Proving the correctness of static analyzers

- The abstract interpretation theory provides a formal basis for proving the soundness (and sometimes the completeness) of static analyzers (abstract semantics).
- The principle is to proceed by induction on the syntax of programs, which yields a proof for the whole programming language.
- This structural proof will be formalized independently of any particular programming language.

- The proof relies on the use of a quite general form of collecting semantics, abstraction and abstract semantics as formalized by concretization functions (or abstract functions or Galois connections in case of existence of best abstractions).
- It is based on the use of fixpoint definitions for monotone operators on cpos (complete lattices).
- In absence of ACC, monotony is lost due to the use of widening/narrowing (and indeed monotony must be lost to enforce convergence).
An abstract formalization of finitary structural analysis by abstract interpretation

- In general we have:
  - Syntactic categories:
    \[ C_i \in \text{Com}_i, \ i \in \Delta \]
  - Immediate subcomponent relation:
    \[ \langle \bigcup_{i \in \Delta} \text{Com}_i, \prec \rangle \text{ is well-founded} \]
  - Example:
    \[ C = \text{while } B \text{ do } C' \text{ od} \]
    \[ B \prec C, T(\neg(B)) \prec C, C' \prec C \]
An abstract definition of the structural (collecting) semantics

- Concrete domains: define the semantics information associated to each syntactic category \( \text{Com}_i, \ i \in \Delta \). For each \( i \in \Delta \) and \( C_i \in \text{Com}_i \):
  \[ \langle \text{D}_C, \sqsubseteq_{C_i}, \sqcap_{C_i}, \sqcup_{C_i} \rangle \text{ is a poset (cpo, complete lattice, ...)} \]

- Concrete (aka collecting) semantics:

  \[ C_i \in \{ C_i \in \text{Com}_i \mapsto \text{D}_C \} \]

  is defined, by structural induction, as

  \[ C_i[C_i] \overset{\text{def}}{=} \mathcal{F}_i[C_i] \left( \prod_{C_j' \prec C_i} C_j[C_j'] \right) \]

  where

  \[ \mathcal{F}_i[C_i] \in \left( \prod_{C_j' \prec C_i} \text{D}_{C_j'} \right) \mapsto \text{D}_C \]

  is the collecting semantics transformer

The collecting semantics transformer is defined in the form:

\[ \mathcal{F}_i[C_i](S_1, \ldots, S_n) \overset{\text{def}}{=} e[D_{C_i}][S_1 : D_{C_i'}, \ldots, S_n : D_{C_i'}](\)\]

where \( \{C' | C' \prec C_i\} = \{C_1', \ldots, C_n'\} \) and the right-hand side is an expression written according to the following attribute grammar, where we are given

- \( S = S_1 : D_{C_i'}, \ldots, S_n : D_{C_i'} \): the collecting semantics of components
- \( X = X_{n+1} : D_{n+1}', \ldots, X_m : D_m' \): fixpoint variables
- \( (\text{D}, \sqsubseteq, \sqcap, \sqcup) \): the domain of the result

The attribute grammar of expressions is as follows:

\[ e[D][S](X) ::= \]

\[ d \]

\[ S_j \]

\[ X_k \]

\[ f_{D_{j_1}} \ldots f_{D_{j_n}} e[D_{j_1}][S](X), \ldots, e[D_{j_n}][S](X) \]

\[ \text{lfp} \left( Y \left. \lambda Y \cdot e[D][S](X, Y : D) \right) \right. \]

where

- \( d \in \text{D} \) is a constant
- \( S_j, j \in [1, n] \) is the semantics of an immediate component of \( C_i \) such that \( D_j = D \)

\[ ^1 Y \text{ must be a new fresh variable} \]
- $X_k, k \in [n+1, m]$ appears inside a fixpoint definition and $\mathcal{D}_k' = \mathcal{D}$
- $f_{j_1...j_k} \in (\prod_{j_1}^{n} D_j) \mapsto D$ is a constant function such as $f(x, y) = x \sqcup y; x \sqcap y, x \circ y, x(y)$, etc.
- The existence of the fixpoint definition should be ensured (by def. of the poset $(\mathcal{D}, \sqsubseteq, \sqcap, \sqcup)$ and properties of the function $\lambda Y. e[\mathcal{D}][S](X, Y : D)$ (such as monotony, continuity, extensivity, etc).
- In particular $\mu_p$ need not be a fixpoint and can be defined as the limit of an iteration process, a solution of constraints, etc.

Notes on typing the structural concrete semantics

- language:
  - We have a set $\Delta$ of indexes $i$ of syntactic categories $\text{Com}_i$
  - The language is $\langle \bigcup_{i \in \Delta} \text{Com}_i, \prec \rangle$ where "$\prec$" is the well-founded “immediate component” relation
- types:
  - We can consider a set $T$ of base types
  - The set $T$ of types is then defined inductively as $\forall t \in T : t \in T$ and if $t_i \in T, i = 1, \ldots, n + 1$ then $t_1 \times \ldots \times t_n \rightarrow t_{n+1} \in T$

- abstract domains:
  - For each type $t \in T$, the (concrete or abstract) semantic domain is $(\mathcal{D}_t, \sqsubseteq, \sqcap_t, \sqcup_t)$
- typing program component:
  - A base type $t_i \in T$ is associated to each program component $\text{Com}_i$, which is written $\text{Com}_i^{t_i}$
  - The intention is that the domain $(\mathcal{D}_t, \sqsubseteq, \sqcap_t, \sqcup_t)$ describes the possible behaviors of program component $\text{Com}_i$
- typing semantic expressions:
  - All expressions are typed:
    
    \[ e_t[S^t_1, \ldots, S^t_n](X^{t_{n+1}}, \ldots, X^{t_m}) \]
  - These expressions are interpreted as functions:
    
    \[ D_{t_1} \times \ldots \times D_{t_n} \times D_{t_{n+1}} \times \ldots \times D_{t_m} \rightarrow D_t \]

- typing rules for semantic expressions:
  - The typing rules for expressions are as follows:
    - \( d_t \in D_t \), \( S^t \) and \( X^t \) have type \( t \)
    - \( f \) has type \( t_1 \times \ldots \times t_n \rightarrow t \) (written \( f^{t_1 \times \ldots \times t_n \rightarrow t} \)) if, whenever \( e_i \) has type \( t_i \), \( i = 1, \ldots, n \), then \( f(e_1, \ldots, e_n) \) has type \( t \)

- type soundness for semantic equational definitions:
  - It follows, by structural induction, that \( C[C^t_i] \in D_t \),
  - If the implementation is in a typed functional language (such as OCaml), this typing is done by the compiler. Otherwise, that may have to be done by hand.

