Recalls on operational and collecting semantics

Programs states (recall from lecture 5)

States \( \langle \ell, \rho \rangle \in \Sigma[P] \) record a program point \( \ell \in \text{inp}[P] \) and an environment \( \rho \in \text{Env}[P] \) assigning values to variables:

\[
\Sigma \in \text{Prog} \mapsto \rho(\mathbb{V} \times \mathbb{R}),
\]

\[
\Sigma[P] \overset{\text{def}}{=} \text{inp}[P] \times \text{Env}[P].
\]
Program transition relation (recall from lecture 5)
The small-step operational semantics of commands, sequences and programs \( C \in \text{Com} \cup \text{Seq} \cup \text{Prog} \) within a program \( P \in \text{Prog} \) involves transition judgements

\[
\langle \ell, \rho \rangle \mathrel{\xrightarrow{[\!\![C]\!\!]}} \langle \ell', \rho' \rangle.
\]

Transition system of a program (recall from lecture 5)
The transition system of a program \( P = S ;; \) is

\[
\langle \Sigma[P], \tau[P] \rangle
\]

where \( \Sigma[P] \) is the set (1) of program states and \( \tau[C] \), \( C \in \text{Cmp}[P] \) is the transition relation for component \( C \) of program \( P \), defined by

\[
\tau[C] \overset{\text{def}}{=} \{ (\langle \ell, \rho \rangle, \langle \ell', \rho' \rangle) \mid \langle \ell, \rho \rangle \mathrel{\xrightarrow{[\!\![C]\!\!]}} \langle \ell', \rho' \rangle \} \tag{2}
\]

Initial States (recall from lecture 5)
Execution starts at the program entry point with all variables uninitialized:

\[
\text{Entry}[P] \overset{\text{def}}{=} \{ \langle \text{at}_P[P], \lambda x \in \text{Var}[P]. \Omega_1 \rangle \} \tag{3}
\]

Final States (recall from lecture 5)
Execution ends without error when control reaches the program exit point

\[
\text{Exit}[P] \overset{\text{def}}{=} \{ \text{after}_P[P] \} \times \text{Env}[P].
\]

When the evaluation of an arithmetic or boolean expression fails with a runtime error, the program execution is blocked so that no further transition is possible.
Big-step operational semantics of a program
(recall from lecture 5)

- The big-step operational semantics of a program $P$ is

$$\langle \Sigma[P], t^*[P] \rangle$$

where $t^*[P] \overset{\text{def}}{=} (t[P])^*$ is the reflexive transitive closure of the transition relation $t[P]$

- Infinite executions are not considered with this semantics

---

Structural big-step operational semantics
(recall from lecture 8)

$$\begin{align*}
\tau^*[\text{skip}] &= 1_\Sigma[P] \cup \tau[\text{skip}] \\
\tau^*[X := A] &= 1_\Sigma[P] \cup \tau[X := A] \\
\tau^*[\text{if } B \text{ then } S_t \text{ else } S_f \text{ fi}] &= \\
&\quad (1_\Sigma[P] \cup \tau^B) \circ \tau^*[S_t] \circ (1_\Sigma[P] \cup \tau^f) \cup \\
&\quad (1_\Sigma[P] \cup \tau^B) \circ \tau^*[S_f] \circ (1_\Sigma[P] \cup \tau^f)
\end{align*}$$

---

where:

$$\begin{align*}
\tau^B &= \{\langle \tau_p[\text{if } B \text{ then } S_t \text{ else } S_f \text{ fi}], \rho \rangle \mid \rho \vdash B \Rightarrow \mathbf{t} \}\} \\
\tau^B &= \{\langle \tau_p[\text{if } B \text{ then } S_t \text{ else } S_f \text{ fi}], \rho \rangle \mid \rho \vdash T(\neg B) \Rightarrow \mathbf{t} \}\} \\
\tau^R &= \{\langle \tau_p[S], \rho \rangle \mid \rho \in \text{Env}[P] \}\} \\
\tau^t &= \{\langle \tau_p[S], \rho \rangle \mid \rho \in \text{Env}[P] \}\}
\end{align*}$$

---

$$\tau^*[\text{while } B \text{ do } S \text{ od}] = (1_\Sigma[P] \cup \tau^*[S] \circ \tau^R) \circ (\tau^B \circ \tau^*[S] \circ \tau^R)^* \circ \\
(1_\Sigma[P] \cup \tau^B \circ \tau^*[S] \cup \tau^B) \cup \tau[S]^*$$

where:

$$\begin{align*}
\tau^B &= \{\langle \tau_p[\text{while } B \text{ do } S \text{ od}], \rho \rangle \mid \rho \vdash B \Rightarrow \mathbf{t} \}\} \\
\tau^B &= \{\langle \tau_p[\text{while } B \text{ do } S \text{ od}], \rho \rangle \mid \rho \vdash T(\neg B) \Rightarrow \mathbf{t} \}\} \\
\tau^R &= \{\langle \text{after}_p[S], \rho \rangle \mid \rho \in \text{Env}[P] \}\} \\
\tau^t &= \{\langle \text{after}_p[S], \rho \rangle \mid \rho \in \text{Env}[P] \}\}
\end{align*}$$
Definition of the forward collecting semantics of boolean expressions (recall from lecture 8)

Recall the collecting semantics $\text{Cbexp}[B] R$ of a boolean expression $B$ from course 8:

$$\text{Cbexp} \in \text{Bexp} \mapsto \wp(\text{Env}[P]) \overset{\uparrow}{\mapsto} \wp(\text{Env}[P]),$$

$$\text{Cbexp}[B] R \overset{\text{def}}{=} \{ \rho \in R \mid \rho \vdash B \Rightarrow \text{tt} \}.$$ (7)

such that:

$$\text{Cbexp}[B] \left( \bigcup_{k \in S} R_k \right) = \bigcup_{k \in S} (\text{Cbexp}[B] R_k)$$

$$\text{Cbexp}[B][0] = \emptyset.$$

Backward collecting semantics of arithmetic expressions (recall)

The backward/top-down collecting semantics $\text{Baexp}[A](R) P$ of an arithmetic expression $A$ defines the subset of possible environments $R$ such that the arithmetic expression may evaluate, without producing a runtime error, to a value belonging to given set $P$:

$$\text{Baexp} \in \text{Aexp} \mapsto \wp(\mathbb{R}) \overset{\uparrow}{\mapsto} \wp(\mathbb{I}) \overset{\uparrow}{\mapsto} \wp(\mathbb{R}),$$

$$\text{Baexp}[A](R) P \overset{\text{def}}{=} \{ \rho \in R \mid \exists z \in P \cap \mathbb{I} : \rho \vdash A \Rightarrow i \}.$$ (8)

$\forall P \in \wp(\mathbb{R}) : \lambda R$. $\text{Baexp}[A](R) P$ is a lower closure operator

Structural specification of the forward collecting semantics of boolean expressions (recall from lecture 8)

- $\text{Cbexp}[\text{true}] R \overset{\text{def}}{=} R$
- $\text{Cbexp}[\text{false}] R \overset{\text{def}}{=} \emptyset$
- $\text{Cbexp}[A_1 \land A_2] \overset{\text{def}}{=} \text{Cbexp}[A_1] \text{Cbexp}[A_2] R$
  where $\text{Cbexp}[A_1] \text{Cbexp}[A_2] R \overset{\text{def}}{=} \{ \rho \in R \mid \exists v_1, v_2 \in \text{G}(\rho) \cap \mathbb{I} : v_1 \land v_2 \Rightarrow \text{tt} \}$
- $\text{Cbexp}[B_1 \lor B_2] R \overset{\text{def}}{=} \text{Cbexp}[B_1] R \cap \text{Cbexp}[B_2] R$
- $\text{Cbexp}[B_1 | B_2] R \overset{\text{def}}{=} \text{Cbexp}[B_1] R \cup \text{Cbexp}[B_2] R$

Structural definition of the backward/bottom-up collecting semantics of arithmetic expressions

- $\text{Baexp}[n](R) P = (n \in P \cap \mathbb{I} \Rightarrow R : \emptyset)$$ (9)$
- $\text{Baexp}[x](R) P = \{ \rho \in R \mid \rho(x) \in P \cap \mathbb{I} \}$$ (10)$
- $\text{Baexp}[?] (R) P = (P \cap \mathbb{I} = \emptyset \Rightarrow R)$$ (11)$
- $\text{Baexp}[u A'] (R) P = \text{Baexp}[A'] (R) (u^\ast (\text{Cbexp}[A'] R, P))$$ (12)$
  where $u^\ast (Q, P) \overset{\text{def}}{=} \{ v \in Q \mid u v \in P \cap \mathbb{I} \}$
- $\text{Baexp}[A_1 \land A_2] (R) P =$
  let $(P_1, P_2) = b^\ast (\text{Cbexp}[A_1] R, \text{Cbexp}[A_2] R, P)$ in $\text{Baexp}[A_1] (R) P_1 \cap \text{Baexp}[A_2] (R) P_2$
  where $b^\ast (P_1, P_2, P) \overset{\text{def}}{=} \{ (v_1, v_2) \in P_1 \times P_2 \mid v_1 \land v_2 \in P \cap \mathbb{I} \}$
Definition of the postcondition semantics of commands (recall from lecture 14)

The postcondition semantics of a command \( C \in \text{Com} \) (within a given program \( P \)) specifies the strongest postcondition \( \text{FPcom}[C]R \) satisfied by environments resulting from the execution of the command \( C \) starting in any of the environments satisfying the precondition \( R \), if and when this execution terminates.

\[
\begin{align*}
\text{FPcom} &\in \text{Com} \mapsto p(\text{Env}[P]) \xrightarrow{\downarrow} p(\text{Env}[P]) \\
\text{FPcom}[C]R &\equiv \{ \rho' \mid \exists \rho \in R : (\langle at_p[C], \rho \rangle, \langle \text{after}_p[C], \rho' \rangle) \in \tau^*[C] \}
\end{align*}
\]

The postcondition semantics of a command can be understood, up to an interpretation, as a predicate transformer.

