_{i \in \Delta} \alpha_i(x) \right) \leq y \\
\iff \left( \bigvee_{i \in \Delta} \alpha_i(x) \right) \leq y \\
\iff \forall i \in \Delta : \alpha_i(x) \leq y \\
\iff \forall i \in \Delta : x \subseteq \gamma_i(y)
\]

\[\text{Galois connection}\]

\[\text{def. glb}\]

\[\text{pointwise def. } \sqcap\]

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

Cartesian product of abstract domains

**Theorem.** If \( \langle L, \subseteq, \bot, \top, \sqcup, \sqcap \rangle \) is a complete lattice and \( \langle M_i, \leq \rangle \) is a family of posets then \( \forall i \in \Delta : \langle L, \subseteq \rangle \xrightarrow{\gamma_i} \langle M_i, \leq \rangle \)

\[
\bigwedge_{i \in \Delta} \gamma_i(a_i) \\
\iff \forall i \in \Delta : \alpha_i(x) \leq y_i
\]

\[\text{def. lub}\]

\[\text{pointwise def. } \leq\]

- 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 \( \langle L, \sqsubseteq \rangle \leftrightarrow_{\alpha} \langle A, \leq \rangle \) where \( A, \leq, 0, 1, \lor, \land \) is a complete lattice. Define

\[ \rho(a) \overset{\text{def}}{=} \land \{a' \in A \mid \gamma(a) \sqsubseteq \gamma(a')\} \]

then \( \rho \) is a lower closure operator and

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

**Proof.** \( \rho \) 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 \( \land \).

- 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 \( \rho(a) = \land \{a' \mid \gamma(a) \sqsubseteq \gamma(a')\} \subseteq \land \{b' \mid \gamma(b) \sqsubseteq \gamma(b')\} = \rho(b) \).

- For idempotence, we have

\[ \rho(\rho(a)) \]

\[ = \land \{a' \in A \mid \gamma(\rho(a)) \sqsubseteq \gamma(a')\} \]

\[ = \land \{a' \in A \mid \gamma(\land \{a'' \in A \mid \gamma(a) \sqsubseteq \gamma(a'')\}) \sqsubseteq \gamma(a')\} \]

\[ = \land \{a' \in A \mid \land \{\gamma(a'' \in A \mid \gamma(a) \sqsubseteq \gamma(a'')\} \sqsubseteq \gamma(a')\} \]

\[ = \land \{a' \in A \mid \gamma(a) \sqsubseteq \gamma(a')\} \quad \text{\textit{(since}} \gamma(a) = \land \{\gamma(a'') \in A \mid \gamma(a) \sqsubseteq \gamma(a'')\}) \] by reflexivity and def. glb

\[ \rho(a) \]

\[ \overset{\text{def. } \rho}{=} \rho(a) \]

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

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

\[ \rho(\rho(a)) \sqsubseteq y \]

\[ \Rightarrow \land \{a' \in A \mid \gamma(\rho(a)) \sqsubseteq \gamma(a')\} \subseteq y \quad \overset{\text{def. } \rho}{=} \]

- 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.** for all \( x \in L \):

\[ \gamma \circ \rho \circ \alpha \overset{\text{def. } \rho}{=} \gamma(\land \{a' \mid \gamma(\alpha(x)) \sqsubseteq \gamma(a')\}) \]

\[ = \land \{\gamma(a') \mid \gamma(\alpha(x)) \sqsubseteq \gamma(a')\} \]

\[ = \gamma(\alpha(x)) \]

\[ \overset{\text{def. } \rho}{=} \gamma \circ \alpha \overset{\text{def. } \rho}{=} \gamma \circ \rho \circ \alpha \]

\[ \overset{\text{since } \gamma \circ \alpha = \gamma \circ \rho \circ \alpha}{=} \gamma \circ \alpha \overset{\text{def. } \rho}{=} \gamma \]

\[ \overset{\text{\textit{Galoi connection}}}{=} \gamma \circ \rho \circ \alpha \overset{\text{def. } \rho}{=} \gamma \]

\[ \overset{\text{\textit{and } \rho}{=} \gamma \circ \alpha \overset{\text{def. } \rho}{=} \gamma \]

- Moreover \( \rho \) is a lower closure operator on \( \langle \prod_{i \in I} A_{i}, \sqsubseteq \rangle \) so \( \rho \) is reductive \( \rho \sqsubseteq \gamma \) whence by monotony \( \gamma \circ \rho \sqsubseteq \gamma \). By antisymmetry, \( \gamma \circ \rho = \gamma \).
The reduced product

**Theorem.** Assume that \( \langle L, \sqsubseteq \rangle \) is a poset, \( \langle A_i, \leq_i, 0_i, 1_i, \forall_i, \land_i, \rangle, \ i \in \Delta \) are complete lattices such that \( \forall i \in \Delta : \langle L, \sqsubseteq \rangle \xleftrightarrow{\gamma_i} \langle A_i, \leq_i \rangle. \) Define \( \leq_{\Delta} \) componentwise in terms of the \( \leq_i, i \in \Delta. \) Let \( \gamma = \lambda a. \prod_{i \in \Delta} \gamma_i(a_i) \) and \( \alpha = \lambda x. \prod_{i \in \Delta} \alpha_i(x) \) so that \( \langle L, \sqsubseteq \rangle \xleftrightarrow{\gamma} \langle \prod_{i \in \Delta} A_i, \leq_{\Delta} \rangle. \) Now let \( \rho = \lambda a. \prod\{a' | \gamma(a) \subseteq \gamma(a')\} \) so that \( \langle L, \sqsubseteq \rangle \xleftrightarrow{\gamma \circ \rho, \alpha} \langle \rho(\prod_{i \in \Delta} A_i), \leq_{\Delta} \rangle. \) 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 than the \( \langle A_i, \leq_i \rangle \) in that \( \gamma \circ (\rho \circ \prod_{i \in \Delta} a_i) \leq_{\Delta} \gamma \circ \alpha_i \)

\[
= \gamma(\rho(\prod_{i \in \Delta} a_i(x))) \quad \text{(def. \( \circ, \prod_{i \in \Delta} \))}
\]

\[
= \bigwedge\{\gamma(a') | \gamma(\prod_{i \in \Delta} a_i(x)) \subseteq \gamma(a')\} \quad \text{(\( \gamma \) preserves existing meets)}
\]

\[
= \gamma(\prod_{i \in \Delta} a_i(x)) \quad \text{(choosing \( a' = \prod_{i \in \Delta} a_i(x) \) and def. glb)}
\]

\[
= \bigwedge_{k \in \Delta} \gamma_k(\prod_{i \in \Delta} a_i(x)_k) \quad \text{(def. \( \gamma \))}
\]

\[
= \bigwedge_{k \in \Delta} \gamma_k(a(x)) \quad \text{(def. index selection)}
\]

---

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 “unreadable 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
- \( \langle L, \sqsubseteq, \sqcup, \top, \sqcap, \sqcap \rangle \) be a complete lattice
- Let \( \langle \Delta, \leq \rangle \) be a totally ordered set of indices\(^1\)
- \( \langle A_i, \leq_i, 0_i, 1_i, \forall_i, \wedge_i \rangle, i \in \Delta \) be complete lattices
- \( \langle L, \sqsubseteq \rangle \overset{\gamma_i}{\iff} \langle A_i, \sqsubseteq_i \rangle \) for all \( i \in \Delta \)

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

\(^1\) naming abstract domains, in practice \( \Delta \) is finite.

**Theorem.** \( \rho^* \) is the glb of the \( \{ \rho_i \overset{\leq \Delta}{\iff} i < j \} \) 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
\[
\gamma(\vec{a}) = \bigwedge_{k \in \Delta \setminus \{i,j\}} \gamma_k(\vec{a}_k) \cap \gamma_j(\vec{a}_j) \quad \text{(def. } \gamma) \\
\rho_{ij}(\vec{a}) = \bigwedge_{k \in \Delta \setminus \{i,j\}} \rho_{kj}(\gamma_k(\vec{a}_k) \cap \gamma_j(\vec{a}_j)) \quad \text{since } \gamma_i \circ \rho_i \text{ and } \gamma_j \circ \rho_j \text{ are extensive}
\]

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

For all \( \vec{a} \in \prod_{i \in \Delta} A_i \), we have \( \rho^*(\vec{a}) = \bigwedge_{i,j \in \Delta; i < j} \rho_{ij} \) whence \( \rho^*(\vec{a}) \subseteq \vec{a} \). So by monotony and def. \( \gamma \circ \rho \subseteq \gamma \).