- Example of structural concrete semantics:
  - forward collecting semantics of arithmetic expressions

- \( X \in \mathbb{V} \), variables
  - \( A ::= n \mid X \mid uA \mid A_1 b A_2 \), \( A \in Aexp \), arithmetic expressions
  - \( \text{Faexp} \in Aexp \rightarrow D_{Aexp} \)
    
    \[ \text{Faexp}[A] \overset{\text{def}}{=} \{ v \mid \exists \rho \, : R : \rho \vdash A \Rightarrow v \}^2 \]
  - \( \text{I} \overset{\text{def}}{=} [\text{min}\_\text{int}, \text{max}\_\text{int}] \), machine integers

---

\[ ^2 \text{where } \rho \vdash A \Rightarrow v, \text{as defined by the operational semantics, holds whenever evaluation of } A \text{ in environment } \rho \text{ may return value } v \]
- \( \mathbb{E} \overset{\text{def}}{=} \{ \Omega_1, \Omega_a \} \), errors
- \( \mathbb{I}_\Omega \overset{\text{def}}{=} \mathbb{I} \cup \mathbb{E} \), machine values
- \( \mathbb{R} \overset{\text{def}}{=} \mathcal{V} \mapsto \mathbb{I}_\Omega \), environments
- \( \mathcal{D}_{\text{Aexp}} \overset{\text{def}}{=} \mu(\mathbb{R}) \mapsto \mu(\mathbb{I}_\Omega) \) (same for all \( A \in \text{Aexp} \)), properties
- \( \mathcal{F}_{\text{Aexp}}[n] \overset{\text{def}}{=} \lambda R \cdot \{ \mu \} \) constant of \( \mathcal{D}_{\text{Aexp}} \)
- \( \mathcal{F}_{\text{Aexp}}[x] \overset{\text{def}}{=} \lambda R \cdot R(x) = \lambda R \cdot \{ \rho(x) | \rho \in R \} \) constant of \( \mathcal{D}_{\text{Aexp}} \)

where

\[ b^C \in \mathcal{D}_{\text{Aexp}} \times \mathcal{D}_{\text{Aexp}} \mapsto \mathcal{D}_{\text{Aexp}} \]

and

\[ b^C(S_1, S_2) \overset{\text{def}}{=} \lambda R \cdot \{ v_1 \triangleright v_2 \mid \exists \rho \in R : v_1 \in S_1(\{ \rho \}) \wedge v_2 \in S_2(\{ \rho \}) \} \]

so that

\[ \mathcal{F}_{\text{Aexp}}[A_1 b A_2] \overset{\text{def}}{=} \lambda R \cdot b^C(\mathcal{F}_{\text{Aexp}}[A_1], \mathcal{F}_{\text{Aexp}}[A_2])R \]

Example of structural concrete semantics:

boolean expressions

- \( B ::= \) true | false | \( A_1 \in A_2 \mid B_1 \& B_2 \mid B_1 \mid B_2 \), \( B \in \text{Bexp} \), boolean expressions
- \( \mathcal{C}_{\text{Bexp}}[B] \in \mathcal{B}_{\text{Bexp}} \mapsto \mathcal{B}_{\text{Bexp}} \)

\[ \mathcal{C}_{\text{Bexp}}[B] \overset{\text{def}}{=} \lambda R \cdot \{ \rho \in R \mid \rho \vdash B \Rightarrow \mathbf{tt} \} \]

\[ \mathcal{D}_{\text{Bexp}} \overset{\text{def}}{=} \mu(\mathbb{R}) \mapsto \mu(\mathbb{R}) \]

- \( \mathcal{C}_{\text{Bexp}}[\mathbf{true}] \overset{\text{def}}{=} \lambda R \cdot R \) constant of \( \mathcal{D}_{\text{Bexp}} \)
- \( \mathcal{C}_{\text{Bexp}}[\mathbf{false}] \overset{\text{def}}{=} \lambda R \cdot \mathbf{0} \) constant of \( \mathcal{D}_{\text{Bexp}} \)

\( ^3 \) where \( \rho \vdash B \Rightarrow \mathbf{tt} \) is defined by the operational semantics as holding in environment \( \rho \) when \( B \) is true and without runtime error.
Example of structural concrete semantics: commands, sequences and programs

- \( C ::= \text{skip} \mid X := A \quad C \in \text{Com}, \text{commands} \)
- \( \text{if } B \text{ then } S \text{ else } S \text{ fi} \)
- \( \text{while } B \text{ do } S \text{ od} \)
- \( S ::= C ; S \mid C \quad S \in \text{Seq}, \text{sequences of commands} \)
- \( P ::= S ; \quad P \in \text{Prog}, \text{programs} \)
- For all \( I \in \text{Com} \cup \text{Seq} \cup \text{Prog} \), we have \( \text{Rcom}[I] \in \text{DCom}[I] \) where

\[
\text{DCom}[I] \triangleq \rho(\mathbb{R}) \cup (\text{in}_P[I] \rightarrow \rho(\mathbb{R}))
\]

so that:

\[
\text{Cbexp}[B_1 \land B_2] \triangleq \lambda R. \text{Cbexp}[B_1](\text{Cbexp}[B_2](R))
\]

- For all \( I \in \text{Com} \cup \text{Seq} \cup \text{Prog} \), we have \( \text{Rcom}[I] \in \text{DCom}[I] \) where

\[
\text{DCom}[I] \triangleq \rho(\mathbb{R}) \cup (\text{in}_P[I] \rightarrow \rho(\mathbb{R}))
\]

- \( \text{Rcom}[I]R\ell \triangleq \lambda R. \lambda \ell. \{ \rho \mid \exists \rho' \in R : \langle \text{at}_P[C], \rho' \rangle, \langle \ell, \rho \rangle \in \tau^*[C] \} \in \tau^*[C] \) reachable states (according to the operational semantics defined in lecture 5)

- \( \text{Rcom}[\text{skip}] = \lambda R. \lambda \ell. R \quad (\text{constant of } \text{DCom}[\text{skip}]) \)
- \( \text{Rcom}[X := A] = \text{FCom}[X := A](\text{Fexp}[A]) \)
- \( \text{FCom}[X := A](S) = f_X := A(S) \)
- \( f_X := A(S) = \lambda R. \lambda \ell. \text{match } \ell \text{ with} \)
- \( \text{at}_P[X := A] \rightarrow R \)
- \( \text{after}_P[X := A] \rightarrow \{ \rho[X := i] \mid \rho \in R \land i \in (S(\{\rho\})) \cap \mathbb{I} \} \)
so that
\[ \text{Rcom}[X := A] \text{R} \ell = \text{match } \ell \text{ with } \]
| \[ \text{at}_p[X := A] \rightarrow R \] 
| \[ \text{after}_p[X := A] \rightarrow \{ \rho[X := i] \mid \rho \in R \land i \in (\text{Faexp}[A]\{\rho}\}) \cap I \} \]
- \[ \text{Rcom}[C] \text{ where } C = \text{if } B \text{ then } S_t \text{ else } S_f \text{ fi} \]
\[ = \mathcal{F}_\text{Com}[C](\text{Cbexp}[B], \text{Cbexp}[T(-(B))], \text{Rcom}[S_t], \text{Rcom}[S_f]) \]
\[ \mathcal{F}_\text{Com}[C](B_1, B_2, S_1, S_2) \overset{\text{def}}{=} f_C(B_1, B_2, S_1, S_2) \]
where
\[ f_C \in \mathcal{D}_\text{Bexp} \times \mathcal{D}_\text{Bexp} \times \mathcal{D}_\text{Com}[S_t] \times \mathcal{D}_\text{Com}[S_f] \rightarrow \mathcal{D}_\text{Com}[C] \]

- \[ \text{Rcom}[C] \text{ where } C = \text{while } B \text{ do } S \text{ od} \]
\[ = \mathcal{F}_\text{Com}[C](\text{Cbexp}[B], \text{Cbexp}[T(-(B))], \text{Rcom}[S]) \]
\[ \mathcal{F}_\text{Com}[C](B_1, B_2, S_1) \overset{\text{def}}{=} f_C(B_1, B_2, S_1, \text{if}_p^\ell \lambda X \cdot f_S(B_1, S_1, X)) \]
where
\[ f_S(B_1, S_1, X) \overset{\text{def}}{=} \lambda R \cdot R \cup S_1(B_1(X))(\text{after}_p[S]) \]
\[ f_C \in \mathcal{D}_\text{Bexp} \times \mathcal{D}_\text{Com}[S] \times \mathcal{D}_\text{Com}[C] \rightarrow \mathcal{D}_\text{Com}[C] \]
\[ f_C(B_1, B_2, S_1, F_1) \overset{\text{def}}{=} \lambda R \cdot \lambda \ell \cdot \text{match } \ell \text{ with } \]
| \[ \text{at}_p[C] \rightarrow I \] 
| \[ \text{in}_p[S] \rightarrow S_1(B_1(F_1(R)))(\ell) \] 
| \[ \text{after}_p[C] \rightarrow B_2(F_1(R)) \]

so that
\[ \text{Rcom}[C] \text{R} \ell \text{ where } C = \text{while } B \text{ do } S \text{ od} = \]
let \[ I = \text{if}_p^\ell \lambda X \cdot R \cup \text{Rcom}[S](\text{Cbexp}[B]X)(\text{after}_p[S]) \text{ in} \]
match \[ \ell \text{ with } \]
| \[ \text{at}_p[C] \rightarrow I \] 
| \[ \text{in}_p[S] \rightarrow \text{Rcom}[S](\text{Cbexp}[B]I)(\ell) \] 
| \[ \text{after}_p[C] \rightarrow \text{Cbexp}[T(-(B))][R]I \]