Definition of the backward/precondition collecting semantics

- The backward collecting semantics of commands \( C \) is

\[
\begin{align*}
\text{BPcom} &\in \text{Com} \mapsto p(\text{Env}[P]) \xrightarrow{\downarrow} p(\text{Env}[P]) \\
\text{BPcom}[C]R &\equiv \{ \rho \mid \exists \rho' \in R : (\langle at_p[C], \rho \rangle, \langle \text{after}_p[C], \rho' \rangle) \in \tau^*[C] \}
\end{align*}
\]

- \( \text{BPcom}[C]R \) is the weakest precondition for execution of \( P \) from entry point of command \( C \) to have the possibility to reach a final state in \( R \) on exit of the command

- The condition is necessary but not sufficient, because of non-determinism. So non-termination or termination in a state not in \( R \) is left open.

Structural specification of the backward collecting semantics

\[
\begin{align*}
\text{BPcom}[\text{skip}]R &= R \\
\text{BPcom}[X := A]R &= \{ \rho \mid \exists i \in I : \rho \vdash A \Rightarrow i \wedge \rho[X := i] \in R \} \\
\text{BPcom}[\text{if} \ B \ \text{then} \ S_0 \ \text{else} \ S_1 \ \text{fi}]R &= \text{Cbexp}[B](\text{BPcom}[S_0]R \cup \text{Cbexp}[T(\neg(B))](\text{BPcom}[S_1]R)) \\
\text{BPcom}[\text{while} \ B \ \text{do} \ S \ \text{od}]R &= \text{while}_{\geq} X \cdot \text{Cbexp}[T(\neg(B)))]R \cup \text{Cbexp}[B](\text{BPcom}[S]X) \\
\text{BPcom}[C ; S]R &= (\text{BPcom}[C] \circ \text{BPcom}[S])R \\
\text{BPcom}[S ;;]R &= \text{BPcom}[S]
\end{align*}
\]
Proof of correctness of the structural specification of the backward collecting semantics

Proof.

We proceed by structural induction on the structure of programs.

$\text{Boo}[\text{loop}] R$

$= \alpha_\text{loop} (\text{loop}) R$

$= \alpha_\text{loop} [\text{loop}] R$

$= \{ p \mid p' \in R : <\alpha_\text{loop}[\text{loop}], p' > <\alpha_\text{loop}[\text{loop}], p > \}

= \{ p \mid p' \in R : <\alpha_\text{loop}[\text{loop}], p' > \}

= \{ p \mid p' \in R : p' \}

= \{ p \mid p' \in R : p' \}

= \{ p \mid p' \in R : p' \}

= \{ p \mid p' \in R : p' \}

$\text{Boo}[x:=A] R$

$= \alpha_\text{Def}[x:=A] (\text{Def}[x:=A]) R$

$= \alpha_\text{Def}[x:=A] (\text{Def}[x:=A], R$

$= \{ p \mid p' \in R : <\alpha_\text{Def}[x:=A], p' > <\alpha_\text{Def}[x:=A], p > \}

= \{ p \mid p' \in R : p' \}

= \{ p \mid p' \in R : p' \}

$\text{Boo}[\text{def} : x := A \Rightarrow c]$
\begin{equation}
\begin{aligned}
\text{Borel}[\text{C}] \quad \text{where} \quad C = \sigma \text{ is the set of states}
\implies \\
\alpha_p[C](t \upharpoonright C) \subseteq \\
\alpha_p[C]((z \in \mathbb{R}^+ \cup \mathbb{B})) \cup (z \cdot 0) \cup (z \cdot 1)
\end{aligned}
\end{equation}

We handle the case of the true alternation, the false
alternation being handled in the same way.

\begin{equation}
\begin{aligned}
\alpha_p[C]((z \in \mathbb{R}^+ \cup \mathbb{B})) \cup (z \cdot 0) \cup (z \cdot 1)
\end{aligned}
\end{equation}

\begin{equation}
\begin{aligned}
\text{We observe that if } \psi \equiv \psi', \text{ then } \phi \equiv \phi'.
\end{aligned}
\end{equation}
where a Galois connection
\[
\langle \Phi \circ (\Pi \times \Pi) \circ \sigma \rangle \leq \leq \langle \Phi \circ \sigma \rangle
\]
such that
\[
\alpha'(t(x)) \quad \text{where} \quad t = (\sigma \circ x \circ \Pi) \in \mathcal{R}
\]
\[
\alpha'(t(x)) = \sigma'(\alpha'(x))
\]
\[
\alpha'(x) \Rightarrow \alpha'(x)
\]
since \(\alpha'\) is a complete join morphism.
\[
\begin{align*}
\{ p \in t.p \mid \exists p'' \in R : \exists p' \in R : \langle \langle a^t_p, x, [S] \rangle, a^t_{p''} \rangle \in \mathcal{X} \}
\end{align*}
\]

\[= \{ p \mid \exists p' \in R : \langle a^t_p, x, [S] \rangle \subseteq \langle a^t_{p'}, x, [S] \rangle \}
\]
Definition of the backward/precondition collecting semantics

– The abstract backward/precondition semantics $\text{Abcom}[C]R$ of command $C$ in abstract environment $R$ satisfies:

$$\text{Abcom}[C]R \uparrow \alpha(\text{BPcom}[C](\gamma(R)))$$
Structural specification of the backward collecting semantics

\[
\text{Abcom}[\text{skip}]R = R
\]

\[
\text{Abcom}[X := A]R \downarrow \alpha\{\rho \mid \exists i \in I : \rho \vdash A \Rightarrow i \land \rho[X := i] \in \gamma(R)\}
\]

\[
\text{Abcom}[\text{if } B \text{ then } S_f \text{ else } S_f]R =
\]

\[
\text{Abexp}[B][\text{Abcom}[S]]R \cup \text{Abexp}[T(-B)](\text{Abcom}[S])R
\]

\[
\text{Abcom}[\text{while } B \text{ do } S \text{ od}]R =
\]

\[
\text{H}_\rho \lambda X : \text{Abexp}[T-(B)]R \cup \text{Abexp}[B](\text{Abcom}[S]X)
\]

\[
\text{Abcom}[C ; S]R = (\text{Abcom}[C] \circ \text{Abcom}[S])R
\]

\[
\text{Abcom}[S ; \text{;} ]R = \text{Abcom}[S]
\]

---

**Proof.**

The soundness condition is:

\[
\text{Bcom}[C] . X \subseteq \text{Abcom}[C]
\]

for Galois connection, or more generally:

\[
\text{Bcom}[C] . X \subseteq \text{Abcom}[C] . X
\]

in absence of best approximation and use of a concretization function only.