We conclude, using the theorem on page 20, that \( \gamma = \gamma \circ \rho \subseteq \gamma \circ \rho^* \) by antisymmetry.
The reduced product (iterative form)

**Theorem.** \( \langle \rho^*(\prod_{i \in \Delta} A_i), \leq \Delta \rangle \) is the reduced product of the \( \langle A_i, \leq_i \rangle, i \in \Delta \).

**Proof.** Let us first prove that we have \( \langle L, \sqsubseteq \rangle \xrightarrow{\eta_{\rho^* \alpha}} \langle \rho(\prod_{i \in \Delta} A_i), \leq \Delta \rangle \). Indeed for all \( x \in L \) and \( y \in \rho(\prod_{i \in \Delta} A_i) \), we have

\[
\rho^* \circ \alpha(x) \leq \Delta y \implies \gamma \circ \rho^* \circ \alpha(x) \subseteq \gamma(y) \\
\implies \gamma \circ \alpha(x) \subseteq \gamma(y) \quad \text{[since } \gamma \circ \rho^* \text{]}
\]

\[
x \subseteq \gamma(y) \quad \text{[since } \gamma \circ \alpha \text{ is extensive]}
\]

\[
\alpha(x) \leq \Delta y \quad \text{[Galois connection]}
\]

\[
\rho^* \circ \alpha(x) \leq \Delta y
\]

We have \( \langle L, \sqsubseteq \rangle \xrightarrow{\eta_{\rho^* \alpha}} \langle \rho(\prod_{i \in \Delta} A_i), \leq \Delta \rangle \) and \( \langle L, \sqsubseteq \rangle \xrightarrow{\eta_{\rho \circ \alpha}} \langle \rho(\prod_{i \in \Delta} A_i), \leq \Delta \rangle \) whence \( \rho \circ \alpha = \rho^* \circ \alpha \) 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 \( \langle \rho^*(\prod_{i \in \Delta} A_i), \leq \Delta \rangle \). \( \square \)

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 \( \rho \) (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 \( \rho \) is a difficult task when \( |\Delta| \) is large
- The design and implementation of \( \rho \) has to be entirely redone when a new abstract domain is added to the list \( \Delta \)
The reduced product of three abstract domains or more

- We can consider instead the iterative reduction $\langle \rho^*(\prod_{i \in A} A_i), \leq \Delta \rangle^2$
- We then consider the reductions, two by two:

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

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

\[\text{Indeed this makes not difference when } |\Delta| \leq 2.\]

- The advantage of this approach are that:
  - The pairwise reductions $\rho_{ij}, i, j \in \Delta, i < j$ are much simpler to design and implement than the global reduction $\rho^3$
  - The iterative implementation is equally precise (recall $\gamma \circ \rho = \gamma \circ \rho^*$)
  - The generic forward abstract interpreter with reduced product

\[\text{The addition of a new abstract domain only requires}
\]
\[\text{- the design and implementation of its reduction with}
\]
\[\text{the existing ones,}
\]
\[\text{- without any modification of the existing reductions}
\]

\[\text{which involves the evaluation of an hidden fixpoint anyway.}
\]
\[\text{replacing any abstract information } d \in \prod_{i \in \Delta} A_i \text{ appearing during the analysis by } \rho^*(d)
\]
\[\text{in which case the exact reduction } \rho \text{ was certainly quite complex if not impossible to compute.}
\]
Implementation of the ternary iterated reduction

1 (* red123.ml *)
2 open Avalues1
3 open Avalues2
4 open Avalues3
5 val reduce : (Avalues1.t * Avalues2.t * Avalues3.t)
6 -> (Avalues1.t * Avalues2.t * Avalues3.t)
7 (* red123.ml *)
8 open Red12
9 open Red23
10 open Red13
11 open Avalues1
12 open Avalues2
13 open Avalues3

The parity and initialization and simple sign reduction

- The main reduction is \( \text{ODD} \land \text{EVEN} \rightarrow \text{BOT} \)
- The abstract values implementation is hidden, whence must be accessed through abstract primitives operations, such as:
  - \( (\text{Avalues1.f\_NAT "0"}) = \text{EVEN} \)
  - \( (\text{Avalues1.f\_NAT "1"}) = \text{ODD} \)
  - \( (\text{Avalues2.f\_NAT "0"}) = \text{ZERO} \)
  - \( (\text{Avalues2.f\_NAT "2"}) = \text{POS} \)
  - ...

Note that we may have to include a narrowing to ensure termination of the iteration.
open Avalues1 (* avalues.ml of Parity *)
35  (* \gamma(BOT) = \{0\} *)
36  (* \gamma(ODD) = \{1, 2, 3, \ldots\} *)
37  (* \gamma(EVEN) = \{0, 2, 4, \ldots\} *)
38  (* \gamma(TOP) = \{0, 2, 4, \ldots\} *)
39  open Avalues2 (* avalues.ml of initialization and simple sign *)
40  (* \gamma(BOT) = \{0\} *)
41  (* \gamma(ODD) = \{1\} *)
42  (* \gamma(EVEN) = \{0\} *)
43  (* \gamma(TOP) = \{0\} *)
44  let reduce (p, i) =
45  if ((Avalues1.eq p (Avalues1.bot ()))) then
46  ((Avalues2.eq i (Avalues2.bot ())))
47  else if ((Avalues1.eq p (Avalues1.f_NAT "1"))) then
48  ((Avalues2.eq i (Avalues2.f_NAT "0")))
49  else ((Avalues2.eq i (Avalues2.f_NAT "0")))
50  then ((Avalues1.bot ()), (Avalues2.bot ()))
51  else if ((Avalues1.eq p (Avalues1.f_NAT "0"))) then
52  ((Avalues2.eq i (Avalues2.f_NAT "0")))
53  then ((Avalues1.bot ()), (Avalues2.bot ()))
54  else if ((Avalues2.eq i (Avalues2.f_NAT "0"))) then
55  then ((Avalues1.f_NAT "0"), i)
56  else (p, i)

The parity and intervals reduction

open Avalues1 (* avalues.ml of Parity *)
59  (* \gamma(BOT) = \{0\} *)
60  (* \gamma(ODD) = \{1\} *)
61  (* \gamma(EVEN) = \{0\} *)
62  open Avalues2 (* avalues.ml of intervals *)
63  (* (a, b) = [a, b] *)
64  let reduce (p, i) =
65  if ((Avalues1.eq p (Avalues1.bot ()))) then
66  ((Avalues1.eq p (Avalues1.bot ())))
67  else if ((Avalues1.eq p (Avalues1.f_NAT "1"))) then
68  ((Avalues1.eq p (Avalues1.f_NAT "0")))
69  else ((Avalues1.eq p (Avalues1.f_NAT "0")))
70  then ((Avalues1.bot ()), (Avalues2.bot ()))
71  else if ((Avalues1.eq p (Avalues1.f_NAT "1"))) then
72  then (p, (reduce_odd i))
73  else ((Avalues1.eq p (Avalues1.f_NAT "0"))) then
74  then (p, (reduce_even i))
75  else ((Avalues2.parity i) = 0) then
76  then ((Avalues2.parity i) = 1) then
77  if ((Avalues2.parity i) = 1) then
78  then ((Avalues1.f_NAT "1"), i)
79  else (p, i)

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

(* avalues.ml *)

(* reductions by parity *)
val reduce_even : t -> t
val reduce_odd : t -> t
In the parity abstract domain, the parity can be improved for constant intervals

(* 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)))

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

The parity information is sent in the form of a constant.
Reduction of intervals with initialization and simple sign

(* red-iss-intervals12.ml *)

open Avalues1 (* ... {_O_(a), _O_(i)} *)

let gamma12 a =

  if (Avalues1.eq a (Avalues1.bot ()))
  then ... (Avalues1.f_NAT "0")
–page –

(* gamma.ml *)

(* information on simple sign = -1:<0, 0:=0, ... then 1
else 2
...
–page –

(* avales.mli *)

... (* reduction with initialization and simple sign *)

... (* information on simple sign = -1:<0, 0:=0, 1:>0, 2:TOP *)

val sign : t -> int
...