- Note that to prove equivalence, we need the following result:
**Theorem.** Let \( (L, \subseteq, \perp, \sqcup) \) be a cpo, \( F \in E \mapsto (L \rightarrow L) \). Then \( \forall R \in E : \)

\[
\text{\texttt{lfp}}_\perp \lambda X. F(R, X) = (\text{\texttt{lfp}}_\perp \lambda Y. \lambda R. F(R, Y(R)))(R)
\]

**Proof.** Let \( X^\delta, \delta \in \mathbb{N} \) be the iterates of \( \text{\texttt{lfp}}_\perp \lambda X. F(R, X) \) with rank \( \varepsilon \).

- Let \( Y^\delta, \delta \in \mathbb{N} \) be the iterates of \( \text{\texttt{lfp}}_\perp \lambda Y. \lambda R. F(R, Y(R)) \) with rank \( \varepsilon' \).
- We prove by transfinite induction that \( X^\delta = Y^\delta(R) \).
  - \( X^0 = \bot = Y^0(R) \)
  - If \( X^\delta = Y^\delta(R) \) then

\[
X^{\delta+1} = F(R, X^\delta) = F(R, Y^\delta(R)) = \lambda R'. F(R', Y^\delta(R'))(R) = \lambda Y. (\lambda R'. F(R', Y^\delta(R'))(R))(Y^\delta) = Y^{\delta+1}(R)
\]

- It \( \lambda \) is a limit ordinal and \( \forall \beta < \lambda : X^\beta = Y^\beta(R) \) then

\[
X^\lambda = \bigsqcup_{\beta < \lambda} X^\beta = \bigsqcup_{\beta < \lambda} Y^\beta(R) = (\lambda R'. \bigsqcup_{\beta < \lambda} Y^\beta(R'))(R)
\]
Theorem. If fixpoints exist then the structural semantic definition is well-defined.

**Proof.** By structural induction.
- For expressions \( e[D][S](X) \in D \), by cases:
  - Basis
    - \( d \in D \) by hypothesis
    - \( S_j \in D \) by hypothesis
    - \( X_k \in D_k = D \) by hypothesis
  - Induction step
    - For all \( k = 1, \ldots, \ell \), \( e_k[D_h][S](X) \in D_h \) by induction hypothesis and
      \( f_{D_1, \ldots, D_\ell}(\prod_{j=1}^\ell D_j) \to D \) by hypothesis, proving that
      \( f_{D_1, \ldots, D_\ell}(e_1[D_h][S](X)), \ldots, e_\ell[D_h][S](X)) \)

Well-definedness of structural semantics

**Definition (fixpoint existence).**
- We say that the **fixpoints exist** if and only if all fixpoints appearing in the structural definition exist.
- A fixpoint \( \mu_0 \subseteq F \) (where \( F : L \to L \), \( \{ \subseteq, \sqsubseteq, \sqcup, \sqcap \) ) is a poset) **exists** whenever the transfinite iteration sequence \( X_0 = a \), \( X_\delta + 1 = F(X_\delta) \), \( X_\lambda = \bigsqcup_{\beta < \lambda} X_\beta \) for limit ordinals is well-defined (i.e. the lub \( \sqcup \) does exist ), ultimately stationary at rank \( e \) (so that \( \forall \delta \geq e : X_\delta = X_e \) in which case we let \( \mu_0 \subseteq F \) be \( X_e \).

Fixpoint existence

In the structural definition of the semantics, all elements are well-defined but, may be, the fixpoints.

- This concludes the proof that the forward collecting semantics of a command (as introduced in lecture 16) is of the general form on which we reason afterwards.
Monotonic structural semantics

**Definition.** The structural semantics is said to be monotonic whenever all $f_{D_{j_1} \cdots D_{j_k}} \in \left( \prod_{j=j_1}^{j_k} D_j \right) \rightarrow D$ are monotonic on the poset $(D, \sqsubseteq)$, that is: $\forall k = 1, \ldots, \ell : \forall X_{j_k}, X'_{j_k} \in D_{j_k}, X_{j_k} \sqsubseteq X'_{j_k} \implies f_{D_{j_1} \cdots D_{j_k}}( \prod_{k=1}^{\ell} X_{j_k} ) \sqsubseteq f_{D_{j_1} \cdots D_{j_k}}( \prod_{k=1}^{\ell} X'_{j_k} )$.

- Induction step:
  - For all $k = 1, \ldots, \ell$, $e_k[D_{j_k}] \in D_{j_k}$ is well-defined and monotone by induction hypothesis and $f_{D_{j_1} \cdots D_{j_k}} \in \left( \prod_{j=j_1}^{j_k} D_j \right) \rightarrow D$ by hypothesis, proving that if $Y \sqsubseteq Y'$ then $e[D][S](X, Y : D) \sqsubseteq e[D][S]'(X', Y' : D)$ so that the function $F \equiv \lambda Y. e[D][S](X, Y : D) \in D \rightarrow D$ is monotonic in its $Y$ parameter. By the constructive version of Tarski’s theorem, $\text{fix}_D F$ does exist on the cpo $(D, \sqsubseteq, \bot, \sqcup)$ and is an element of $D$.
  - Moreover if we let
    \[ F \equiv \lambda Y. e[D][S](X, Y : D) \]
    and $F' \equiv \lambda Y. e[D][S]'(X', Y' : D)$
    then $F \subseteq F'$ pointwise and to $\text{fix}_D F \subseteq \text{fix}_D F'$, proving monotony.

Well-definedness of monotonic structural semantics

**Theorem.** In a monotonic structural semantics, all expressions are monotonic, whence the fixpoints exist on cpos, so the semantics is well-defined on cpos.

**Proof.** - For expressions if $(D_1, \sqsubseteq_1, \bot_1, \sqcup_1)$ and $(D, \sqsubseteq, \bot, \sqcup)$ are cpos then $\prod_{i=1}^{n_1} D_i$ and $\prod_{i=n_1+1}^{n} D_i$ are cpos for the componentwise orderings $\sqsubseteq_1$ and $\sqsubseteq \sqsubseteq_1$. Assume $S \sqsubseteq S'$, $X \sqsubseteq X'$, $X \sqsubseteq X'$, prove by structural induction on expression $e$ that $e[D][S](X) \subseteq e[D][S'](X')$ and the expression is well-defined in $D$.
- Base:
  - $d \sqsubseteq d$ by reflexivity and $d \in D$, by hypothesis
  - $S_j \sqsubseteq S_j'$ by def. componentwise ordering and $\subseteq_j = \subseteq$ with $S_j, S_j' \in D_j = \sqcup D_j$ by hypothesis
  - $X_k \sqsubseteq X_k'$ by def. componentwise ordering and $\subseteq_k = \subseteq$ with $X_k \in \sqcup D_k$ by hypothesis

\[
F_i[C_i] \subseteq \prod_{C_i \leq C_i} D_{C_i}^m \rightarrow D_{C_i}
\]

and well-defined
- Since fixpoints exist, the structural semantics is well-defined.
Structural abstract semantics

The abstract semantics is in the same structural form as the collecting semantics. More precisely:

- **Abstract domains:** define the abstract information associated to each syntactic category $\text{Com}_i$, $i \in \Delta$. For each $i \in \Delta$ and $C_i \in \text{Com}_i$:
  $\langle D_{C_i}, \sqsubseteq_{C_i}, \sqcup_{C_i}, \sqcap_{C_i} \rangle$ is a poset (cpo, complete lattice, ...)