The proof is by structural induction on commands.
In case of a $\kappa$-based abstract interpretation, we would have

$\alpha(\text{Exp} \# (\text{Exp} \# (\text{Exp} [\text{Exp} \# (\text{Exp} \# \{X\} X))))$

$\equiv 2 \text{ induction hypothesis, } \alpha(\text{Exp} \# (\text{Exp} \# \{X\} X))$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\alpha(\text{Exp} \# (\text{Exp} \# \{X\} X))$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\alpha(\text{Exp} \# (\text{Exp} \# \{X\} X))$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\alpha(\text{Exp} \# (\text{Exp} \# \{X\} X))$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\alpha(\text{Exp} \# (\text{Exp} \# \{X\} X))$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\alpha(\text{Exp} \# (\text{Exp} \# \{X\} X))$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\equiv 2 \text{ or } \text{Exp} \# (\text{Exp} \# \{X\} X) \in \text{Exp} \# (\text{Exp} \# \{X\} X)$

$\alpha(\text{Exp} \# (\text{Exp} \# \{X\} X))$
so by the transfinite approximation, we conclude that

\[ x(\text{Scan} \oplus \text{while } \text{Scan } \text{do } \text{Loop } ) || x(R) \]

\[ = x ( \text{Scan} \oplus \text{while } \text{Scan } \text{do } \text{Loop } ) R || x(R) \]

\[ = \text{Eexp} || x. \text{Scan} \oplus (x) \oplus \text{Eexp} \ (\text{Scan} \oplus \text{Scan} \oplus \text{Scan}) \]

The case of a \( X \)-based abstraction is similar. We have:

\[ \text{Eexp} || (x) \oplus \text{Eexp} \ (\text{Scan} \oplus \text{Scan} \oplus \text{Scan}) \]

\[ \leq \text{Eexp (Scan)} \oplus \text{monotonic hypothesis on Eexp} \]

And conclude hypothesis S.
In the case of a $\mathcal{F}$-based abstractly, we have similarly:

\begin{align*}
\text{Blom} \left[ \text{GCD} \cdot \mathcal{L}_0 \right] \left( \mathcal{R}(x) \right) & = \text{Blom} \left[ \text{GCD} \cdot \left( \text{Blom} \left[ \mathcal{L}_0 \right] \left( \mathcal{R}(x) \right) \right) \right] \\
& \leq \text{I monad and ind. hyp.} \\
& \quad \text{I monad and ind. hyp.} \\
& \quad \text{I monad and ind. hyp.} \\
& = \mathcal{L}_0 \left( \mathcal{R}(x) \right) \\
& = \mathcal{L}_0 \left( \mathcal{R}(x) \right).
\end{align*}

\[\]
Linear syntax

We just have to add a function at mapping linearized commands to their entry point:
1 (* linear_Syntax.mli *)
2 open Abstract_Syntax
3 (* A linear arithmetic expression al.x1+...+an.xn+b, where n is the *)
4 (* number of program variables, is represented by a vector: *)
5 (* LINEAR_AEXP a1 ... an b. A non-linear arithmetic expression is *)
6 (* represented by RANDOM_AEXP. *)
7 type laexp =
8   | RANDOM_AEXP (* random expression *)
9   | LINEAR_AEXP of int array (* linear expression *)
10 and l1exp =
11   | LTRUE | LFALSE (* constant boolean expression *)
12   | RANDOM_BEXP (* random boolean expression *)
13   | LAND of l1exp list (* boolean conjunction *)

We just have to add a function at mapping linearized commands to their entry point:
1 (* linear_Syntax.mli *)
2 open Abstract_Syntax
3 (* A linear arithmetic expression al.x1+...+an.xn+b, where n is the *)
4 (* number of program variables, is represented by a vector: *)
5 (* LINEAR_AEXP a1 ... an b. A non-linear arithmetic expression is *)
6 (* represented by RANDOM_AEXP. *)
7 type laexp =
8   | RANDOM_AEXP (* random expression *)
9   | LINEAR_AEXP of int array (* linear expression *)
10 and l1exp =
11   | LTRUE | LFALSE (* constant boolean expression *)
12   | RANDOM_BEXP (* random boolean expression *)
13   | LAND of l1exp list (* boolean conjunction *)

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 * l1exp * l1exp * lcom * lcom * label
23   | LWHILE of label * l1exp * l1exp * lcom * label

val at : lcom -> label (* command entry label *)
25 val after : lcom -> label (* command exit label *)
26 val incom : label -> lcom -> bool (* label in command *)
27 (* linear_Syntax.mli *)
28 open Abstract_Syntax
29 (* A linear arithmetic expression al.x1+...+an.xn+b, where n is the *)
30 (* number of program variables, is represented by a vector: *)
backward relational abstract interpretation of commands

(* bcom.mli *)
open Linear_Syntax
open Aenv
(* backward abstract interpretation of commands *)
val bcom : lcom -> Aenv.t -> label -> Aenv.t

(* backward abstract semantics of commands *)
exception Error of string
The generic backward/precondition linear relational static analyzer

122 (* main.ml = main-bw.ml *)
123 open Program_To_Abstract_Syntax
124 open Labels
125 open Pretty_Print
126 open Lpretty_Print
127 open Abstract_To_Linear_Syntax
128 open Linear_Syntax
129 open Aenv
130 open Acom
131 open Bcom
132 let _ =
133 let arg = if (Array.length Sys.argv) = 1 then ""
134 else Sys.argv.(1) in

Polyhedral backward/precondition analysis

135 Random.self_init ();
136 let p = (abstract_syntax_of_program arg) in
137 (print_string "** Program:\n";
138 pretty_print p;
139 let p' = (linearize_com p) in
140 print_string "** Linearized program:\n";
141 lpretty_print p';
142 init ();
143 print_string "** Postcondition:\n";
144 print (top ());
145 print_string "** Precondition:\n";
146 print (bcom p' (top ()) (at p'));
147 quit ())

The polyhedral abstract environment for backward static analysis

– We just have to add the definition of b_ASSIGN to handle backward assignments:
148 (* aenv.mli *)
149 open Linear_Syntax
150 open Array
151 open Variables
152 (* set of environments *)
153 type t
154 (* relational library initialization *)
155 val init : unit -> unit
156 (* relational library exit *)
157 val quit : unit -> unit
158 (* infimum *)
159 val bot : unit -> t
160 (* check for infimum *)
161 val is_bot : t -> bool
162 (* uninitialization *)
163 val initerr : unit -> t
164 (* supremum *)
165 val top : unit -> t
166 (* least upper bound *)
167 val join : t -> t -> t
168 (* greatest lower bound *)
169 val meet : t -> t -> t
170 (* approximation ordering *)
171 val leq : t -> t -> bool
172 (* equality *)
173 val eq : t -> t -> bool
174 (* printing *)
175 val print : t -> unit
176 (* forward collecting semantics of assignment *)

177 (* f_ASSIGN x f r = {e[x <- i] | e in r \ i in f({e}) cap I} *)
178 val f_ASSIGN : variable -> laexp -> t ->...

194 open Linear_Syntax
195 type lattice = BOT | TOP
196 type t =
197 NULL of lattice (* in absence of any variable, dimension = 0 *)
198 | POLY of Poly.t (* must be of dimension > 0 *)
199 exception PolyError of string
200 (* relational library initialization *)
201 let maxdims = 10000
202 let maxrows = 100
203 let init () = (Polka.initialize false maxdims maxrows;
204 Polka.strict := false)
205 (* relational library exit (* print statistics *) *)
206 let quit () = Polka.finalize ()
207 (* infimum *)
208 let bot () = match (number_of_variables ()) with
209 | 0 -> NULL BOT
210 | n -> if (n < 0) then
211 raise (PolyError "negative number of variables (bot")

212 else if (n > maxdims) then
213 raise (PolyError "too many variables (bot")
214 else
215 POLY (Poly.empty n) (* 1 <= n <= polka_maxcolumns-polka_dec *)
216 (* check for infimum *)
217 let is_bot r = match r with
218 | NULL BOT -> true
219 | NULL TOP -> false
220 | POLY p -> (Polys.equal p (Poly.empty (number_of_variables ()))))
221 (* uninitialization *)
222 let initerr () = match (number_of_variables ()) with
223 | 0 -> NULL TOP
224 | n -> if (n < 0) then
225 raise (PolyError "negative number of variables (initerr")
226 else if (n > maxdims) then
227 raise (PolyError "too many variables (initerr")
228 else
229 POLY (Poly.universe n)
230 (* greatest lower bound *)
231 let lmeet l1 l2 = match (l1, l2) with
232 | TOP, _ -> l2
233 | _, TOP -> l1
234 | _ _, _ -> raise (PolyError "too many variables (top")
235 | else if (n > maxdims) then
236 raise (PolyError "negative number of variables (l1")
237 else
238 POLY (Poly.universe n)
239 (* least upper bound *)
240 let ljoin 11 12 = match (11, 12) with
241 | BOT, _ -> 12
242 | _, BOT -> 11
243 | _, _ -> TOP
244 let join r1 r2 = match (r1, r2) with
245 | NULL 11, NULL 12 -> (NULL (ljoin 11 12))
246 | POLY 1, POLY 1 -> (POLY (Poly.union p1 p2))
247 | _, _ -> raise (PolyError "join")

(* equality *)
266 let eq r1 r2 = match (r1, r2) with
268 | NULL 11, NULL 12 -> (11 = 12)
269 | POLY 1, POLY 1 -> (Poly.is_equal p1 p2)
270 | _ _, _ -> raise (PolyError "eq")
271 (* printing *)
272 let print r = match r with
273 | NULL B -> (print_string "{ \n")
274 | NULL T -> (print_string "T \n")
275 | POLY p ->
276 (Poly.minimize p; (* to get the constraints and generators of p *)
277 Poly.print_constraints string_of_variable Format.std_formatter p;
278 Format.pp_print_newline Format.std_formatter ()
279 (* convert a0.v0+...+an-1.vn-1+an where n = (number_of_variables () int
280 (* vector [1, an, a0, ..., an]
281 let vector_of_in_expr a =
282 let v = Vector.make ((number_of_variables ()) + 2) in
283 (Vector.set v 0 1;