(* avales.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

113 (* red-Errors-Intervals12.ml *)
114 open Avalues1 (* avalues.ml of Errors *)
115 (* gamma(HER) = [min_int,max_int] U \{0\} *)
116 (* gamma(AER) = [min_int,max_int] U \{0\} *)
117 (* gamma(IER) = [min_int,max_int] U \{0\} *)
118 (* gamma(ERR) = [min_int,max_int] U \{0\} *)
119 open Avalues2 (* avalues.ml of Intervals *)
120 (* gamma(a,b) = [a,b] U \{0\} *)
121 (* min_int <= a <= b <= max_int *)
122 (* a = max_int > min_int = b *)
123 let reduce (a,b) = (a,b)

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

The ternary reduced product

125 (* avalues123.ml *)
126 open Avalues1
127 open Avalues2
128 open Avalues3
129 open Red123
130 (* reduced product *)
131 (*)
132 (* ABSTRACT VALUES *)
133 (*)
134 type t = Avalues1.t * Avalues2.t * Avalues3.t
135 (* gamma(a,b,c) = Avalues1.gamma(a) \ Avalues2.gamma(b) \ *)
136 (* Avalues3.gamma(c) *)
137 (* infimum: bot () = alpha({}) *)
138 let bot () = reduce ((Avalues1.bot ()), (Avalues2.bot ()),
139 (Avalues3.bot ()))

140 (* isbotempty () = gamma(bot ()) = {} *)
141 let isbotempty () = (Avalues1.isbotempty ()) ||
142 (Avalues2.isbotempty ()) || (Avalues3.isbotempty ())
143 (* uninitialization: initerr () = alpha({0}) *)
144 let initerr () = reduce ((Avalues1.initerr ()), (Avalues2.initerr ()),
145 (Avalues3.initerr ()))
146 (* supremum: top () = alpha({0}, [0,\inf]) *)
147 let top () = reduce (Avalues1.top (), Avalues2.top (), Avalues3.top ())
148 (* least upper bound join: p q = alpha(gamma(p) U gamma(q)) *)
149 let join (v,w,t) (x,y,u) = reduce ((Avalues1.join v x),
150 (Avalues2.join w y), (Avalues3.join t u))
151 (* greatest lower bound meet p q = alpha(gamma(p) cap gamma(q)) *)
152 let meet (v,w,t) (x,y,u) = reduce ((Avalues1.meet v x),
153 (Avalues2.meet w y), (Avalues3.meet t u))
154 (* approximation ordering: leq p q = gamma(p) subseteqgamma(q) *)
155 let leq (v,w,t) (x,y,u) = (Avalues1.leq v x) & (Avalues2.leq w y)
156 & (Avalues3.leq t u)
157 (* equality: eq p q = gamma(p) = gamma(q) *)

158 let eq (v,w,t) (x,y,u) = (Avalues1.eq v x) & (Avalues2.eq w y)
159 & (Avalues3.eq t u)
160 (* included in errors?: in_errors p = gamma(p) subseteq \{0\} *)
161 let in_errors (x,y,z) = (Avalues1.in_errors x) ||
162 (Avalues2.in_errors y) || (Avalues3.in_errors z)
163 (* printing *)
164 let print (x,y,z) =
165 (print_string "\(; Avalues1.print x; print_string ",\);"
166 (print_string "\(; Avalues2.print y; print_string ",\);"
167 (print_string "\(; Avalues3.print z; print_string ")\))
168 (*)
169 (* ABSTRACT TRANSFORMERS *)
170 (*)
171 (* forward abstract semantics of arithmetic expressions *)
172 (* f_NAT s = alpha((machine_int_of_string s)) *)
173 let f_NAT s = reduce (Avalues1.f_NAT s, Avalues2.f_NAT s,
174 (Avalues3.f_NAT s))
175 (* f_RANDOM () = alpha([min_int, max_int]) *)
let f_RANDOM () = reduce (Avalues1.f_RANDOM (),
Avalues2.f_RANDOM (), Avalues3.f_RANDOM ()),

(* forward abstract semantics of boolean expressions *)

let f_UPLUS a = alpha((machine_binary_binarith i j) | *)

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)

(* backward abstract semantics of arithmetic expressions *)

let f_NOMINUS a = alpha((machine_unary_minus x) | x \in gamma(a))

let f_RANDOM () = reduce (Avalues1.f_RANDOM (),
Avalues2.f_RANDOM (), Avalues3.f_RANDOM ()),

(* forward abstract semantics of boolean expressions *)

let f_UPLUS a = alpha((machine_binary_binarith i j) | *)

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)
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:

(make help) : this help
(make reset) : erase all mode choices
(2) choose tracing mode:
(make trace) : tracing all
(make traceexp) : 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 interpreter
(make nr) : non-relational abstract interpreter
(3b) boolean expressions:
(make fbbool) : 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)
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 frrbool
    Forward/backward analysis of boolean expressions with reduction
    % make fhsassign
    Forward/backward analysis of assignments

Note that in the assignment to y, \((536870912 \times 2) = 1073741824\) > \(\text{max\_int} = 1073741823\). 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 \times 2);
    1:
    y := (536870912 \times 2);
    2:
    z := ((-1073741823 - 1) \times 1);
    3:
    t := ((-1073741823 - 1) \times 1073741823)
    4:
    { x:NEG; y:POS; z:NEG; t:NEG }
- **“Interval”** static analysis:

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

- Binary/pairwise reductions:

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

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

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

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

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

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

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

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

\[ a_0 x_0 + \ldots + a_{m-2} x_{m-2} \]  
\[ a_m x_m + a_{m+1} \]  

in the form:  
\[ \begin{pmatrix} a_0 & \ldots & a_m \end{pmatrix} \begin{pmatrix} x_0 \end{pmatrix} + \begin{pmatrix} a_{m+1} \end{pmatrix} \]

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:  
  \[ 2 \times X + 1 \]
  This will miss:
  \[ T := 2 \times X + 1 \]

Dynamic linear/affine expressions recognition

- In the dynamic view (as used in ASTREE), the expression is partially evaluated using information presently available to transform it into linear form. If constant propagation is used above, then \( T := 2 \times X + 1 \) yields the affine expression \( 2 \times X + 1 \). 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 $\text{Var}[P] = \{X_1, \ldots, X_k\}$, $k \in \mathbb{N}$ are defined as

\[
L ::= ? \\
| \sum_{i=1}^{k} n_i \times X_i + n_{k+1}
\]

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:

\[
\begin{align*}
\text{Faexp}\left[?\right] & \text{R} \overset{\text{def}}{=} \varnothing \\
\text{Faexp}\left[\sum_{i=1}^{k} n_i \times X_i + n_{k+1}\right] & \text{R} \overset{\text{def}}{=} \left\{ \sum_{i=1}^{k} n_i \times \rho(X_i) + n_{k+1} \mid \rho \in \text{R} \right\}
\end{align*}
\]

The linear abstraction of arithmetic expressions

We define a syntactic abstraction:

\[
\alpha : A \rightarrow L
\]

such that for all arithmetic expressions:

\[
\text{Exexp}\left[A\right] \subseteq \text{Exexp}\left[\alpha\left(A\right)\right]
\]

so that $\text{Exexp}\left[\alpha\left(A\right)\right]$ is an upper approximation of $\text{Exexp}\left[A\right]$. This proves the soundness of the linearization $\alpha$.