(The nature of the correspondence between the abstract domains and the corresponding concrete ones will be considered later).

- **Abstract semantics:**
  $\overline{C}_i \in [C_i \in \text{Com}_i \mapsto D_{C_i}]$

is defined, by structural induction, as

$$\overline{C}_i[C_i] \defeq \mathcal{F}_i[C_i] \left( \prod_{C_j' \prec C_i} \overline{C}_j[C_j'] \right)$$

where

$$\mathcal{F}_i[C_i] \in \left( \prod_{C_j' \prec C_i} D_{C_j'} \right) \mapsto D_{C_i}$$

is the abstract transformer

The abstract transformer is defined in the form:

$$\mathcal{F}_i[C_i](S_1, \ldots, S_n) \defeq \mathcal{F}[\mathcal{D}_{C_i}][\mathcal{S}_1 : \mathcal{D}_{C_i'} : \mathcal{S}_n : \mathcal{D}_{C_n}()]$$

where $\{C' \mid C' \prec C_i\} = \{C'_1, \ldots, C'_n\}$ and the right-hand side is an expression written according to the following attribute grammar, where we are given

- $\mathcal{S} = \mathcal{S}_1 : \mathcal{D}_{C_1'} : \mathcal{S}_n : \mathcal{D}_{C_n'}$: the abstract semantics of components
- $\mathcal{X} = \mathcal{X}_{n+1} : \mathcal{D}_{n+1} : \mathcal{X}_m : \mathcal{D}_m$: fixpoint variables
- $\langle \mathcal{D}, \sqsubseteq, \sqcup, \sqcap \rangle$: the abstract domain of the result

- **Abstract domains:** define the abstract information associated to each syntactic category $\text{Com}_i$, $i \in \Delta$. For each $i \in \Delta$ and $C_i \in \text{Com}_i$:
  $\langle D_{C_i}, \sqsubseteq_{C_i}, \sqcup_{C_i}, \sqcap_{C_i} \rangle$ is a poset (cpo, complete lattice, ...)

(The nature of the correspondence between the abstract domains and the corresponding concrete ones will be considered later).

- **Abstract semantics:**
  $\overline{C}_i \in [C_i \in \text{Com}_i \mapsto D_{C_i}]$

is defined, by structural induction, as

$$\overline{C}_i[C_i] \defeq \mathcal{F}_i[C_i] \left( \prod_{C_j' \prec C_i} \overline{C}_j[C_j'] \right)$$

where

$$\mathcal{F}_i[C_i] \in \left( \prod_{C_j' \prec C_i} D_{C_j'} \right) \mapsto D_{C_i}$$

is the abstract transformer

The attribute grammar of expressions is as follows:

$$\mathcal{E}[[\mathcal{D}][\mathcal{S}]](\mathcal{X}) ::=$$

$$\bar{d}$$

$$\mathcal{S}_j$$

$$\mathcal{X}_k$$

$$\bar{f}_{\mathcal{D}_{j1}} \cdots \bar{f}_{\mathcal{D}_{j}} \mathcal{E}(e_1[[\mathcal{D}_{j1}][\mathcal{S}]](\mathcal{X})), \ldots, e_k[[\mathcal{D}_{j_k}][\mathcal{S}]](\mathcal{X}))$$

$$\lambda \bar{Y} . \mathcal{E}[[\mathcal{D}][\mathcal{S}]](\mathcal{X}, \bar{Y} : \mathcal{D})$$

where, by hypothesis:

- $\bar{d} \in \mathcal{D}$ is a constant

\[ \bar{Y} \notin \mathcal{X} \cup \mathcal{D} \] must be a new fresh variable
Local abstraction hypotheses on the correspondence between concrete and abstract semantics

- Observe that the concrete and abstract semantics have the same structural form (and so are stated to be structurally identical)
- So, intuitively, if the ingredients of the abstract semantics are upper-approximations of their concrete counterparts, then the abstract semantics should be an upper-approximation of the concrete semantics
- This is made precise and proved in what follows
A soundness of the correspondence between expressions

**THEOREM.** If \( e \) and \( \overline{e} \) are structurally identical, (LA1), (LA2) and (LA3) hold, concrete and abstract fixpoints exist, and

- \( S_j \sqsubseteq_j \gamma_i(\overline{S}_j), j = 1, \ldots, n \) (a)
- \( X_k \sqsubseteq_k \gamma_i(\overline{X}_k), k = n + 1, \ldots, m \) (b)

then

\[
e[D][S](X) \sqsubseteq \gamma(\overline{e}[\overline{D}][\overline{S}](\overline{X}))
\]

**PROOF.** By structural induction on expressions.

A soundness theorem on the correspondence between a concrete semantics and its local abstraction

- Given an abstract semantics which is a local abstraction of a structurally identical concrete semantics, we prove that this abstract semantics is a sound upper-approximation of the concrete semantics
- We proceed by structural induction on the considered programming language
\( \mathcal{F}(Y) \subseteq \gamma(\mathcal{F}(Y)) \) \hspace{1cm} (c)

for all \( Y \in \mathcal{D} \) and \( \mathcal{F} \in \mathcal{D} \)

- Let us now consider the iterates \( \langle X^\delta, \delta \in \mathcal{O} \rangle \) of \( \mathcal{F} \) and \( \langle Y^\delta, \delta \in \mathcal{O} \rangle \) of \( \mathcal{F} \) which are respectively stationary at \( e \) and \( e' \), by fixpoint existence hypothesis
  - \( X^0 \overset{\text{def}}{=} \bot \subseteq \gamma(\mathcal{F}) \overset{\text{def}}{=} Y^0 \)
  - If \( X^\delta \subseteq \gamma(Y^\delta) \) by induction hypothesis, then \( X^{\delta+1} = \mathcal{F}(X^\delta) \subseteq \gamma(\mathcal{F}(Y^\delta)) \)
    by (c) proving that \( Y^{\delta+1} = \mathcal{F}(Y^\delta) \)
  - If \( \lambda \) is a limit ordinal and \( \forall \beta < \lambda : X^\beta \subseteq \gamma(Y^\beta) \) then \( X^\lambda = \bigcup\beta<\lambda X^\beta \)
    \( \subseteq \bigcup\beta<\lambda \gamma(Y^\beta) \subseteq \gamma(\bigcup\beta<\lambda Y^\beta) = \gamma(Y^\lambda) \) (which are well-defined by fixpoint existence)
- It follows that \( \text{lfp} F = X' = X^{\text{max}(\delta,e')} \subseteq \gamma(\bigcup\beta<\lambda Y^\beta) = \gamma(\bigcup\beta<\lambda Y^\beta) = \gamma(Y') = \gamma(\text{lfp} F) \)

\[ C[C_i] \]

\[ = \mathcal{F}[C_i](\prod_{C_j < C_i} C[C_j]) \]

\[ = \text{e}[\mathcal{D}[C_i]](\prod_{C_j < C_i} \mathcal{C}[C_j] : \mathcal{D}[C_i](\prod_{C_j < C_i} \mathcal{C}[C_j])) \]

\[ \subseteq \gamma_i(\mathcal{F}[\mathcal{C}[C_i]](\prod_{C_j < C_i} \mathcal{C}[C_j])) \]

\[ = \gamma_i(\mathcal{C}[C_i](\prod_{C_j < C_i} \mathcal{C}[C_j])) \]

\[ \square \]

Soundness of the correspondence between a concrete and abstract semantics

**Theorem.** If (LA1), (LA2) and (LA3) do hold, concrete and abstract fixpoints exist, then for all \( i \in \Delta \) and \( C_i \in \text{Com}_i \), we have

\[ C[C_i] \subseteq \gamma_i(C[C_i]) \]

**Proof.** By structural induction on the well-founded relation \( \langle \bigcup_{i \in \Delta} \text{Com}_i, \prec \rangle \).