248 (* greatest lower bound *)
249 let lmeet l1 l2 = match (l1, l2) with
250 | TOP, _ -> 12
251 | _, TOP -> 11
252 | _, _ -> BOT
253 let meet r1 r2 = match (r1, r2) with
254 | NULL 11, NULL 12 -> (NULL (lmeet 11 12))
255 | POLY 1, POLY 1 -> (POLY (Poly.inter 1 p1 p2))
256 | _, _ -> raise (PolyError "meet")
257 (* approximation ordering *)
258 let lleq 11 12 = match (11, 12) with
259 | BOT, _ -> true
260 | _, TOP -> true
261 | TOP, BOT -> false
262 let lq r1 r2 = match (r1, r2) with
263 | NULL 11, NULL 12 -> (lleq 11 12)
264 | POLY 1, POLY 1 -> (Poly.is_included_in 1 p1 p2)
265 | _, _ -> raise (PolyError "lq")

284 Vector.set v 1 (a.(number_of_variables ());
285 for i = 0 to ((number_of_variables ()) - 1) do
286 Vector.set v (i+2) a.(i)
287 done;
288 (*
289 Vector.print v;
290 Vector.print_constraint string_of_variable Format.std_formatter v;
291 Format.pp_print_newline Format.std_formatter (;
292 *)
293 v)
294 (* f_ASSIGN x f r = {e[x <- i] | e in r \ i in f({e}) cap I } *)
295 let f_ASSIGN x f r =
296 match r with
297 | NULL _ -> r
298 | POLY p -> (match f with
299 | RANDOM_AEXP ->
300 let d = [(Polka.pos = x; Polka.ndims = 1)\] in
301 (POLY (Poly.add_dims_and_embed_multi (Polka.del_dims_multi p d) d))]
Examples of backward polyhedral static analysis

Generic-FW-BW-REL-Abstract-Interpreter % ./a.out
  x := 1;
  while (x > 0) do
    x := x + 10000000;
    od;
  ...
  ** Postcondition:
  {i >= 0}
  ** Precondition:
  empty(1)

  - The only way for the program execution to exit the loop is to never start it (because of arithmetic overflow when \( \mathbb{I} \) is unbounded or non-termination if \( N \subseteq \mathbb{I} \))

The precondition/postcondition collecting semantics

Examples of backward polyhedral static analysis
Definition of the forward/backward semantics

- We assume that a transition system $\langle \Sigma, \tau, I, F \rangle$ is given such that
  - $\Sigma$ is a set of states
  - $\tau \in \wp(\Sigma \times \Sigma)$ is the transition relation between a state and its potential successors
  - $I \subseteq \Sigma$ is the set of initial states
  - $F \subseteq \Sigma$ is the set of final states

- The backward or precondition semantics is the set of states from which the final states $F$ may be potentially reached
  \[
  \text{pre}[\tau^*]F = \{ s \in \Sigma \mid \exists s' \in F : \tau^*(s, s') \}
  \]
  \[
  = \text{lfp}_B \mathcal{F}
  \]
  where $\mathcal{F}(X) = \mathcal{I} \cup \text{pre}[^t]X$
  and $\text{pre}[^t]X = \{ s \in \Sigma \mid \exists s' \in X : t(s, s') \}$

- The forward or postcondition semantics is the set of states which are reachable from the initial states $I$
  \[
  \text{post}[\tau^*]I = \{ s' \in \Sigma \mid \exists s \in I : \tau^*(s, s') \}
  \]
  \[
  = \text{lfp}_0 \mathcal{F}
  \]
  where $\mathcal{F}(X) = \mathcal{I} \cup \text{post}[^t]X$
  and $\text{post}[^t]X = \{ s' \in \Sigma \mid \exists s \in X : t(s, s') \}$
– The forward/backward or precondition/postcondition semantics is the pair
\[ \langle I \cap \text{pre}[\tau^*] F, \text{post}[\tau^*] I \cap F \rangle \]

– In the concrete, there is nothing new
– However, in the abstract, we can do much better than separate overapproximations of \(\text{pre}[\tau^*] F\) and \(\text{post}[\tau^*] I\)

Example of abstract forward/backward semantics

```
Generic-FW-BW-REL-Abstract-Interpreter % make bw
Backward analysis
Generic-FW-BW-REL-Abstract-Interpreter % make pol
Polyhedral analysis
Generic-FW-BW-REL-Abstract-Interpreter % ./a.out ../Examples/example48.sil
** Postcondition:
{1>=0}
** Precondition:
{X+2Y>=2}
Generic-FW-BW-REL-Abstract-Interpreter % make iter
Iterated forward/backward analysis
Generic-FW-BW-REL-Abstract-Interpreter % make pol
Generic-FW-BW-REL-Abstract-Interpreter % ./a.out ../Examples/example48.sil
** Postcondition:
{X+2Y>=2}
** Precondition:
{X+Y=5}
```

Properties of the forward/backward collecting semantics

**THEOREM.** \( \langle I \cap \text{pre}[\tau^*] F, F \cap \text{post}[\tau^*] I \rangle = \text{gfp}^{\leq 2} \lambda (X, Y). \langle I, F \rangle \cap_2 (X \cap \text{pre}[\tau^*] Y, Y \cap \text{post}[\tau^*] X) \)

**PROOF.** We prove that \( \langle I \cap \text{pre}[\tau^*] F, F \cap \text{post}[\tau^*] I \rangle \) is a fixpoint of \(\lambda (X, Y). \langle I, F \rangle \cap_2 (X \cap \text{pre}[\tau^*] Y, Y \cap \text{post}[\tau^*] X)\) componentwise. For the first component, we have

\[
I \cap (I \cap \text{pre}[\tau^*] F) \cap \text{pre}[\tau^*] (F \cap \text{post}[\tau^*] I)
\]

```prolog
\{\text{def. pre}\}
= I \cap \text{pre}[\tau^*] F \cap \{s | \exists s_2 : \tau^*(s, s_2) \land s_2 \in F \land s_2 \in \text{post}[\tau^*] I\}
\{\text{def. post}\}
= I \cap \text{pre}[\tau^*] F \cap \{s | \exists s_2 : \tau^*(s, s_2) \land s_2 \in F \land \exists s_3 : s_3 \in I \land \tau^*(s_3, s_2)\}
```
The precondition/postcondition abstract semantics

- We are given an abstraction \( \langle \rho(\Sigma), \subseteq \rangle \xrightarrow{\gamma} \langle L, \subseteq \rangle \)
- We want to compute an overapproximation of:

\[
\langle \alpha(I \cap \text{pre}[\tau^*]F), \alpha(F \cap \text{post}[\tau^*]I) \rangle = \langle \alpha(I \cap \text{lfp} \lambda Z . F \cup \text{pre}[\tau]Z), \alpha(F \cap \text{lfp} \lambda Z . I \cup \text{post}[\tau]Z) \rangle
\]

- One solution is to overapproximate

\[
\langle \alpha(I) \cap \text{lfp} \lambda Z . \alpha(F) \cup \alpha(\text{pre}[\tau](\gamma(Z)), \alpha(F) \cap \text{lfp} \lambda Z . \alpha(I) \cup \alpha(\text{post}[\tau]\gamma(Z)) \rangle
\]

- A better solution is to use the greatest fixpoint characterization and to overapproximate:

\[
\gamma_2(\text{gfp} \subseteq \lambda(X, Y) \cdot \langle I, F \rangle \cap_2 \langle X \cap \text{pre}[\tau^*]Y, Y \cap \text{post}[\tau^*]X \rangle)
\]

\[
\langle \text{fixpoint overapproximation} \rangle
\]

\[
\subseteq_2 \text{gfp} \subseteq \lambda(X, Y) \cdot \langle \alpha(I), \alpha(F) \rangle \cap_2
\]

\[
\langle \alpha(\gamma(X) \cap \text{pre}[\tau^*]\gamma(Y)), \alpha(\gamma(Y) \cap \text{post}[\tau^*]\gamma(X)) \rangle
\]

\[
\langle \text{fixpoint definition of pre[\tau^*]} \rangle
\]

\[
= \text{gfp} \subseteq \lambda(X, Y) \cdot \langle \alpha(I), \alpha(F) \rangle \cap_2
\]

\[
\langle \alpha(\gamma(X) \cap \text{lfp} \lambda Z \cdot \gamma(Y) \cup \text{pre}[\tau]Z),
\alpha(\gamma(Y) \cap \text{lfp} \lambda Z \cdot \gamma(X) \cup \text{post}[\tau]Z) \rangle
\]
{\alpha \text{ monotone}}
\square \{ \mu \in L_\varnothing \} \implies \mu \subseteq \lambda (X, Y) : \langle \alpha(I), \alpha(F) \rangle
\begin{align*}
\quad & \langle \alpha(\gamma(X)) \cap \mu Z . \alpha(\gamma(Y)) \cup \alpha(\text{pre}[\tau](\gamma(Z))), \\
\quad & \quad \alpha(\gamma(Y)) \cap \lambda Z . \alpha(\gamma(X)) \cup \alpha(\text{post}[\tau](\gamma(Z))) \rangle \\
\quad & \langle \alpha \circ \gamma \text{ reductive and monotony} \rangle
\end{align*}
\square \{ \mu \in L_\varnothing \} \implies \mu \subseteq \lambda (X, Y) : \langle \alpha(I), \alpha(F) \rangle \cap \mu
\begin{align*}
\quad & \langle X \cap \mu Z . Y \cup \alpha(\text{pre}[\tau](\gamma(Z))), \\
\quad & \quad Y \cap \lambda Z . X \cup \alpha(\text{post}[\tau](\gamma(Z))) \rangle
\end{align*}