\[
\begin{align*}
\alpha(\langle n \rangle) & = \sum_{i=0}^{n} 0 \times X_i + n \\
\alpha(\langle X \rangle) & = \sum_{i=0}^{n} 0 \times X_i + 1 \times X_j + 0 \\
\alpha(?) & = ?
\end{align*}
\]

\[
\begin{align*}
\alpha(A + B) & = \begin{cases} \alpha(A) + \alpha(B) & \text{if } \alpha(A) + \alpha(B) \text{ is an integer} \\
\text{?} & \text{otherwise}
\end{cases} \\
\alpha(A \times B) & = \begin{cases} \alpha(A) \times \alpha(B) & \text{if } \alpha(A) \times \alpha(B) \text{ is an integer} \\
\text{?} & \text{otherwise}
\end{cases} \\
\end{align*}
\]

where $n \mod m = r$ such that $r = n + m$ and in case of overflow the result is $\text{?}$.
Definition of the forward collecting semantics of arithmetic expressions

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

\[
\text{Faexp} \in \text{Aexp} \leftrightarrow \varphi(\text{Env}[P]) \downarrow \varphi(\Pi \Omega),
\]

\[
\text{Faexp}[A] = \{v \mid \exists \rho \in R : \rho \vdash A \Rightarrow v\}.
\]

such that:

\[
\text{Faexp}[A] \left( \bigcup_{k \in S} R_k \right) = \bigcup_{k \in S} (\text{Faexp}[A] R_k) \quad \text{Faexp}[A] 0 = 0.
\]

Structural specification of the forward collecting semantics of arithmetic expressions

\[
\text{Faexp}[n] = \{n\}^6
\]

\[
\text{Faexp}[x] = \text{R}(x)
\]

where \(\text{R}(x) = \{\rho(x) \mid \rho \in R\}\)

\[
\text{Faexp}[?] = \{
\]

\[
\text{Faexp}[u A'] = \{u^c \text{Faexp}[A'] \}^C
\]

where \(u^c(V) = \{u(v) \mid v \in V\}\)

\[
\text{Faexp}[A_1 \& A_2] = \{\text{Faexp}[A_1], \text{Faexp}[A_2]\}^C
\]

where \(\text{Faexp}[F_1, F_2] = \{v_1 \& v_2 \mid \exists \rho \in R : v_1 \in F_1(\rho) \land v_2 \in F_2(\rho)\}\)

\[^6 \text{ For short, the case Faexp}[A] 0 = 0 is not recalled.\]
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

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

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: \( \alpha(\text{Faexp}[A]) \overset{\text{def}}{=} \text{Faexp}[\alpha(A)] \)

---

Proof of correctness of the linear abstraction

**Proof.** By structural induction on \( A \)

- if \( A \) is \( n \) then:
  \[
  \text{Eaeop}[\alpha(n)] R
  \]
- if \( A \) is \( \frac{a \cdot x_i + b}{c} \) then:
  \[
  \text{Eaeop}[\frac{a \cdot x_i + b}{c}] R
  \]
- if \( A \) is \( \frac{a \cdot x_i + b}{c} \) then:
  \[
  \text{Eaeop}[\frac{a \cdot x_i + b}{c}] R
  \]
- if \( A \) is \( ? \) then:
  \[
  \text{Eaeop}[\alpha(\ ? \ )] R \subseteq R(n)
  \]

(1) Recall that \( \text{Eaeop}[A] R \). The reason is that since the evaluation of \( A \) might yield \( \frac{a \cdot x_i + b}{c} \) so that this possibility must be included in the abstraction \( \alpha(A) \). Notice that to be more precise we could have distinguished \( ? \) and \( \frac{a \cdot x_i + b}{c} \) in the abstraction.
Syntax of linear boolean expressions

We define linear boolean expressions as:

\[
BL ::= BL_1 \lor BL_2 \\
    BL_1 \land BL_2 \\
    \ldots
\]

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

\begin{align*}
\text{Cbexp}[\text{true}] R & \overset{\text{def}}{=} R \\
\text{Cbexp}[\text{false}] R & \overset{\text{def}}{=} 0 \\
\text{Cbexp}[AL_1 \lor AL_2] & \overset{\text{def}}{=} \underbrace{\text{Cbexp}[AL_1], \text{Cbexp}[AL_2]}_{\in C} R \\
\text{where } & \quad \underbrace{\text{C}(F,G) R \overset{\text{def}}{=} \{ \rho \in R \mid \exists \nu_1 \in F(\{\rho\}) \cap I : \exists \nu_2 \in G(\{\rho\}) \cap I: \nu_1 \leq \nu_2 = \nu \}}_{\in C}
\end{align*}

\text{Cbexp}[BL_1 \land BL_2] R \overset{\text{def}}{=} \text{Cbexp}[BL_1] R \cap \text{Cbexp}[BL_2] R \\
\text{Cbexp}[BL_1 \mid BL_2] R \overset{\text{def}}{=} \text{Cbexp}[BL_1] R \cup \text{Cbexp}[BL_2] R

Soundness of the linearization of boolean expressions

\text{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

\[ \forall R \in \rho(\text{Env}[P]) \setminus \{0\} : \text{Cbexp}[B] R \subseteq \text{Cbexp}[\alpha(B)] R \]

Note: again the semantic abstraction \( \alpha \in (\rho(\text{Env}[P])) \mapsto \rho(\text{Env}[P])) \) is defined in terms of a syntactic transformation of boolean expressions into linear boolean expressions: \( \alpha(\text{Cbexp}[B]) \overset{\text{def}}{=} \text{Cbexp}[\alpha(B)] \)

\text{PROOF.} We proceed by structural induction on \( B \).
Syntax of linear commands and programs

$$LC ::= \begin{align*} & \text{skip} \\ & X := \text{AL} \\ & \text{if BL then LCL}_0 \text{ else } LCL_1 \text{ fi} \\ & \text{while BL do LCL}_0 \text{ od} \end{align*}$$

$$LCL ::= \begin{align*} & \text{CL} \\ & LCL \& LCL_0 \end{align*}$$

$$PL ::= \begin{align*} & LCL; \end{align*}$$

Program linearization

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

$\alpha(\text{skip}) \overset{\text{def}}{=} \text{skip}$

$\alpha(X := \text{AL}) \overset{\text{def}}{=} X := \alpha(\text{AL})$

$\alpha(\text{if } B \text{ then } LCL_t \text{ else } LCL_f \text{ fi}) \overset{\text{def}}{=} \begin{align*} & \text{if } \alpha(B) \text{ then } \alpha(LCL_t) \text{ else } \alpha(LCL_f) \text{ fi} \\ & \alpha(\text{while } B \text{ do } LCL \text{ od}) \overset{\text{def}}{=} \text{while } \alpha(B) \text{ do } \alpha(LCL) \text{ od} \end{align*}$

$\alpha(B ; LCL_0) \overset{\text{def}}{=} \alpha(B) ; \alpha(LCL_0)$

$\alpha(LCL_0 ; ;) \overset{\text{def}}{=} \alpha(LCL_0) ; ;$
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

$$\forall R \in \varphi(\mathit{Env}[P]) \setminus \{0\} : \mathit{Pcom}[P]R \subseteq \mathit{Pcom}[\alpha(P)]R$$

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

---

**Proof.** We proceed by structural induction on $P$.

Implementation of the syntactic linear abstraction
The non-initialization ($\Omega_i$) and arithmetic errors ($\Omega_a$) 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 $\ell$ $c$ to test that the label $\ell$ does appear within command $c$.