Given any \( i \in \Delta \) and \( C_i \in \text{Com}_i \), assume by induction hypothesis that

\[ \forall C_j \prec C_i : C[C_j] \subseteq C[C_j] \]

then

\[ C[C_i] \subseteq \gamma_i(C[C_j]) \]

An abstract formalization of infinitary structural analysis by abstract interpretation
Hypotheses on widenings

Given a poset \( \langle L, \sqsubseteq \rangle \), a widening operator on \( L \) is \( \sqcup \in L \times L \mapsto L \) satisfying

\[(W1) \quad y \sqsubseteq x \sqcup y \]

\[(W2) \quad \text{ For all sequences } x^0, x^1, \ldots \text{ in } L^\omega, \text{ the sequence defined by } \]

\[y^0 \overset{\text{def}}{=} x^0 \]

\[y^{n+1} \overset{\text{def}}{=} \left\{ \begin{array}{ll}
  y^n \sqcup x^n & \text{ if } \exists \ell \leq n: x^\ell \sqsubseteq y^\ell \\
  y^n & \text{ otherwise}
\end{array} \right. \]

is not strictly increasing.


Theorem. The sequence \( \langle y^k, k \in \mathbb{N} \rangle \) is strictly increasing up to a least \( \ell \in \mathbb{N} \) such that \( x^\ell \sqsubseteq y^\ell \) and the sequence is stationary at \( \ell \) onwards.

Proof. The sequence \( \langle y^k, k \in \mathbb{N} \rangle \) cannot by strictly increasing by (W2). So there is a least \( \ell \) such that \( y^\ell \not\sqsubseteq y^{\ell+1} \). We cannot have \( y^{\ell+1} = x^\ell \sqcup y^\ell \) since by (W1), thus would imply that \( y^\ell \sqsubseteq x^\ell \sqcup y^\ell \). Hence, by definition of the sequence \( \langle y^k, k \in \mathbb{N} \rangle \), we must have \( y^{\ell+1} = y^k \) where \( k \leq \ell \) and \( x^k \sqsubseteq y^k \).

We cannot have \( k < \ell \) since for the smallest such \( k \) we would have \( x^k \sqsubseteq y^k \) whence \( y^{k+1} = y^k \) whence, by reflexivity, \( y^k \sqsubseteq y^{k+1} \) in contradiction with the hypothesis that \( \ell \) is the smallest natural with that property. It follows that \( k = \ell \) and so by (W2): \( y^{\ell+1} = y^\ell \) and \( x^\ell \sqsubseteq y^\ell \). For all \( n \geq \ell \), we have \( \exists \ell \leq n: x^\ell \sqsubseteq y^\ell \) whence \( y^{n+1} = y^n \) proving that the sequence is stationary at \( \ell \).

Note: \( \langle x^k, k \in \mathbb{N} \rangle \) not assumed to be increasing.


Structural abstract semantics with widenings

The abstract semantics with widening is in the same structural form as the collecting semantics. More precisely:

- Abstract domains: For each \( i \in \Delta \) and \( C_i \in \text{Com}_i \):
  \( \langle \overline{D}_{C_i}, \sqsubseteq_{C_i}, \sqcup_{C_i}, \sqcap_{C_i} \rangle \) is a poset

- Widensings: For each \( i \in \Delta \) and \( C_i \in \text{Com}_i \):
  \( \sqcup_{C_i} \in \overline{D}_{C_i} \times \overline{D}_{C_i} \mapsto \overline{D}_{C_i} \) is a widening satisfying (W1) and (W2)

- Abstract semantics with widening:
  \( \overline{C_i} \in [C_i \in \text{Com}_i \mapsto \overline{D}_{C_i}] \)

is defined, by structural induction, as

\[ \overline{C_i}[C_i] \overset{\text{def}}{=} \overline{F_i}[C_i] \left( \prod_{C_j \prec C_i} \overline{C_j}[C'_j] \right) \]

where

\[ \overline{F_i}[C_i] \in \left( \prod_{C_j \prec C_i} D_{C_j'} \right) \mapsto \overline{D}_{C_i} \]

is the abstract transformer

Course 16.399: “Abstract interpretation”, Tuesday May 5th, 2005 — 64 — P. Cousot, 2005
The abstract transformer is defined in the form:

\[ \mathcal{F}_i[[C_i]](s_1, \ldots, s_n) \overset{\text{def}}{=} \varepsilon[[D_C_i]][S_1 : D_{C_1}, \ldots, S_n : D_{C_n}]() \]

where \( \{C' \mid C' \prec C_i\} = \{C_1', \ldots, C_n'\} \) and the right-hand side is an expression written according to the following attribute grammar, where we are given
- \( S = S_1 : D_{C_1}, \ldots, S_n : D_{C_n} \): the abstract semantics of components
- \( X = X_{n+1} : D'_{n+1}, \ldots, X_m : D'_m \): fixpoint variables
- \( (D, \subseteq, \sqsubseteq, \sqsupseteq) \): the abstract domain of the result

The attribute grammar of expressions is as follows:

\[ \varepsilon[[D]][S](X) ::= \]

\[ \begin{align*}
&| D \\
&| S_j \\
&| X_k \\
&| \mathcal{F}_{D_j \ldots D_k} \left( e_1[[D_j]][[S]](X), \ldots, e_\ell[[D_j]][S](X) \right) \\
&| \text{let} \\
&\quad \mathcal{F} = \lambda Y . \varepsilon[[D]][S](X, Y : D) \text{ and} \\
&\quad \mathcal{G} = \lambda X . \text{let } Y = \mathcal{F}(X) \text{ in } (Y \subseteq X \land X \sqsubseteq Y) \\
&\text{in} \\
&\quad \text{if}_{\mu} \bot \mathcal{G}
\end{align*} \]

where, by hypothesis:
- \( \overline{d} \in D \) is a constant
- \( S_j, j \in [1, n] \) is the abstract semantics of an immediate component of \( C_i \) such that \( D_j = \overline{D} \)
- \( X_k, k \in [n+1, m] \) appears inside a fixpoint definition and \( D_k = \overline{D} \)
- \( \mathcal{F}_{D_j \ldots D_k} \in \left( \prod_{j=\ell}^{j'} D_j \right) \mapsto \overline{D} \) is a constant function
- \( \nabla \in D \times D \mapsto D \) is a widening
- \( \text{if}_{\mu} \bot F \) is a shorthand for the limit \( X^\varepsilon \) of the transfinite iteration sequence \( X^0 = \bot, X^{\delta+1} = F(X^\delta) \) and \( X^\lambda = \bigcup_{\beta<\lambda} X^\beta, \lambda \text{ limit ordinal}, \forall \delta \geq \varepsilon : X^\delta = X^\varepsilon \), whenever it exists.