Pre/postcondition static analysis

Assuming $\alpha(I) \subseteq I^\#, \alpha(F) \subseteq F^\#, \alpha \circ \text{pre}[\tau] \circ \gamma \subseteq B^\#$,
$\alpha \circ \text{post}[\tau] \circ \gamma \subseteq F^\#$, and using chaotic iterations, the algorithm is

\begin{align*}
X^0 &= I^# \\
Y^0 &= F^# \\
\ldots \ldots \\
X^{n+1} &= X^n \cap \mu Z . Y^n \cup B^#(Z) \\
Y^{n+1} &= Y^n \cap \mu Z . X^{n+1} \cup F^#(Z) \\
\ldots \ldots 
\end{align*}

Convergence may have to be enforced via a narrowing.

Implementation of the pre/postcondition static analysis

In the implementation the initial precondition is that variables are uninitialized and the postcondition is $\top$, that is $T$ is both cases.

```plaintext
let _ = 
let narrowing_limit = 100 in 
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 (); 
let rec iterate pre n = 
  (print_string "** Precondition:\n"; 
   print pre; 
   let post = (acom p' pre (after p')) in 
   (print_string "** Postcondition:\n"; 
    print post; 
```
Examples of pre/postcondition static analysis

In the examples, we enforce a postcondition $B$ as:

```plaintext
while :B do skip ...
```

which, since termination is enforced by the backward analysis, requires $B$ to hold just before the loop.

**Program:**

```plaintext
0: x := y;
1: z := (2 * x + 1);
2: while ((y < 0) | (0 < y)) do
3:   skip
4:   od {(y = 0)}
5:
```

**Linearized program:**

```plaintext
0: x := 1.y + 0.x + 0.z + 0;
1: z := 0.y + 2.x + 0.z + 1;
2: while (-1.y + 0.x + 0.z + -1 >= 0 | 1.y + 0.x + 0.z + -1 >= 0) do
3:   skip
4:   od {-1.y + 0.x + 0.z + 0 = 0}
5:
```
** Precondition:
{1>=0}
** Postcondition:
{z=1,x=0,y=0}
unstable precondition after 1 iteration(s).
** Precondition:
{y=0}
** Postcondition:
{z=1,x=0,y=0}
stable precondition after 2 iteration(s).

Comments on this example:
- The initial precondition \(1>=0\) is \(\top\) whence states no hypothesis on the input data
- The forward analysis provides the first postcondition stating that on termination \(y=0\) whence \(z=1\) and \(x=0\)
- Since \(y\) is not modified in the program, the next backward analysis provides the second precondition that we must have \(y=0\) initially
- The last forward iteration shows that the fixpoint is reached

The following program is proved to never terminate:

Generic-FW-BW-REL-Abstract-Interpreter % ./a.out
\[x := 1;\]
while \((x > 0)\) do
\[x := x + 1000000\]
\[od;\]
** Precondition:
{1>=0}
** Postcondition:
empty(1)
unstable precondition after 1 iteration(s).
** Precondition:
empty(1)
** Postcondition:
empty(1)
stable precondition after 2 iteration(s).

Generic-FW-BW-REL-Abstract-Interpreter % cat ../Examples/example45.sil
\% example45.sil
\% x := x + y;
while \((x + y <> 0)\) do
\[skip\]
\[od;\]
Generic-FW-BW-REL-Abstract-Interpreter % ./a.out ../Examples/example45.sil
** Precondition:
{1>=0}
** Postcondition:
\{x+y=0\}
unstable precondition after 1 iteration(s).
** Precondition:
\{x+2y=0\}
** Postcondition:
\{x+y=0\}
stable precondition after 2 iteration(s).
Generic-FW-BW-REL-Abstract-Interpreter % cat ../Examples/example46.sil
% example46.sil %
X := Y;
while (X + Y + Z <> 0) do
  skip
od;;

Generic-FW-BW-REL-Abstract-Interpreter % ./a.out ../Examples/example46.sil
** Precondition:
{1}<>0
** Postcondition:
{2Y+Z=0,Y=X}
unstable precondition after 1 iteration(s).
** Precondition:
{2Y+Z=0}
** Postcondition:
{2Y+Z=0,Y=X}
stable precondition after 2 iteration(s).

The forward/backward collecting semantics

The forward reachability collecting semantics

\[ \text{post}[\tau^*]I = \{x \in \mathcal{L} X. I \cup \text{post}[\tau]X \}\]

is the set of descendants of the initial states \( I \) by transitions \( \tau \).
Recall the following fixpoint characterization of the forward reachability collecting semantics:

**Theorem.** \( \mathcal{D} = \text{post}[\tau \star] \mathcal{J} = \text{llfp} F[p] \) where \( F[p] \in \langle \rho(S), \cup \rangle \) \( \leadsto \langle \rho(S), \cup \rangle \) is defined by \( F[p] X = J \cup \text{post}[\tau X]. \)

**Proof.** Observe that \( \rho(\bar{S} \times \bar{S}) \subseteq \bar{S} \times \bar{S} \cup \bar{S} \cup \bar{S} \) and \( \rho(\bar{S}) \subseteq \bar{S} \cup \bar{S} \cup \bar{S} \) are complete lattices. Define \( \alpha \in \rho(\bar{S} \times \bar{S}) \mapsto \rho(\bar{S}) \) by \( \alpha(X) = \text{post}[X]. \) It is a complete \( \cup \)-morphism so that there exists \( \gamma \) such that \( \rho(\bar{S} \times \bar{S})(\gamma) = \gamma \rho(\bar{S}). \) We have \( \tau \star = \text{llfp} T = \bigcup_{n \in \mathbb{N}} T^n(\emptyset) \) where \( T(X) = 1 \cup X \circ \tau, \emptyset = \alpha(\emptyset) \) and \( F[p] \in \langle \rho(S), \cup \rangle \mapsto \langle \rho(S), \cup \rangle \) is such that for all \( X \in \rho(\bar{S} \times \bar{S}), \) we have \( \alpha \circ T(X) = \text{post}[T(X)]J = \{ s | \exists s' \in J : (s', s) \in T(X) \} = \{ s | \exists s' \in J : (s', s) \in T(X) \} = \{ (s, s) \in 1 \cup X \circ \tau = \{ (s, s) \in 1 \cup X \circ \tau \circ \alpha = \{ (s, s) \in 1 \cup X \circ \tau \}. \)

The forward/backward reachability collecting semantics

- We are interested in the forward/backward collecting semantics which is defined as \( \text{post}[\tau \star] J \cap \text{pre}[\tau \star] F \) that is the intersection of two fixpoints \( \text{llfp} \subseteq \lambda X \cdot J \cup \text{post}[\tau X] \cap \lambda Y \cdot F \cup \text{pre}[\tau Y] \)

- We look for a more precise analysis than the mere intersection of the independent overapproximations of \( \text{llfp} \subseteq \lambda X \cdot J \cup \text{post}[\tau X] \) and \( \text{llfp} \subseteq \lambda Y \cdot F \cup \text{pre}[\tau Y] \)

**The backward reachability collecting semantics**

- The backward reachability semantics
  
  \[ \text{pre}[\tau \star] F = \text{llfp} \subseteq \lambda Y \cdot F \cup \text{pre}[\tau Y] \]

  is the set of potential ancestors of the final states \( F \) by transitions \( \tau \)
Properties of the forward/backward collecting semantics

**Theorem.** For all transition systems \((\Sigma, I, F, \tau)\) where \(F(X) \overset{\text{def}}{=} I \cup \text{post}[\tau]X, B(X) \overset{\text{def}}{=} I \cup \text{pre}[\tau]X,\) and \(X \subseteq \Sigma,\) we have:

\[
\begin{align*}
(p_{\text{pre}}(\tau)X) \cap \text{lfp } F &\subseteq \text{p}_{\text{pre}}(X \cap \text{lfp } F) \quad (16) \\
(p_{\text{post}}(\tau)X) \cap \text{lfp } B[B] &\subseteq \text{p}_{\text{post}}(X \cap \text{lfp } B) \quad (17) \\
\text{lfp } F \cap \text{lfp } B &\subseteq \text{p}_{\text{pre}}(\tau)(X \cap \text{lfp } F) \\
= \text{lfp } \lambda X \cdot (\text{lfp } F \cap B(X)) \quad (18) \\
= \text{lfp } \lambda X \cdot (\text{lfp } B \cap F(X)) \quad (19) \\
= \text{lfp } \lambda X \cdot (\text{lfp } F \cap \text{lfp } B \cap F(X)) \quad (20) \\
= \text{lfp } \lambda X \cdot (\text{lfp } F \cap \text{lfp } B \cap F(X)) \quad (21)
\end{align*}
\]

**Proof.** To prove (16), observe that the theorem on page 113 implies that \((p_{\text{pre}}(\tau)X) \cap \text{lfp } F = \{ s \mid \exists s' \in X : s' \subseteq s \} \cap \{ s \mid \exists s'' \in I : s'' \subseteq s'' \} \subseteq \{ s \mid \exists s' \in X : \exists s'' \in I : s'' \subseteq s'' \} \). Since \(s'' \subseteq s''\) implies \(s'' \subseteq s''\), this is precisely \(p_{\text{pre}}(X \cap \text{lfp } F) \). The proof of (17) is similar to that of (16).