10 and lbexp =
11 | LTRUE | LFALSE (* constant boolean expression *)
12 | RANDOM_BEXP (* random boolean expression *)
13 | LAND of lbexp list (* boolean conjunction *)
14 | LOR of lbexp list (* boolean disjunction *)
15 | LGE of int array (* LGE $a$ ... $an$ $b$ is $a1$ $+$ ... $+an$ $\geq$ $b$ *)
16 | LEQ of int array (* LGE $a$ ... $an$ $b$ is $a1$ $+$ ... $+an$ $=$ $b$ *)
17 and label = Abstract_Syntax.label
18 and lcom =
19 | LSKIP of label * label
20 | LASSIGN of label * variable * laexp * label
21 | LSEQ of label * (lcom list) * label
22 | LIF of label * lbexp * lbexp * lcom * lcom * label
23 | LWHILE of label * lbexp * lbexp * lcom * lcom * label
24 val after : lcom -> label (* command exit label *)
25 val incom : label -> lcom -> bool (* label in command *)

The linear abstraction of programs

- The linear abstraction of programs consists in replacing all non-linear expressions by a random choice (which is safe). Moreover the linear expressions are transformed into the array form defined in Linear_Syntax.mli.
- The linearization is by induction on the syntax of expressions by combination of the linear 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:

$$(a_{0}x_{0} + \ldots a_{n}x_{n} + a_{n+1}) + (b_{0}x_{0} + \ldots b_{n}x_{n} + b_{n+1}) = (a_{0} + b_{0})x_{0} + \ldots (a_{n} + b_{n})x_{n} + (a_{n+1} + b_{n+1})$$

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.
84   | (TIMES (a1, a2)) -> LINEAR_AEXP 1)  
85     | (match (linearize_aexp a1, linearize_aexp a2) with  
86     | (RANDOM_AEXP, _) | (_, RANDOM_AEXP) -> RANDOM_AEXP  
87     | (LINEAR_AEXP 11, LINEAR_AEXP 12) ->  
88     try  
89     for i=0 to n do  
90       let l = Array.make (n+1) 0 in  
91       (for i=0 to n do  
92         machine_binary_times (NAT l1.(i)) (NAT l2.(n)) ...  
93         done;  
94         LINEAR_AEXP 1)  
95     with Not_constant ->  
96     try  
97     for i=0 to n-1 do if 12.(i)<0 then raise Not_constant done;  
98     let l = Array.make (n+1) 0 in  
99     (for i=0 to n do  
100       machine_binary_times (NAT l1.(i)) (NAT l2.(n)) ...  
101       done;  
102       LINEAR_AEXP 1)  
103   | (MINUS (a1, a2)) ->  
104     (match (linearize_aexp a1, linearize_aexp a2) with  
105     | (RANDOM_AEXP, _) | (_, RANDOM_AEXP) -> RANDOM_AEXP  
106     | (LINEAR_AEXP 11, LINEAR_AEXP 12) ->  
107     try  
108     for i=0 to n do if 12.(i)<0 then raise Not_constant done;  
109     let l = Array.make (n+1) 0 in  
110     (for i=0 to n do  
111       machine_binary_minus (NAT l1.(i)) (NAT l2.(i)) with  
112       | ERROR_NAT _ -> raise Abstract_To_Linear_Syntax_error  
113       | NAT v -> 1.(i) <- v  
114       done;  
115       LINEAR_AEXP 1)  
116   | (DIV (a1, a2)) ->  
117     (match (linearize_aexp a1, linearize_aexp a2) with  
118     | (RANDOM_AEXP, _) | (_, RANDOM_AEXP) -> RANDOM_AEXP  
119     | (LINEAR_AEXP 11, LINEAR_AEXP 12) ->  
120     try  
121     for i=0 to n do if 12.(i)<0 then raise Not_constant done;  
122     let l = Array.make (n+1) 0 in  
123     (for i=0 to n do  
124       machine_binary_div (NAT l1.(i)) (NAT l2.(i)) with  
125       | ERROR_NAT _ -> raise Abstract_To_Linear_Syntax_error  
126       | NAT v -> 1.(i) <- v  
127       done;  
128       LINEAR_AEXP 1)  
129   | (MOD (a1, a2)) -> RANDOM_AEXP  
130     with Abstract_To_Linear_Syntax_error -> RANDOM_AEXP  
131     (* Linearization of boolean operations *)  
132     let rec linearize_bexp b =  
133     match b with  
134     | TRUE -> LTRUE  
135     | FALSE -> LFALSE  
136     | (EQ (a1, a2)) ->  
137     (match (linearize_aexp a1), (linearize_aexp a2) with  
138     | (RANDOM_AEXP, _) | (_, RANDOM_AEXP) -> RANDOM_BEXP  
139     | (LINEAR_AEXP 11, LINEAR_AEXP 12) ->
let t = Array.make ((number_of_variables ()) + 1) 0 in
for i = 0 to (number_of_variables ()) do
  t.(i) <- l2.(i)
  if (i, LFALSE) -> LFALSE
  (match (linearize_aexp a1),
   (linearize_aexp (MINUS (a2, (Abstract_Syntax.NAT "1")))) with
   | (RANDOM_AEXP, _) | (_, RANDOM_AEXP) -> RANDOM_BEXP
   | (LINEAR_AEXP 11, LINEAR_AEXP 12) ->
     let t = Array.make ((number_of_variables ()) + 1) 0 in
     for i = 0 to (number_of_variables ()) do
       t.(i) <- l2.(i)
     done;
   | (LTRUE, b) -> b
   | (a, LTRUE) -> a
   | (RANDOM_BEXP, _) | (_, RANDOM_BEXP) -> RANDOM_BEXP
   | (LAND l1, ...) of commands *)
   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 11, LAND 12) -> LAND (11012)
| (LAND 1, b) -> LAND (10[b])
| (b, LAND 1) -> LAND (b:1)
| (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 11, LOR 12) -> LOR (11012)
  | (LOR 1, b) -> LOR (10[b])
  | (b, LOR 1) -> LOR (b:1)
  | (b1', b2') -> LOR [b1';b2']
(* Linearization of commands *)

let rec linearize_com c =
match c with
| SKIP (11, 12) -> (LSKIP (11, 12))
| ASSIGN (11, v, a, 12) -> (LASSIGN (11, v, (linearize_aexp a), 12))
| SEQ (11, cl, 12) -> (LSEQ (11, (linearize_com_list cl), 12))
| IF (11, b, nb, ct, cf, 12) ->
  (LIF (11, (linearize_bexp b), (linearize_bexp nb),
        (linearize_com ct), (linearize_com cf), 12))
| WHILE (11, b, nb, c, 12) ->
  (LWHILE (11, (linearize_bexp b), (linearize_bexp nb),
            (linearize_com c), 12))
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
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_string " = 0")
| (LOR b1) -> print_string "(";
  (print_Lbexp_or b1);
  print_string "")
| (LAND b1) -> print_string "(";
  (print_Lbexp_and b1);
  print_string "")
and print_Lbexp_or b1 = match b1 with
  | [] -> ()
  | b :: [] -> print_Lbexp b
  | b :: b1' -> print_Lbexp b;
  | print_string " | ";
  | print_Lbexp_or b1'
  | print_Lbexp_and b1 = match b1 with
  | [] -> ()
  | b :: [] -> print_Lbexp b
  | b :: b1' -> print_Lbexp b;
  | print_string " & ";
  | print_Lbexp_or b1'
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 ()
         and print_seq n s')
and print_com n c' =
257     match c' with
258     | (LSKIP (1,m)) ->
259         print_margin_label n l; print_margin (n+1);
260         print_string "skip"
261     | (LASSIGN (1,v,a,m)) ->
262         print_margin_label n l;
263         print_margin (n+1); print_variable v; print_string " := ";
264         print_Laexp a
265     | (LSEQ (1,s,m)) ->
266         print_seq n s
267     | (LIF (1,b,nb,t,f,m)) ->
268         print_margin_label n l; print_margin (n+1);
269         print_string "if "; print_Laexp b;
270         print_string " then"; print_newline();
271         print_com_line (n+2) t;
272         print_margin (n+1); print_string "else";
273         print_string " {"; print_Laexp nb; print_string "}\";
274         print_newline ()

-- 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;

print_com_line (n+2) f;
print_margin (n+1); print_string "fi"

| (LWHILE (1,b,nb,c',m)) ->

print_margin_label n l; print_margin (n+1);
print_string "while "; print_Laexp b;
print_string " do"; print_newline();
print_com_line (n+2) c';
print_margin (n+1); print_string "od";
print_string " {"; print_Laexp 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
** 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:

---

** Linear abstraction

A generic linear relational abstract interpreter

---
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:

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

which are implemented as follows:

```ocaml
(* string of variable v in symbol table *)
exception Error_string_of_variable of string
let string_of_variable v =
let ... !p
```