Theorem. Any structural abstract semantics with widening satisfying (W1) and (W2) is well-defined.
PROOF. By structural induction on the inductive definition of the abstract semantics.
- For expressions \( \pi[D]|S(X) \in D \), by cases:
  - Basis
    - \( D \in D \), by hypothesis
    - \( D_1 \in D_2 \), by hypothesis
    - \( X_k \in \overline{D}_k \), by hypothesis
  - Induction step
    - For all \( k = 1, \ldots, \ell \), \( \pi[D_k]|S(X) \in D_k \) by induction hypothesis and \( \overline{f}_{\pi[D_k]|S(X)} \rightarrow D \) by hypothesis, proving that
      \( \pi[D_1]|S(X); \ldots; \pi[D_k]|S(X)) \)

is well-defined and belongs to \( \overline{D} \)
- If \( Y \in \overline{D} \) then, by induction hypothesis, \( \pi[D]|S(X, Y : \overline{D}) \) is well-defined and belongs to \( \overline{D} \), so that the locally defined function \( \overline{f} = \lambda Y. \pi[D]|S(X, Y : \overline{D}) \) is well-defined and belongs to \( \overline{D} \rightarrow \overline{D} \). Since, by hypothesis, \( \overline{f} \in \overline{D} \times \overline{D} \rightarrow \overline{D} \), it follows that for all \( X \in \overline{D}, \overline{f}(X) \) is well-defined and belongs to \( \overline{D} \rightarrow \overline{D} \). Let us now consider the iterates \( \overline{Z}^n, \delta \in N \) of \( \overline{D} \) starting from \( \overline{I} \in \overline{D} \). They are defined as:

\[
\begin{align*}
\overline{Z}^0 & \overset{\text{def}}{=} \overline{I} \\
\overline{Z}^{n+1} & \overset{\text{def}}{=} \overline{Z}^n \if \overline{f}(\overline{Z}^n) \subseteq \overline{Z}^n \\
& \overset{\text{def}}{=} \overline{Z}^n \setminus \overline{f}(\overline{Z}^n) \otherwise
\end{align*}
\]

Let \( \ell \in N \) be the smallest \( n \), if any, such that \( \overline{f}(\overline{Z}^\ell) \subseteq \overline{Z}^\ell \) then by recurrence, \( \forall k \geq \ell : \overline{f}(\overline{Z}^k) \subseteq \overline{Z}^k \) and so the above \( \overline{Z}^n, n \in N \) can be defined in the equivalent form

- For the induction step, \( \overline{T}_1[C_i] \in \overline{D}_1 \) by induction hypothesis and so \( \overline{T}_1[C_i] = \overline{T}_1[C_i] \), \( \overline{T}_1[C_i] = \overline{T}_1[C_i] \) where \( \{ C' \mid C' < C_i \} = \{ C_1, \ldots, C_n \} \) is well-defined and belongs to \( \overline{D}_1 \)

\[\square\]
A soundness theorem on the correspondence between concrete semantics and its local abstraction by a structural abstract semantics with widening

THEOREM. If \( e \) and \( \overline{e} \) are structurally identical, (LA1), (LA2) and (LA3) hold, concrete fixpoints exist, (W1) and (W2) hold, and
- \( S_j \subseteq \gamma_i(\overline{S}_j) \), \( j = 1, \ldots, n \) \hspace{1cm} (a)
- \( X_k \subseteq \gamma_i(\overline{X}_k) \), \( k = n + 1, \ldots, m \) \hspace{1cm} (b)

then

\[ e[D][S](X) \subseteq \gamma(\overline{e[D][S]})(X) \]

\[ \square \]

PROOF. By structural induction on expressions.
- \( d \subseteq (\overline{d}) \) by (LA2)
- \( S_j \subseteq \gamma_i(\overline{S}_j) = \gamma(\overline{S}_j) \) by (a) and \( D = D_j \), \( \overline{D} = D_j \), \( j = 1, \ldots, n \)
- \( X_k \subseteq \gamma_i(\overline{X}_k) = \gamma(\overline{X}_k) \) by (b) and \( D = D_k \), \( \overline{D} = \overline{D}_k \), \( k = n + 1, \ldots, m \)
- By induction hypothesis, we have:

\[ e_k[D][S](X) \subseteq j_k \gamma_j(\overline{e_k[D][S]})(X) \]

and so by (LA3):

\[ f_{D_j \cdots D_k}(\bigcup_{k=1}^\ell e_k[D][S](X)) \subseteq \gamma(\overline{f_{D_j \cdots D_k}(\bigcup_{k=1}^\ell e_k[D][S](X))}) \]

- In the case of a fixpoint definition with widening, we let

\[ \mathcal{F} \triangleq \lambda Y . e[D][S](X, Y : D) \]

If \( Y \subseteq \gamma(\overline{Y}) \) then, by induction hypothesis on the identical structures of \( e \) and \( \overline{e} \), we have

\[ \mathcal{F}(Y) \subseteq \gamma(\overline{\mathcal{F}(Y)}) \]

for all \( Y \in D \) and \( Y \in \overline{D} \)

- Since the concrete fixpoint \( \operatorname{lp}_D \mathcal{F} \) is well-defined, the corresponding iterates \( \mathcal{F}^\omega \) of \( \mathcal{F} \) are stationary at rank \( \epsilon \in \Omega \)

- We have seen in the well-defined theorem proof that the iterates for \( \operatorname{lp}_D \mathcal{G} \) are defined as:

\[ Z^0 \triangleq \top \]

\[ Z^{n+1} \triangleq Z^n \]

if \( \mathcal{F}(Z^n) \subseteq Z^n \) \hspace{1cm} (b)

\[ Z^n \uplus \mathcal{F}(Z^n) \]

otherwise \hspace{1cm} (c)

and proved using (W1), (W2) that they are ultimately stationary at rank \( \epsilon' < \omega \) and that \( \forall \delta \geq \epsilon' : Z^\delta = Z' = \operatorname{lp}_D \mathcal{G} \). We have:

- \( X^0 \subseteq \gamma(\overline{X}) \triangleq Z^0 \)
Theorem. If (LA1), (LA2) and (LA3) do hold, concrete fixpoints exist, (W1) and (W2) hold, then for all $i \in \Delta$ and $C_i \in \text{Com}_i$, we have

$$C[C_i] \subseteq C_i \cap C[C_i]$$

Proof. By structural induction on the well-founded relation $(\bigcup_{i \in \Delta} \text{Com}_i, \prec)$. Given any $i \in \Delta$ and $C_i \in \text{Com}_i$, assume by induction hypothesis that

$$\forall C_j \prec C_i : C[C_j] \subseteq C_C[C_j]$$

then

$$C[C_i]$$
Hypotheses on narrowings

Given a poset \( \langle L, \sqsubseteq \rangle \), a narrowing operator on \( L \) is \( \Delta \in L \times L \mapsto L \) satisfying

\[
\forall y \sqsubseteq x : y \sqsubseteq x \Delta y \sqsubseteq x
\]

\( (N1) \) For all sequences \( x^0, x^1, \ldots \) in \( L^\omega \), the sequence defined by \[
y^0 \overset{\text{def}}{=} x^0 \\
y^{n+1} \overset{\text{def}}{=} y^n \Delta x^n \quad \text{if} \quad x^n \sqsubseteq y^n \\
y^{n+1} \overset{\text{def}}{=} y^n \quad \text{otherwise}
\]
is not strictly increasing (although it is decreasing by \( (N1) \)).

Structural abstract semantics with widening/narrowing

- The structural definition is essential the same as the abstract semantics with widening (page 63), except for the use of a narrowing operator
- Narrowing: For each \( i \in \Delta \) and \( C_i \in \text{Com}_i \):
  \[
  \Delta_{C_i} \in \mathcal{D}_{C_i} \times \mathcal{D}_{C_i} \mapsto \mathcal{D}_{C_i} \text{ is a narrowing satisfying \( (N1) \) and \( (N2) \)}
  \]
- For fixpoints in the attribute grammar of expressions, we now have:

\[
\begin{align*}
\mathcal{E}[\mathcal{D}] \mathcal{E}(\mathcal{X}) & := \ldots \\
\text{let} \quad \mathcal{F} = \lambda Y. \mathcal{E}[\mathcal{D}] \mathcal{E}(\mathcal{X}, Y : \mathcal{D}) \quad \text{and} \\
\mathcal{G} = \lambda X. \text{let} \quad Y = \mathcal{F}(X) \quad \text{in} \quad (Y \sqsubseteq X \ ? X \ ? X \ \triangledown Y) \quad \text{and} \\
\mathcal{A} = \text{lfp} \mathcal{G} \quad \text{and} \\
\mathcal{H} = \lambda X. \text{let} \quad Y = \mathcal{F}(X) \quad \text{in} \quad (Y \sqsubseteq X \ ? X \ \Delta Y \ ? X) \quad \text{in} \\
\text{gfp} \mathcal{A} \mathcal{H}
\end{align*}
\]

Well-definedness of the structural abstract semantics with widening/narrowing

**Theorem.** A structural definition with widenings and narrowings respectively satisfying hypotheses \( (W1), (W2) \) and \( (N1), (N2) \) is well-defined.