To prove (19), let \(X^n, n \in \mathbb{N}\) and \(Y^n, n \in \mathbb{N}\) be the iteration sequences starting from the infimum \(\emptyset = 0\) and \(\text{lfp } \lambda X \cdot (\text{lfp } F \cap B)[X]\) respectively. We have \(\text{lfp } F \cap X^n = \emptyset = 0\). Assume that \(\text{lfp } F \cap X^n \subseteq Y^n\) by induction hypothesis. Then \(\text{lfp } F \cap X^{n+1} = \text{lfp } F \cap B(X^n) = \text{lfp } F \cap (p_{\text{pre}}(\tau)X^n \cup F) = \text{lfp } F \cap (p_{\text{pre}}(X^n \cap \text{lfp } F) \cup F),\) which, by (16), is included in \(\text{lfp } F \cap (p_{\text{pre}}(X^n \cap \text{lfp } F) \cup F)\) which, by induction hypothesis and monotony, is included in \(\text{lfp } F \cap (p_{\text{pre}}(X^n \cap F) \cup F) = \text{lfp } F \cap BY^n = Y^{n+1}\). It follows that \(\text{lfp } F \cap \text{lfp } B = (\cup_{n \in \mathbb{N}} X^n) \cap \text{lfp } F = \cup_{n \in \mathbb{N}} (X^n \cap \text{lfp } F) = \cup_{n \in \mathbb{N}} Y^n = \text{lfp } \lambda X \cdot (\text{lfp } F \cap B(X)).\) But \(\text{lfp } \lambda X \cdot (\text{lfp } F \cap B(X)) \subseteq \text{lfp } \lambda X \cdot (\text{lfp } F) \cap \text{lfp } \lambda X \cdot B(X) = \text{lfp } F \cap \text{lfp } B\). Equality follows by antisymmetry. The proofs of (20) to (22) are similar.  

The forward/backward reachability abstract semantics

Let us recall the following fixpoint approximation result:

**Theorem.** If \(P(\leq, f, t, \wedge, \vee)\) is a complete lattice, \(F, F' \in P(\leq) \overset{\text{m}}{\rightarrow} P(\leq)\), and \(F \leq F'\), then \(\text{lfp } F \leq \text{lfp } F'.\)

**Proof.** We have \(F(\text{lfp } F) \leq F(\text{lfp } F') = \text{lfp } F'\) whence \(\text{lfp } F \leq \text{lfp } F'\) since \(\text{lfp } F = \bigwedge\{X \mid F(X) \subseteq X\}\) by Tarski’s fixpoint theorem.

as well as fixpoint abstraction:

**Theorem.** If \(P(\leq, f, t, \wedge, \vee)\) and \(P(\leq, f, t, \wedge, \vee)\) are complete lattices, \(P(\leq) \overset{\text{m}}{\rightarrow} P(\leq)\) and \(F \in P(\leq)\), then \(\alpha(\text{lfp } F) \leq \text{lfp } \alpha \circ F \circ \gamma\).

\[\alpha(\text{lfp } F) \leq \text{lfp } \alpha \circ F \circ \gamma.\]
Proof. By Tarski’s fixpoint theorem, the least fixpoints exist. So let \( p^g = \text{lfp} \circ F \circ \gamma \). We have \( \alpha \circ F \circ \gamma(p^g) = p^g \) whence \( F \circ \gamma(p^g) \leq \gamma(p^g) \) by def. Galois connection. It follows that \( \gamma(p^g) \) is a postfixpoint of \( F \) whence \( \text{lfp} F \leq \gamma(p^g) \) by Tarski’s fixpoint theorem or equivalently \( \alpha(\text{lfp} F) \leq p^g = \text{lfp} \alpha \circ F \circ \gamma \).

\[
\begin{align*}
  _X0 &= \text{lfp} F^\parallel \\
  _X2n+1 &= \text{lfp} \lambda X. _X2n \land B^\parallel(X) \\
  _X2n+2 &= \text{lfp} \lambda X. _X2n+1 \land B^\parallel(X)
\end{align*}
\]

for all \( n \in \mathbb{N} \).

– Observe that by theorem on page 117 there is no improvement when considering the exact collecting semantics.

– However, when considering approximations of the collecting semantics, not all information can be collected in one pass and iteration can lead definite improvement.

– If the abstract lattice does not satisfy the descending chain condition then \cite{2} also suggests to use a narrowing operator \( \Delta \) to enforce convergence of the downward iteration \( X^k \), \( k \in \mathbb{N} \).

– The same way a widening/narrowing approach can be used to enforce convergence of the iterates for \( \lambda X. _X2n \land B^\parallel(X) \) and \( \lambda X. _X2n+1 \land B^\parallel(X) \).

– The correctness of this approach follows from the following result on fixpoint meet approximation:
We have $\alpha(\lfp F \land \lfp B) \preceq \alpha(\lfp F)$ since $\alpha$ is monotone and $\alpha(\lfp F) \preceq \lfp F^k$ by theorems on pages 120 and 120, thus proving the proposition for $k = 0$. Let us observe that $\alpha \circ F \circ \gamma \preceq F^k \circ \alpha$. Assume now by induction hypothesis that $\alpha(\lfp F \land \lfp B) \preceq \gamma^k(X^{2n})$, whence $\lfp F \land \lfp B \preceq \gamma(X^{2n})$ by def. Galois connection so that in particular for an argument of the form $\alpha(X), F \circ \gamma \preceq \gamma \circ F^k \circ \alpha$. Assume now by induction hypothesis that $\alpha(\lfp F \land \lfp B) \preceq \gamma(X^{2n})$, whence $\lfp F \land \lfp B \preceq \gamma(X^{2n})$ by def. Galois connection.

We have $\lfp F \circ \gamma \preceq \gamma \circ F^k \circ \alpha$. Assume now by induction hypothesis that $\alpha(\lfp F \land \lfp B) \preceq \gamma(X^{2n})$, whence $\lfp F \land \lfp B \preceq \gamma(X^{2n})$ by def. Galois connection. Since $\gamma \preceq \gamma \circ F^k \circ \alpha$, it follows that $\lambda X \cdot \lfp F \land \lfp B \circ \gamma \circ F^k \circ \alpha(X) = \lambda X \cdot \gamma(X^{2n} \land \lfp F \circ \alpha(X))$ since $\gamma$ is a complete meet morphism. Now by hypothesis (21) of theorem, we have $\lfp F \land \lfp B = \lfp \lambda X \cdot \lfp F \land \lfp B \circ \gamma \circ F^k \circ \alpha(X))$ by theorem on page 120. Let $G$ be $\lfp \lambda X \cdot X^{2n} \land \lfp F \circ \alpha(X)$ extentive so that by monotony $G \circ \alpha \circ \gamma \preceq \gamma G$ and $\alpha \circ \gamma \circ G \preceq \gamma \circ F^k \circ \alpha \circ \gamma \preceq \gamma G$. By the theorem on page 120 we have $\alpha(\lfp F \land \lfp B) \preceq \lfp F \land \lfp B \circ \gamma \circ F^k \circ \alpha(X)$ by hypothesis (22) of theorems.