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)

```ocaml
1 (* variables.mli *)
2 open Symbol_Table
3 type variable = Symbol_Table.variable
4 val number_of_variables : unit -> int
5 val for_all_variables : (variable -> 'a) -> unit
6 val print_variable : variable -> unit
7 val map_variables : (variable -> unit) -> (variable -> unit) -> unit
8 val string_of_variable : variable -> string
9 (* variables.ml *)
10 open Symbol_Table
11 type variable = Symbol_Table.variable
12 let number_of_variables = number_of_variables
13 let for_all_variables = for_all_variables
14 let print_variable = print_variable
15 let map_variables = map_variables
16 let string_of_variable = string_of_variable
```
Generic linear relational abstract domains

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

18 (* senv.mli *)
19 open Linear_Syntax
20 open Array
21 open Variables
22 (* set of environments *)
23 type t
24 (* relational library initialization *)
25 val init : unit -> t
26 (* relational library exit *)
27 val quit : unit -> unit
28 (* infimum *)
29 val bot : unit -> t
30 (* check for infimum *)

- The modules handling the concrete values are unchanged:
values.mli, values.ml
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 \texttt{f\_ASSIGN x f r} where \( f = [a_0, a_1; \ldots; a_n; a_{n+1}] \) which an upper-approximation of

\[ \alpha(\{\rho \in \gamma(x) \mid a_0.\rho(x_0) + \ldots + a_n.\rho(x_n) \geq a_{n+1}\}) \]

which is \( \Omega_\alpha \) 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 \texttt{f\_LGE a r} where \( a = [a_0, a_1; \ldots; a_n; a_{n+1}] \) which an upper-approximation of

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

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

---

Generic linear relational analysis of boolean expressions

- The boolean expressions can be handled generically:

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

by \texttt{f\_LGE a r} where \( a = [a_0, a_1; \ldots; a_n; a_{n+1}] \) which an upper-approximation of

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

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

---
Generic linear relational analysis of 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:

70   match b with
71       | RANDOM_BEXP -> r
72       | LTRUE   -> r
73       | LFALSE  -> (Aenv.bot ()
74       | (LGE a) -> (Aenv.f_LGE a r)
75       | (LEQ a) -> (Aenv.f_LEQ a r)
76       | (LAND l) -> let rec andlist l = match l with
77             | []     -> (raise (Error "empty LAND incoherence"))
78             | b':[]  -> a_bexp b' r
79             | b':l'  -> Aenv.meet (a_bexp b' r) (andlist l')
80       in andlist l
81       | (LOR l) -> let rec orlist l = match l with
82             | []     -> (raise (Error "empty LOR incoherence"))
83             | b':[]  -> a_bexp b' r
84             | b':l'  -> Aenv.join (a_bexp b' r) (orlist l')
85       in orlist l
86
87 (* acom.mli *)
88 open Linear_Syntax
89 open Aenv
90 (* forward abstract interpretation of commands *)
91 val acom : lcom -> Aenv.t -> label -> Aenv.t

- The implementation is as follows:

92 (* acom.ml *)
93 open Linear_Syntax
94 open Aenv

95 open Abexp
96 open Fixpoint
97 (* collecting semantics of commands *)
98
d99 exception Error of string
100 let rec acom c r l =
101     match c with
102         | (LSKIP (l', l'')) ->
103             if (l = l') then r
104             else if (l = l'') then r
105             else (raise (Error "SKIP incoherence"))
106         | (LASSIGN (l', x, a, l'')) ->
107             if (l = l') then r
108             else if (l = l'') then
109                 f_ASSIGN x a r
110             else (raise (Error "ASSIGN incoherence"))
111         | (LSEQ (l', s, l'')) ->
112             (acomseq s r 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

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

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

```ocaml
142 print_string "luis: x =\n";
143 print x;
144 let x' = (f x) in
145 print_string "luis: f(x) =\n";
146 print x';
147 if (c x' x) then
148 (print_string "luis: f(x) <= x, convergence\n";
149 x)
150 else
151 (let x'' = (w x') in
152 print_string "luis: x \\/ f(x) =\n";
153 print x'');
154 luis x'' c w f)
155 (* iteration of f from postfixpoint x with ordering c and narrowing n *)
156 let rec llis x c n f =
157 print_string "llis: x =\n";
158 print x;
159 let x' = (f x) in
```

```ocaml
152 print_string "llis: x \\/ f(x) =\n";
153 print x';
154 llis x'' c w f)
```
The generic linear relational abstract interpreter

```
160    print_string "llis: f(x) =\n";
161    print x';
162    let x'' = (n x x') in
163    print_string "llis: x /\ f(x) =\n";
164    print x'';
165    if (c x x'') then
166       (print_string "llis: x <= x /\ f(x), convergence\n";
167       x')
168      else llis x'' c n f
169      (* lfp x c w n f : iterative computation of a c-postfixpoint of f *)
170      (* c-greater than or equal to the prefixpoint x (x <= f(x)) with *)
171      (* widening w and narrowing n *)
172      let lfp x c w n f = llis (luis x c w f) c n f
173      (* gfp x c n f : iterative computation of a c-postfixpoint of f *)
174      (* c-less than or equal to the postfixpoint x (f(x) <= x) with *)
175      (* narrowing n *)
176      let gfp x c n f = llis x c n f
```

```
190    let p = (abstract_syntax_of_program arg) in
191    (print_string "** Program:\n";
192    pretty_print p;
193    let p' = (linearize_com p) in
194    print_string "** Linearized program:\n";
195    lpretty_print p';
196    init ();
197    print_string "** Precondition:\n";
198    print (initerr ()));
199    print_string "** Postcondition:\n";
200    print (acom p' (initerr ()) (after p'));
201    quit ()
```

makefile

```
202 EXAMPLES = ../Examples
203 SOURCES = \n204 symbol_Table.mli \n205 symbol_Table.ml \n206 ... \n215 parser.ml
```
234  fixpoint.ml \\
235   acom.mli \\
236   acom.ml \\
237  main.ml \\
238  \\
239  .PHONY : help \\
240   help : \\
241     @echo "" \\
242     @echo "Forward relational static analysis:" \\
243     @echo "make [help] : this help" \\
244     @echo "make pol : polyhedral analysis" \\
245     @echo ".a.out file.sil : analyze file.sil" \\
246     @echo "make examples : analyze all examples" \\
247     @echo "make clean : remove auxiliary files" \\
248     @echo "" \\
249   \\
250  .PHONY : pol 

251  pol:

---

252  @/bin/rm -f aenv.ml \\
253  @ln -s ../Relational-FW/Polyhedra/aenv.ml aenv.ml \\
254  @echo "Polyhedral analysis" \\
255  ocamlyacc parser.mly \\
256  ocamllex lexer.ml \\
257  ocamic -custom -I /usr/local/lib -I /usr/local/lib/ocaml \\
258     -cclib "-L/usr/local/lib -L/usr/local/lib/ocaml -lpolkag_caml" \\
259  polka.cma \{$(SOURCES)$
260  \\
261  include \{$(EXAMPLES)\}/makefile \\
262  \\
263  .PHONY : clean \\
264  clean : \\
265     /bin/rm -f *.cml *.cmo a.out lexer.ml parser.mli parser.ml

---

**Polyhedral relational static analysis**
Polyhedral abstract domain

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

\[
\begin{align*}
\{ \sum_{i=1}^{n} a_{ji} \cdot x_i \leq a_{j,n+1} \\
\end{align*}
\]

\( j = 1, \ldots, m \)

\( * \) which is an unsolved soundness problem whence implementing the algorithms in \( \mathbb{R} \) with floats.