**Proof.** - The proof, by structural induction, is essentially the same as in the previous case of “structural abstract semantics with widening”, but for the case of fixpoints
- For fixpoints, we have already shown in this proof that \( \mathcal{A} = \text{lfp} \mathcal{G} \) is well-defined as the limit of an increasing chain stabilizing, in a finite number of steps, at a postfixpoint: \( \mathcal{F}(\mathcal{A}) \sqsubseteq \mathcal{A} \).
If follows that the iterates \( \langle X^i, \delta \in \mathbb{Q} \rangle \) of \( \text{gfp}_{\mathcal{D}} \) are of the following form:

- If \( X^0 = \mathcal{A} \), where \( \mathcal{F}(\mathcal{A}) \subseteq \mathcal{A} \)
- \( X^{i+1} = X^i \setminus \text{F}(X^k) \), if \( \mathcal{F}(X^k) \subset X^k \)
- \( X^{i+1} = X^k \), otherwise
- \( X^k = \prod_{\delta < \omega} X^\delta \) when \( \lambda \) is a limit ordinal

- Observe that by def. of \( \mathcal{F} = \lambda Y \cdot e[D]\mathcal{S}(X, Y : \mathcal{D}) \), \( \mathcal{F} \) is well-defined by induction hypothesis
- By its def., the sequence \( \langle X^i, \delta < \omega \rangle \) is a decreasing chain, which is obvious in cases (c) and follow from (N1) in case (b)
- By (N2), the decreasing chain \( \langle X^i, \delta < \omega \rangle \) is not strictly decreasing so its is ultimately stationary at some rank \( \epsilon < \omega \)
- Because \( \epsilon < \omega \), the chain \( \langle X^i, \delta < \mathbb{Q} \rangle \) is well-defined since \( \lambda \geq \epsilon \) in case (d) implies that \( \prod_{\delta < \lambda} X^\delta \) is well-defined and indeed equal to \( X^\lambda = X^\epsilon \). So, by transfinite induction, \( \langle X^i, \delta < \mathbb{Q} \rangle \) is also well-defined and ultimately stationary at rank \( \epsilon \) and so \( \text{gfp}_{\mathcal{D}} = X^\epsilon \) is well-defined. \( \square \)

\begin{proof}

The proof is similar to the case of expressions with widenings (on page 73), except for the use of narrowings
- From this proof, we already know that by letting

\[
\mathcal{F} = \lambda Y \cdot e[D]\mathcal{S}(X, Y : \mathcal{D})
\]

we have \( \text{lfp}_{\mathcal{D}} \subseteq \gamma(\mathcal{A}) \).
- Let \( \langle X^i, \delta < \mathbb{Q} \rangle \) be the iterates for \( \mathcal{A} \). We have shown that they are well-defined and ultimately stationary at rank \( \epsilon \) such that \( \text{gfp}_{\mathcal{D}} = X^\epsilon \).
- We have

\[
\forall \delta \in \mathbb{Q} : \text{lfp}_{\mathcal{D}} \subseteq \gamma(X^\delta)
\]

The proof is by transfinite induction.

- We have \( \text{lfp}_{\mathcal{D}} \subseteq \gamma(\mathcal{A}) \) whence \( \text{lfp}_{\mathcal{D}} \subseteq \gamma(X^0) \) since \( X^0 = \mathcal{A} \)

\end{proof}

A soundness theorem on the correspondence between a concrete semantics and its local abstraction by a structural abstract semantics with widening/narrowing

**Theorem.** If \( e \) and \( \mathcal{E} \) are structurally identical, (LA1), (LA2) and (LA3) hold, concrete fixpoints exist, (W1), (W2), (N1) and (N2) hold, and
- \( S_j \subseteq \gamma(\mathcal{S}_j) \), \( j = 1, \ldots, n \) \hspace{1cm} (a)
- \( X_k \subseteq \gamma(\mathcal{X}_k) \), \( k = n + 1, \ldots, m \) \hspace{1cm} (b)

then

\[
e[D]\mathcal{S}(X) \subseteq \gamma(\mathcal{E}[D]\mathcal{S}(X))
\]

\begin{proof}

If \( \text{lfp}_{\mathcal{D}} \subseteq \gamma(X^\delta) \) by induction hypothesis, then

- if \( \mathcal{F}(X^k) \subseteq X^k \) then \( X^{k+1} = X^k \setminus \mathcal{F}(X^k) \), whence by (N1) \( X^k \subseteq X^{k+1} \)
  - whence \( \gamma(X^k) \subseteq \gamma(X^{k+1}) \) and so \( \text{lfp}_{\mathcal{D}} \subseteq \mathcal{F} \subseteq \gamma(X^{k+1}) \)
- otherwise, \( X^{k+1} = X^k \) and so \( \text{lfp}_{\mathcal{D}} \subseteq \gamma(X^{k+1}) \)
- If \( \lambda \) is a limit ordinal then we know that \( X^\lambda = X^\epsilon \) where \( \epsilon < \omega \leq \lambda \) and so \( \text{lfp}_{\mathcal{D}} \subseteq \mathcal{F} \subseteq \gamma(X^\lambda) \)

- We conclude that \( \text{lfp}_{\mathcal{D}} \subseteq \gamma(X^\delta) = \gamma(\text{gfp}_{\mathcal{D}}) \) whence \( e[D]\mathcal{S}(X) = \text{lfp}_{\mathcal{D}} \subseteq \mathcal{F} \subseteq \gamma(\text{gfp}_{\mathcal{D}}) = \mathcal{E}[D]\mathcal{S}(X) \) in that case.

\( \square \)

\end{proof}
**Theorem.** If (LA1), (LA2) and (LA3) do hold, concrete fixpoints exist, (W1), (W2), (N1) and (N2) hold, then for all $i \in \Delta$ and $C_i \in \text{Com}_i$, we have

$$C[C_i] \subseteq C_i \quad \forall C_i \in \text{Com}_i$$

**Proof.** Same as in the “structural abstract semantics with widening” case. □

---

**On monotony**

- The abstract structural definitions are not assumed to be monotone (because of the presence of widenings which are essentially not monotone)
- Nevertheless, they have been shown to be
  - well-defined
  - sound abstractions
  using “local abstraction conditions” only
- The proof is by structural induction on the programming language syntax, but formulated independently of any particular programming language

---

**On the use of widening/narrowing**

- In lattices satisfying the ACC, one can chose $x \vee y = x \cup y$ and $x \wedge y = x \cap y$
- In case of monotony and iteration form a pre/postfixpoint, one prefers $x \vee y = y$ and $x \wedge y = y$