**Example**

- Let us consider the original example given in [2], with interval analysis, so that a widening/narrowing is required:

  - $p^1 = \lfp F(p(x))$
  - $p^2 = p^1 \times k^1$
  - $\cdots$
  - $p^{k+1} = p^k \times k^{k+1}$
  - $p^{k+2} = p^k \times k^2$
  - $\cdots$

- We illustrate on bubble sort [Z. Manna, 1974, p. 191]

- The analysis in [2] is not structural but equational and so the forward system of equations is:

- The backward system of equations is:
- The first forward iteration is:

\[ P_1 = P_1(1 \times \{w, \ldots \}, n + 1) \]

\[ P_2 = P_2 \]

\[ P_3 = P_3 \]

\[ P_4 = P_4 \]

\[ P_5 = P_5 \]

\[ P_6 = P_6 \]

\[ P_7 = P_7 \]

\[ P_8 = P_8(1 \times \{j = 6, \ldots \}) \]

\[ P_9 = P_9 \]

\[ P_{10} = P_{10} \]

\[ P_{11} = P_{11} \]

- The second iteration is:

\[ P^2 = P^2 \cdot \mathcal{O} \quad \alpha^3 \cdot \mathcal{O}(\mathcal{L}(X), I(X)) \]

\[ \mathcal{L}(X) = \{(1 \times \{j = 3, \ldots \}, 0, 1, 0, 1) \} \]

- The considered program specification is:

\[ \mathcal{L}(X) = \{(i = \{i = 0, \ldots \}, j = \{j = 0, \ldots \}) \} \]

Since \( i \) is decremented by 1 in the loop which ends with \( i = 0 \), \( i \) must necessarily be positive on loop entry, so \( n \) must be initially positive.

- The third iteration is:

\[ \mathcal{L}(X) = \{(i = \{i = 0, \ldots \}, j = \{j = 0, \ldots \}) \} \]

This propagates forward the information on \( n \)

- The next iteration shows that a fixpoint is reached

- The analysis proves that if \( n \) is initially negative then the program execution either terminates by an error or does not terminate.

- By propagating forward the invariants, one determines where execution tests have to be placed.

- With polyhedral analysis, the result is the following:

```
Generic-FW-BW-REL-Abstract-Interpreter % cat ..../Examples/example30.sil
i := n;
while (i <> 1) do
    j := 0;
    while (j <> i) do
        j := j + 1
    od;
    i := i - 1
od;
```
Generic-FW-BW-REL-Abstract-Interpreter % ./a.out ../Examples/example30.sil

** Precondition:
{1>=0}

** Postcondition:
{i=1,n>=1}

unstable precondition after 1 iteration(s).

** Precondition:
{n=1}

** Postcondition:
{i=1,n=1}

stable precondition after 2 iteration(s).

---

** Optimality

** Lemma 1.** Let $P^o(\Sigma, \lambda, \gamma, \tau, \iota, \mu)$ and $P^o(\Sigma, \lambda, \gamma, \tau, \iota, \mu)$ be complete lattices with a Galois connection $P^o(\Sigma, \lambda, \gamma, \tau, \iota, \mu) \leq P^o(\Sigma, \lambda, \gamma, \tau, \iota, \mu)$ and $P^o(\Sigma, \lambda, \gamma, \tau, \iota, \mu)$ in $P^o(\Sigma, \lambda, \gamma, \tau, \iota, \mu) \leq P^o(\Sigma, \lambda, \gamma, \tau, \iota, \mu)$ be two monotonic functions, and let

$$L^o = \text{gfp}(\lambda Z \in \mathcal{P}(\Sigma) . \left( F^o(\Sigma) \uplus B^o(\Sigma) \right))$$

If $F^o \in P^o(\Sigma, \lambda, \gamma, \tau, \iota, \mu)$ is monotonic and satisfy $\sigma = F^o \circ \gamma \leq F^o$ and $\mu = B^o \circ \tau \geq B^o$, then the sequence $(X_n)_{n \in \mathbb{N}}$ defined by $X_0 = \tau \lambda, X_{n+1} = X_n \circ F^o(X_n)$, and $X_\infty = X_n \circ F^o(X_n)$, we have $L^o \subseteq X_\infty$ and such that:

$$\forall n \geq 0, \alpha(L^o) \subseteq X_\infty \subseteq X_n$$

The optimality of this approach has been proved in [1]; it has been shown that $L^o = \text{gfp}(\lambda Z \in \mathcal{P}(\Sigma) . \left( F^o(\Sigma) \uplus B^o(\Sigma) \right))$ is the greatest lower bound of the set $E^o$ defined inductively as:

- $\tau \in E^o$
- If $Z \in E^o$ then so are $F^o(Z)$ and $B^o(Z)$.
- If $Z_1$ and $Z_2$ are in $E^o$ then so are $Z_1 \circ \gamma Z_2$ and $Z_1 \uplus Z_2$.

Therefore, $L^o$ is the best upper approximation of $\alpha(L^o)$ that can be obtained using $F^o$ and $B^o$.

---

** Standard backward-forward combination**

The standard backward-forward combination [4] derives from an application of this lemma to the particular case of backward and forward collecting semantics:

$$F^o, B^o, F^d, B^d$$

and $F^o, B^o, F^d, B^d$ are instantiated as follows:

$$F^o = \lambda X . (Y \uplus f(X))$$
$$B^o = \lambda X . (Y \uplus b(X))$$
$$F^d = \lambda X . (Y \uplus f(X))$$
$$B^d = \lambda X . (Y \uplus b(X))$$

where gfp means either lfp or gfp, and $f, b \in P^o \rightarrow P^o, f, b \in P^o \rightarrow P^d$ are monotonic. When $\sigma = f \circ \gamma \leq f$ and $\mu = b \circ \tau \geq b$, the conditions of Lemma 1 are satisfied [7].

---

Now, let $F^d = \rho(\Sigma)$, $\Sigma$ a set of states, and $\tau \in \rho(\Sigma \times \Sigma)$ a transition relation. As usual, we define $\text{pre}, \text{pre}_r, \text{post}, \text{post}_r$ as:

$$\text{post}(X) = \{s' \mid \exists s : (s, s') \in \tau \land s \in X\}$$
$$\text{pre}_r(X) = \{s \mid \forall s' : (s, s') \in \tau \land s' \in X\}$$
$$\text{pre}(X) = \{s \mid \exists s' : (s, s') \in \tau \land s' \in X\}$$

Given $I, F \subseteq \Sigma$, sets of initial and final states, $f^o = \lambda X . (I \uplus \text{post}(X))$ and $b^o = \lambda X . (F \uplus \text{pre}(X))$ (and $\text{gfp}_1 = \text{gfp}_2 = \text{gfp}_3$), we have [4]:

$$I^o = F^o(I^o \uplus B^o(\Sigma)) = \text{gfp}_1 F^o \cap \text{gfp}_2 B^o$$

By computing $\gamma(I^o)$, we obtain a good upper approximation of $F^o(\Sigma) \cap B^o(\Sigma)$ (at least equal to $\gamma(F^o(\Sigma) \cap B^o(\Sigma))$).
The ASTRÉE static analyzer

ASTRÉE: A Sound, Automatic, Specializable, Domain-Aware, Parametric, Modular, Efficient and Precise Static Program Analyzer

www.astree.ens.fr

- C programs:
  - with
    - pointers (including on functions), structures and arrays
    - floating point computations
    - tests, loops and function calls
    - limited branching (forward goto, break, continue)

- without
  - union
  - dynamic memory allocation
  - recursive function calls
  - backward branching
  - conflict side effects
  - C libraries

- Application Domain: safety critical embedded real-time synchronous software for non-linear control of very complex control/command systems.

Concrete Operational Semantics

- International norm of C (ISO/IEC 9899:1999)
- restricted by implementation-specific behaviors depending upon the machine and compiler (e.g. representation and size of integers, IEEE 754-1985 norm for floats and doubles)
- restricted by user-defined programming guidelines (such as no modular arithmetic for signed integers, even though this might be the hardware choice)
- restricted by program specific user requirements (e.g. assert)
Abstract Semantics

- Trace-based refinement of the reachable states for the concrete operational semantics
- Volatile environment is specified by a trusted configuration file.

Implicit Specification: Absence of Runtime Errors

- No violation of the norm of C (e.g. array index out of bounds)
- No implementation-specific undefined behaviors (e.g. maximum short integer is 32767)
- No violation of the programming guidelines (e.g. static variables cannot be assumed to be initialized to 0)
- No violation of the programmer assertions (must all be statically verified).

Example application

- Primary flight control software of the A340/A380 fly-by-wire system
  ![A340/A380](image)
- C program, automatically generated from a proprietary high-level specification (à la Simulink/SCADé)
- A340 family: 132,000 lines, 75,000 LOCs after preprocessing, 10,000 global variables, over 21,000 after expansion of small arrays
- A380: ☒ 3

The Class of Considered Periodic Synchronous Programs

declare volatile input, state and output variables;
initialize state and output variables;
loop forever
  - read volatile input variables,
  - compute output and state variables,
  - write to volatile output variables;
  wait_for_clock();
end loop

- **Requirements:** the only interrupts are clock ticks;
- Execution time of loop body less than a clock tick [1].