Example of polyhedral static analysis

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

\[
\begin{align*}
% \text{example40.mil} \\
I:=2; J:=0; B:=?; \\
\text{while } B<0 \text{ do} \\
\quad \text{if } B<1 \text{ then} \\
\quad \quad I:=I+1 \\
\quad \text{else} \\
\quad \quad I:=I+2; \\
\quad J:=J+1 \\
\quad fi \\
\text{od;} ; \\
\text{** Program:} \\
\ldots
\end{align*}
\]
**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>=0,1>=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

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

- **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

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

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

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

- **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

\[
\gamma(\langle A, B \rangle) \overset{\text{def}}{=} \{ X \in \mathbb{F}^n | AX \geq B \}
\]

Example

\[
\begin{align*}
\text{System of linear inequalities:} & \\
\text{System of generators:} & \\
\end{align*}
\]

\[
\begin{align*}
P &= \{ (x, y) | \begin{array}{c} -x + y \leq 1 \\ -x + y \leq 3 \\ y \geq 1 \\ y \leq 3 \\ x \geq 3 \\ x \geq 0 \\ y \geq 0 \\
\end{array} \} \\
V &= \{ v_0(2,1), v_1(1) \} \\
R &= \{ r_0(1), r_1(1) \}
\end{align*}
\]

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 vertex of ray can be eliminated without changing the polyhedron \( \gamma(\langle V, R \rangle) \)

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

Chenikova algorithm

- Start with $P_0 = \mathbb{Q}^n$ given by the system of generators $V_0 = \{0\}$ and $R_0 = \{\hat{v}_1, \ldots, \hat{v}_n, -\hat{v}_1, \ldots, -\hat{v}_n\}$ where $\{\hat{v}_1, \ldots, \hat{v}_n\}$ 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 $(v, v')$ of vertices in $V_{k-1}$ such that $av \geq b$ and $av' < b$, their convex combination $\frac{b-a v'}{a v-a v'} v - \frac{b-a v}{a v-a v'} v'$ belongs to $V_k$;

References

4. for any pair \( \langle v, r \rangle \) of vertex 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 \( v + \frac{b-av}{ar} \cdot r \) 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 \( R_k \).

Remarks on Chenikova algorithm
- In the worst case the algorithm can generate an exponential number of generators (an hypercube in dimension \( n \) 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
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:
  \[
  \forall u \in V : Au \geq B \quad \land \quad \forall r \in R : Ar \geq 0
  \]
- **Test for equality**: \( P = Q \) iff \( P \subseteq Q \land P \supseteq Q \).

- **Intersection \( \cap \)**: conjunction of systems of linear inequalities;

- These operations "\( = \)", "\( \subseteq \)", "\( = \)" and "\( \cap \)" 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.
Abstract polyhedral transfer functions

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

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

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

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

  $$ R' = \{Ar \mid r \in R\} $$

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

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 $\langle V, R, L \rangle$;
- $P \lor Q$ is $Q$ if $P$ is empty;
- $P \lor Q$ 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:**
Polyhedral widening improvements

Possible improvements:

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

Example 1 of polyhedral analysis

Generic-FW-REL-Abstract-Interpreter % ./a.out
./Examples/example41.sil
** Program:
$X := 7; \ Y := X;$
while ($((0 < X) \ (X = 0)) \ & \ ((0 < Y) \ (Y = 0))$) do
$Y := (Y + 1)$
cd $((X < 0) \ (Y < 0))$
** Linearized program:
$X := 7; \ Y := 1.X + 0.Y + 0;$
while ($(-1.X + +1.Y + -1 >= 0 \ | \ -1.X + 0.Y + 0 = 0) \ & \ (0.X + 1.Y + 1 >= 0)\ |
\ 0.X + 1.Y + 0 = 0)$) do
$Y := 0.X + 1.Y + 1$
cd $(-1.X + 0.Y + -1 >= 0 \ | \ 0.X + -1.Y + -1 >= 0)$
** Precondition:
$\{1>=0\}$
** Postcondition:
$\{1=0, X<=Y, X<1<0\}$

Example 2:

$P = \{ (x, y) | 0 \leq x \land x \leq y \land y \leq x \}$
$Q = \{ (x, y) | 0 \leq x \leq y \leq x + 1 \}$
$P \lor Q = \{ (x, y) | 0 \leq x \leq y \}$

Example 3:

$P = \{ (x, y) | x \leq 0 \land x \geq 0 \land y \leq 0 \land y \geq 0 \}$
$Q = \{ (x, y) | 0 \leq y \leq x \leq 1 \}$
$P \lor Q = \{ (x, y) | 0 \leq y \leq x \}$
instead of $\{ (x, y) | 0 \leq y \land 0 \leq x \}$. 
The loop invariant:

\[ \{ (x, y) \mid x \geq 0 \wedge y \geq 0 \wedge x \leq y \} \]

is the least fixpoint of:

\[ F(X) = \{ (x, y) \mid x \geq 0 \wedge y \geq 0 \wedge ((x = y) \vee ((x, y - 1) \in X)) \} \]

The iterates are as follows:

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

\[ \begin{array}{c}
0 \\
1 \\
\end{array} \begin{array}{c}
y \\
x \\
\end{array} \]

- \( \hat{X}^2 = \hat{X}^1 \cap F(\hat{X}^1) = \{ (x, y) \mid 0 \leq x \leq y \leq x + 1 \} \)

\[ \begin{array}{c}
0 \\
1 \\
\end{array} \begin{array}{c}
y \\
x \\
\end{array} \]

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

Example 2 of polyhedral analysis

\[ \begin{array}{c}
X = 2; \ I = 0; \\
\text{while } \bullet \ (I < 10) \{ \\
\quad \text{if } (?) \ X = X + 2; \ \text{else } X = X - 3; \\
\quad I = I + 1; \\
\} \end{array} \]

- The narrowing is simply a finite number of intersections.
- The iterations with widening/narrowing are as follows (the result is given at program point \( \bullet \)):

\[ \begin{array}{c}
X^{21} \{ X = 2; I = 0 \} \\
X^{22} \{ X = 2; I = 0 \} \vee \{ X \in [-1; 4], I = 1 \} = \\
\{ I \geq 0, X \in [2 - 3I; 2I + 2] \} \\
X^{23} \{ I \geq 0, X \in [2 - 3I; 2I + 2] \} \cap \{ I \in [0; 10], X \in [2 - 3I; 2I + 2] \} = \\
\{ I \in [0; 10], X \in [2 - 3I; 2I + 2] \} \\
\end{array} \]

- The final result at program point \( \bullet \) 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
%
```

```
B := ?; X := 0.B + 0.X ...
```

Course 16.399: "Abstract interpretation", Tuesday May 10th, 2005 — 201 — ľ P. Cousot, 2005

```
– 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
%
```

```
B := ?; X := 0.B + 0.X ...
```


```
B := ?; X := 0.B + 0.X ...
```

Course 16.399: "Abstract interpretation", Tuesday May 10th, 2005 — 203 — ľ P. Cousot, 2005

```
\text{Strict inequalities}
```

– Polyhedral analysis can be extended to include strict constraints:

\[
\{ X ~|~ AX \ge B \land A'X > B' \}
\]

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

\[
a_1X_1 + \ldots + a_nX_n \ge 0 \text{ is encoded as } a_1X_1 + \ldots + a_nX_n + 0.X_\varepsilon \ge 0
\]

– The concretization is

\[
\gamma(\text{P}) \overset{\text{def}}{=} \{ \langle X_1, \ldots, X_n \rangle \mid \exists X_\varepsilon > 0 : \langle X_1, \ldots, X_n, X_\varepsilon \rangle \in \gamma(\text{P}) \}
\]

– Minimal representations must be adapted to \( X_\varepsilon \)

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

\[\text{References}\]


Course 16.399: "Abstract interpretation", Tuesday May 10th, 2005 — 204 — ľ P. Cousot, 2005
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 $x + 2y + 5z \leq 7$ 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/bjeanet/archives/polka/

---

Interface with the New Polka Library

**Data types:**

```plaintext
type dimsup = {
  pos: int;
  abdimm: int;
};
```

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

---

Initialization and finalization functions:

**initialize** : bool -> int -> int -> unit

```
initialze strict maxdim maxrows initializes internal data-structures and global variables of the library:

- **strict** indicates whether strict inequalities are enabled or not;
- **maxdim** 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

Free internal data-structure used in the library.
- **Polyhedra:**

  - Define the type of polyhedron:
    
    ```
    t : the type of polyhedron
    ```

  - Constructors for OCaml polyhedra:
    
    ```
    empty : int -> t
    universe : int -> t
    ```

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

  - Change of dimension of OCaml polyhedra:
    
    ```
    add_dims_and_embed_multi : t -> Polka.dimsup array -> t
    ```

    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
    ```

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

  - Intersection and convex hull of OCaml polyhedra:
    
    ```
    inter : t -> t -> t
    ```

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

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

    - Linear transformation on OCaml polyhedra:
      
      ```
      assign_var : t -> int -> Vector.t -> t
      ```

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

      Same as C function `poly_assign_variable`.

      Same as C function `poly_substitute_variable`.

- **Vector:**

  - Define the abstract datatype for vectors:
    
    ```
    t
    ```

    Abstract datatype for vectors.

    ```
    make : int -> t
    make size
    ```

    Returns a vector of size size with all coefficients initialized to 0. size=0 is accepted.

    ```
    set : t -> int -> int -> unit
    set vec index val
    ```

    Stores the value val in the corresponding coefficient of the vector.
- Widening operator on OCaml polyhedra:

\[
\text{widening : } t \rightarrow t \rightarrow t
\]

Function

- Input and output of OCaml polyhedra:
This functions use the pretty input/output facilities described in module Polka.

\[
\text{print_constraints : Format.formatter} \rightarrow t \rightarrow \text{unit}
\]

Function

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

The polyhedral abstract environment domain

1  (* aenv.ml *)
2  open Linear_Syntax
3  open Variables
4  type lattice = BOT | TOP
5  type t = NULL of lattice (* if no variable, dimension = 0 *)
6  | POLY of Poly.t (* must be of dimension > 0 *)
7  exception PolyError of string
8  (* relational library initialization *)
9  let init () = (Polka.initialize false 10000 100; Polka.strict := false)
10  (* relational library exit (* print statistics *) *)
11  let quit () = Polka.finalize ()
12  (* infimum *)
13  let bot () = match (number_of_variables ()) with
14  | 0 -> NULL BOT
15  | n -> POLY (Poly.empty n) (* 1 <= n <= polka_maxcolumns-polka_dec *)
16  (* check for infimum *)
17  let is_bot r = match r with
18  | NULL BOT -> true
19  | NULL TOP -> false
20  | POLY p -> (Poly.is_equal p (Poly.empty (number_of_variables ()))))
21  (* uninitialization *)
22  let initerr () = match (number_of_variables ()) with
23  | 0 -> NULL TOP
24  | n -> POLY (Poly.universe n)
25  (* supremum *)
26  let top () = match (number_of_variables ()) with
27  | 0 -> NULL TOP
28  | n -> POLY (Poly.universe n)
29  (* least upper bound *)
30  let ljoin 11 12 = match (11, 12) with
31  | BOT, _ -> 12
32  | _, BOT -> 11

- Compilation of the New Polka library:

The library can be implemented with various representations of integer (32, 64 bit, OTH, ...).

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/
51 | TOP, BOT -> false
52 let leq r1 r2 = match (r1, r2) with
53 | NULL 11, NULL 12 -> (NULL (ljoin 11 12))
54 | POLY p1, POLY p2 -> (Poly.is_equal_in p1 p2)
55 | _, _ -> raise (PolyError "leq")
56 (* equality *)
57 let eq r1 r2 = match (r1, r2) with
58 | NULL 11, NULL 12 -> (11 = 12)
59 | POLY p1, POLY p2 -> (Poly.is_equal p1 p2)
60 | _, _ -> raise (PolyError "eq")
61 (* printing *)
62 let print r = match r with
63 | NULL BOT -> (print_string "(_).\n")
64 | NULL TOP -> (print_string "(T\n")
65 | POLY p ->
66 | (Poly.minimize p; (* to get the constraints and generators of p *)
67 Poly.print_constraints string_of_variable Format.std_formatter p;
68 Format.pp_print_newline Format.std_formatter ()

---

51 | TOP, BOT -> false
52 let leq r1 r2 = match (r1, r2) with
53 | NULL 11, NULL 12 -> (NULL (ljoin 11 12))
54 | POLY p1, POLY p2 -> (POLY (Poly.union p1 p2))
55 | _, _ -> raise (PolyError "join")
56 (* greatest lower bound *)
57 let immeet 11 12 = match (11, 12) with
58 | TOP, _ -> 12
59 | _, TOP -> 11
60 | _, _ -> BOT
61 let meet r1 r2 = match (r1, r2) with
62 | NULL l1, NULL l2 -> (NULL (ljoin l1 l2))
63 | POLY p1, ... | _, TOP -> true
64 | _, _ -> TOP
65 let join r1 r2 = match (r1, r2) with
66 | NULL l1, NULL l2 -> (NULL (ljoin l1 l2))
67 | POLY p1, ... | _, TOP -> true
68 let vector_of_lin_expr a =
69 let v = Vector.make (number_of_variables () + 2) in
70 (Vector.set v 0 1;
71 for i = 0 to (number_of_variables () - 1) do
72 (Vector.set v (i+2) a.(i))
73 done;
74 (*
75 Vector._print v;
76 Vector.print_constraint string_of_variable Format.std_formatter v;
77 Format.pp_print_newline Format.std_formatter ();
78 *)
79 v)
80 (* f_ASSIGN x f r = {e[x <- i] | e in r / i in f({e} cap 1} *)
81 let f_ASSIGN x f r =
82 match r with
83 | NULL _ -> r
84 | POLY p -> (match f with
85 | RANDOM_AEXP ->
86 let d = [{Polka.pos = x; Polka.nbdims = 1}]
87 in
88 (POLY (Poly.add_dims_and_embed_multi (Poly.del_dims_multi p d) d))
89 | LINEAR_AEXP a ->
90 (POLY (Polka.assign_var p x (vector_of_lin_expr a)))))
91 (* f_LGE a r = {e in r | a0.v0+...+an-1.vn-1+an >= 0} *)
92 let f_LGE a r =
93 match r with
94 | NULL _ -> r
95 | POLY p -> POLY (Poly.add_constraint p (vector_of_lin_expr a))
96 | (* f_LEq a r = {e in r | a0.v0+...+an-1.vn-1+an = 0} *)
97 let f_LEq a r =
98 match r with
99 | NULL _ -> r
100 let minus = Array.map (fun x -> (- x))
101 let f_LEq a r = meet (f_LGE a r) (f_LEq minus a r)
102 (* widening *)
103 let widen r1 r2 = match (r1, r2) with
104 | NULL 11, NULL 12 -> (NULL (ljoin 11 12))
105 | POLY p1, POLY p2 -> (POLY (Poly.union p1 p2))
106 | _, _ -> raise (PolyError "join")
107 (* greatest lower bound *)
108 let immeet 11 12 = match (11, 12) with
109 | TOP, _ -> 12
110 | _, TOP -> 11
111 | _, _ -> BOT
112 let meet r1 r2 = match (r1, r2) with
113 | NULL l1, NULL l2 -> (NULL (ljoin l1 l2))
114 | POLY p1, ... | _, TOP -> true
115 | _, _ -> TOP
116 let join r1 r2 = match (r1, r2) with
117 | NULL l1, NULL l2 -> (NULL (ljoin l1 l2))
118 | POLY p1, ... | _, TOP -> true
119 | _, _ -> TOP
120
Typescript examples of affine inequality analyses

Top element of the lattice:

Generic-FW-REL-Abstract-Interpreter % ./a.out

```
POLY p1, POLY p2 -> (POLY (Poly.widening p1 p2))
POLY p1, POLY p2 -> (POLY (Poly.widening p1 p2))
let narrow a = a (* does not ensure termination *)
```

Bottom element of the lattice:

Generic-FW-REL-Abstract-Interpreter % ./a.out

```
x := 1;
while (x > 0) do
 x := x + 1000000
end;
** Linearized program:
 x := 0.x + 1;
 while 1.x + -1 >= 0 do
  x := 1.x + 1000000
 end {(-1.x + -1 >= 0 | -1.x + 0 = 0)}
** Precondition:
 {1>=0}
** Postcondition:
 empty(1)
```
** Precondition:
\{1\geq 0\}

** Postcondition:
\{10x+y=200, y\leq 200, y\geq 0\}

---

** Bibliography **


---

My MIT web site is [http://www.mit.edu/~cousot/](http://www.mit.edu/~cousot/)