Reference

Characteristics of the ASTRÉE Analyzer

Static: compile time analysis (≠ run time analysis Rational Purify, Parasoft Insure++)

Program Analyzer: analyzes programs not micromodels of programs (≠ PROMELA in SPIN or Alloy in the Alloy Analyzer)

Automatic: no end-user intervention needed (≠ ESC Java, ESC Java 2)

Sound: covers the whole state space (≠ MAGIC, CBMC) so never omit potential errors (≠ UNO, CMC from coverity.com) or sort most probable ones (≠ Splint)

Characteristics of the ASTRÉE Analyzer (Cont’d)

Multiabstraction: uses many numerical/symbolic abstract domains (≠ symbolic constraints in Bane or the canonical abstraction of TVLA)

Infinitary: all abstractions use infinite abstract domains with widening/narrowing (≠ model checking based analyzers such as VeriSoft, Bandera, Java PathFinder)

Efficient: always terminate (≠ counterexample-driven automatic abstraction refinement BLAST, SLAM)

Specializable: can easily incorporate new abstractions (and reduction with already existing abstract domains) (≠ general-purpose analyzers PolySpace Verifier)

Domain-Aware: knows about control/command (e.g. digital filters) (as opposed to specialization to a mere programming style in C Global Surveyor)

Parametric: the precision/cost can be tailored to user needs by options and directives in the code

Automatic Parametrization: the generation of parametric directives in the code can be programmed (to be specialized for a specific application domain)

Modular: an analyzer instance is built by selection of OCAML modules from a collection each implementing an abstract domain

Precise: very few or no false alarm when adapted to an application domain \(\rightarrow\) it is a VERIFIER!
Benchmarks (A340 Primary Flight Control Software)
- 132,000 lines, 75,000 LOCs after preprocessing
- Comparative results (commercial software):
  4,200 (false?) alarms,
  3.5 days;
- Our results, November 2003:
  0 alarms,
  40mn on 2.8 GHz PC,
  300 Megabytes
  → A world première!

(A380 Primary Flight Control Software)
- 350,000 lines
- 0 alarms (mid-October 2004!),
  7h on 2.8 GHz PC,
  1 Gigabyte
  → A world grand première!

Examples of Abstractions
General-Purpose Abstract Domains: Intervals and Octagons

**Intervals:**
\begin{align*}
1 & \leq x \leq 9 \\
1 & \leq y \leq 20
\end{align*}

**Octagons:**
\begin{align*}
1 & \leq x < 9 \\
x + y & \leq 77 \\
1 & \leq y \leq 20 \\
x - y & \leq 04
\end{align*}

**Difficulties:** many global variables, arrays (smashed or not), IEEE 754 floating-point arithmetic (in program and analyzer)

---

Floating-Point Computations

- **Code Sample:**

```c
/* float-error.c */
int main ()
{
float x, y, z, r;
x = 1.000000019e+38;
y = x + 1.0e21;
z = x - 1.0e21;
r = y - z;
printf("f\n", r);
} % gcc float-error.c
% .a.out
0.000000
```

\[(x + a) - (x - a) \neq 2a\]

---

Floating-Point Computations

- **Code Sample:**

```c
/* double-error.c */
int main ()
{
double x; float y, z, r;
x = 1.000000019e+38;
y = x + 1.0e21;
z = x - 1.0e21;
r = y - z;
printf("f\n", r);
} % gcc double-error.c
% .a.out
0.000000
```

\[(x + a) - (x - a) \neq 2a\]

---

Symbolic abstract domain

- **Interval analysis:** if \( x \in [a, b] \), \( y \in [c, d] \) & \( a, c \geq 0 \) then \( x - y \in [a - d, b - c] \) so if \( x \in [0, 100] \) then \( x - x \in [-100, 100] \)!!!

- The symbolic abstract domain propagates the symbolic values of variables and performs simplifications;

- Must maintain the maximal possible rounding error for float computations (overestimated with intervals);

```c
% cat -n x-x.c
1 void main () { int X, Y;
2 _ASTREE_isnum_fact(((D <= X) && (X <= 100)));
3 Y = (X - X);
4 _ASTREE_log_var(Y); }
```

**Call main(x).c:**

```c
1 Call main(x).c:5-x-x.c:1:5:
2 <interval: Y in [-100, 100]>
```

**astree -exec-fn main -no-relational x-x.c:**

```c
1 astree -exec-fn main x-x.c
```

---

Clock Abstract Domain for Counters

- Code Sample:
  ```c
  R = 0;
  while (1) {
    if (I)
      { R = R+1; }
    else
      { R = 0; }
  }
  ```
  - Output T is true if the volatile input I has been true for the last n clock ticks.
  - The clock ticks every s seconds for at most h hours, thus R is bounded.
  - To prove that R cannot overflow, we must prove that R cannot exceed the elapsed clock ticks (impossible using only intervals).

- Solution:
  - We add a phantom variable `clock` in the concrete user semantics to track elapsed clock ticks.
  - For each variable X, we abstract three intervals: X, X+clock, and X-clock.
  - If X-clock or X-clock is bounded, so is X.

Boolean Relations for Boolean Control

- Code Sample:
  ```c
  typedef enum {F=0,T=1} BOOL;
  BOOL B;
  void main () {
    unsigned int X, Y;
    while (1) {
      ... B = (X == 0);
      ... if (!B) {
      Y = 1 / X;
      }
    }
  }
  ```
  The boolean relation abstract domain is parameterized by the height of the decision tree (an analyzer option) and the abstract domain at the leaves.

Control Partitionning for Case Analysis

- Code Sample:
  ```c
  void main() {
    float c[] = (0.0, 2.0, 2.0, 0.0);
    float d[] = (0.0, 2.0, 2.0, 0.0);
    float x, r;
    int i = 0;
    ... found invariant -100 ≤ x ≤ 100 ... while (i <= 3) && (x := t[i+1]) {
      ... r = (x - t[i]) * c[i] + d[i];
    }
  }
  ```
  Delaying abstract unions in tests and loops is more precise for non-distributive abstract domains (and much less expensive than disjunctive completion).

2nd Order Digital Filter: Ellipsoid Abstract Domain for Filters

- Computes $X_n = \{ \alpha X_{n-1} + \beta X_{n-2} + Y_n \}$
- The concrete computation is bounded, which must be proved in the abstract.
- There is no stable interval or octagon.
- The simplest stable surface is an ellipsoid.
**Filter Example**

```c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN INIT; float P, X;
void filter () {
  static float E[2], S[2];
  if (INIT) { S[0] = X; P = X; E[0] = X; }
  else { P = ((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4)) + (S[0] * 1.5)) - (S[1] * 0.7)); }
  E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
  /* S[0], S[1] in [-1327.02698354, 1327.02698354] */
}
void main () { X = 0.2 * X + 5; INIT = TRUE;
  while (1) {
    X = 0.9 * X + 35; /* simulated filter input */
    filter (); INIT = FALSE; }
}
```

**Arithmetic-Geometric Progressions**

```c
% cat retro.c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN FIRST;
volatile BOOLEAN SWITCH;
volatile float E;
float P, X, A, B;
void dev() {
  X=E;
  if (FIRST) { P = X; }
  else { P = (P - (((2.0 * P) - A) - B) * 4.491048e-03)); }
  B = A;
  if (SWITCH) { A = P; }
  else { A = X; }
}
void main() {
  FIRST = TRUE;
  while (TRUE) {
    dev();
    FIRST = FALSE;
    __ASTREE_wait_for_clock();
  }
}
```

(Automatic) Parameterization

- All abstract domains of ASTRÉE are parameterized, e.g.
  - variable packing for octagones and decision trees,
  - partition/merge program points,
  - loop unrollings,
  - thresholds in widenings, ...;
- End-users can either parameterize by hand (analyzer options, directives in the code), or
  choose the automatic parameterization (default options, directives for pattern-matched predefined program schemata).

**The main loop invariant for the A340**

A textual file over 4.5 Mb with

- 6,900 boolean interval assertions \( x \in [0; 1] \)
- 9,600 interval assertions \( x \in [a; b] \)
- 25,400 clock assertions \( x + \text{clk} \in [a; b] \land x-\text{clk} \in [a; b] \)
- 19,100 additive octagonal assertions \( a \leq x + y \leq b \)
- 19,200 subtractive octagonal assertions \( a \leq x - y \leq b \)
- 100 decision trees
- 60 ellipse invariants, etc ...

involving over 16,000 floating point constants (only 550 appearing in the program text) \( \times 75,000 \) LOCs.
Possible origins of imprecision and how to fix it

In case of false alarm, the imprecision can come from:
- **Abstract transformers** (not best possible) $\rightarrow$ improve algorithm;
- **Automatized parametrization** (e.g. variable packing) $\rightarrow$ improve pattern-matched program schemata;
- **Iteration strategy** for fixpoints $\rightarrow$ fix widening $^3$;
- **Inexpressivity** i.e. indispensable local inductive invariant are inexpressible in the abstract $\rightarrow$ add a new abstract domain to the reduced product (e.g. filters).

---

**Bibliography**


---

$^3$ This can be very hard since at the limit only a precise infinite iteration might be able to compute the proper abstract invariant. In that case, it might be better to design a more refined abstract domain.

$^4$ The editor of JLP has mistakenly published the unreadable galley proof. For a correct version of this paper, see http://www.ens.fr/~cousot.