---

**An abstract formalization of structural verification by abstract interpretation**
Structural safety specification

- We consider a language \( \mathcal{L} = \bigcup_{i \in \Delta} \text{Com}_i, \prec \) with syntactic components \( C_i \in \text{Com}_i \) and well-founded "immediate subcomponent relation" \( \prec \).
- The concrete semantics is given for all \( i \in \Delta, C_i \in \text{Com}_i \) by:
  - \( \langle \mathcal{P}_{C_i}, \sqsubseteq_{C_i}, \sqsubseteq_{C_i}, \sqsubseteq_{C_i} \rangle \) concrete semantic domain \( (a) \)
  - \( C[C_i] \in \mathcal{P}_{C_i} \) concrete semantics \( (b) \)
- A safety specification is:
  \( S : C_i \in \text{Com}_i \mapsto \mathcal{D}_{C_i}, i \in \Delta \) \( (c) \)

Structural abstract safety specification and proof

- An abstract safety specification is:
  - \( \langle \mathcal{D}_{C_i}, \mathcal{T}_{C_i}, \mathcal{I}_{C_i}, \mathcal{I}_{C_i} \rangle \) abstract domains \( (e) \)
  - \( \mathcal{S}[C_i] \in \mathcal{D}_{C_i} \) abstract spec. \( (f) \)
  - \( \mathcal{C}_{C_i} \in \mathcal{D}_{C_i} \mapsto \mathcal{D}_{C_i} \) spec. concretization \( (g) \)
- An abstract safety proof is the proof that:
  \( \forall i \in \Delta : \forall C_i \in \text{Com}_i : C[C_i] \sqsubseteq_{C_i} \mathcal{S}[C_i] \) \( (d) \)

Structural safety proof

- A safety proof is the proof that:
  \( \forall i \in \Delta : \forall C_i \in \text{Com}_i : C[C_i] \sqsubseteq_{C_i} S[C_i] \) \( (d) \)
- Informally: the semantics of commands satisfies their specification

Abstract semantics

- An abstract safety verification by abstract interpretation consists in designing an abstract semantics for all \( i \in \Delta, C_i \in \text{Com}_i \):
  - \( \langle \mathcal{D}_{C_i}, \mathcal{T}_{C_i}, \mathcal{I}_{C_i}, \mathcal{I}_{C_i} \rangle \) abstract semantic domain \( (i) \)
  - \( C[C_i] \in \mathcal{D}_{C_i} \) abstract semantics \( (j) \)
  - \( \mathcal{C}_{C_i} \in \mathcal{D}_{C_i} \mapsto \mathcal{D}_{C_i} \) concretization \( (k) \)
which are sound, in that
\( \forall i \in \Delta : \forall C_i \in \text{Com}_i : C[C_i] \sqsubseteq_{C_i} \mathcal{C}_{C_i}(\mathcal{S}[C_i]) \) \( (\ell) \)
and effectively computable (thanks to the choice of computer representable abstract domains, transfer functions and widening/narrowing)
Choice of the abstractions

- The abstract domains $\mathcal{D}_i$, $\mathcal{E}_i$, $\mathcal{I}_i$, $\mathcal{S}_i$ are chosen to be more precise than the abstract specification domains $\mathcal{D}_i$, $\mathcal{E}_i$, $\mathcal{I}_i$, $\mathcal{S}_i$.

- This can be formalized by the existence of concretizations:

  \[ \mathcal{D}_i \subseteq \mathcal{D}_i \]

  satisfying

  \[ \mathcal{E}_i \supseteq \mathcal{E}_i \mathcal{I}_i \mathcal{D}_i \]

Soundness of the abstract safety specification

\[ \text{THEOREM. An abstract structural safety verification is sound.} \]

\[ \text{PROOF. By the abstract check (o), we have } \mathcal{E}_i \subseteq \mathcal{E}_i \mathcal{I}_i \mathcal{D}_i \mathcal{S}_i \]

\[ \text{whence by monotony (m) } \mathcal{E}_i \subseteq \mathcal{E}_i \mathcal{I}_i \mathcal{S}_i \mathcal{D}_i \mathcal{S}_i \]

\[ \text{and so by (n) and transitivity, } \mathcal{E}_i \subseteq \mathcal{E}_i \mathcal{I}_i \mathcal{S}_i \mathcal{D}_i \mathcal{S}_i \]

\[ \text{whence, by soundness (t) of the abstraction, we conclude } \mathcal{E}_i \subseteq \mathcal{E}_i \mathcal{I}_i \mathcal{S}_i \mathcal{D}_i \mathcal{S}_i \], proving soundness. \[ \square \]

Abstract structural safety verification

- The abstract safety verification consists in checking that:

  \[ \mathcal{E}_i \subseteq \mathcal{E}_i \mathcal{I}_i \mathcal{D}_i \mathcal{S}_i \]

Example of structural safety specification for arithmetic expressions: absence of runtime errors

- The execution of an arithmetic expression $A$ in any environment $\rho \in R \subseteq (\text{Var}[P] \rightarrow \text{I}_R)$ is without any runtime error if and only if

  \[ \text{Faexp}[A] R \cap \text{E} = \emptyset \]

  \[ \iff \text{Faexp}[A] R \subseteq \text{I} \]

where $\text{Faexp}$ is the forward collecting semantics of arithmetic expressions.
Example of too imprecise abstraction

- Observe that this cannot be checked with the initialization and simple sign abstraction:

\[
\begin{align*}
\gamma(\text{BUT}) & \equiv \{ \Omega_a \} \\
\gamma(\text{NEG}) & \equiv \text{min}_{\text{int}} , -1 \cup \{ \Omega_a \} \\
\gamma(\text{INIT}) & \equiv \Omega_a \\
\gamma(\text{ERR}) & \equiv \{ \Omega_1, \Omega_a \} \\
\gamma(\text{ZERO}) & \equiv \{ 0, \Omega_a \} \\
\gamma(\text{POS}) & \equiv \text{max}_{\text{int}} \cup \{ \Omega_a \}
\end{align*}
\]

since defining \( \widehat{\gamma}_A \) satisfying (m) and (n) is impossible since \( \Omega_a \notin \widehat{\gamma}_A(\llbracket \text{I} \rrbracket)(R) \)

- We can only strengthen the analysis by refining the abstraction or weaken the specification

Example of concrete structural safety specification for arithmetic expressions: proper initialization

- The execution of an arithmetic expression \( A \) in any environment \( \rho \in R \subseteq (\text{Var} \llbracket P \rrbracket \mapsto \llbracket \text{I} \rrbracket) \) is without any initialization error if and only if

\[
\begin{align*}
\text{Faexp}[A] R \cap \{ \Omega_1 \} &= \emptyset \\
\iff \text{Faexp}[A] R \subseteq \text{I} \cup \{ \Omega_a \}
\end{align*}
\]

where \( \text{Faexp} \) is the forward collecting semantics of arithmetic expressions, so we define in that case the concrete specification

\[
\begin{align*}
\llbracket A \rrbracket \equiv \lambda R \cdot \text{I} \cup \{ \Omega_a \} \quad \text{or} \\
\equiv \lambda R \cdot (P(R) \cap \text{I} \cup \{ \Omega_a \}) \cup \llbracket \text{I} \rrbracket
\end{align*}
\]

if we want to check absence initialization error under the hypothesis that some condition \( P(R) \) holds on the precondition \( R \)
Example of abstract structural safety specification for arithmetic expressions: proper initialization

- An abstract safety specification is
  \[ \hat{D}_A = \top \]
  \[ \hat{S}[A] = \text{INI} \]
- \( \hat{\gamma}(\top) = \lambda R.\top \), \( \hat{\gamma}(\text{INI}) = \lambda R.\top \cup \{\omega\} \)
- \( \hat{S}[A] = \text{INI} \)

The abstract safety verification condition is:

\[ \gamma(A[\hat{\gamma}, A(\hat{S}[A])]) \quad \iff \quad \gamma(A[\hat{\gamma}, \lambda R.\text{INI}]) \]

which implies

\[ \gamma(\text{Faexp}[A]) \subseteq \gamma(\lambda R.\text{INI}) \]

whence

\[ \forall R : \text{Faexp}[A]R \subseteq \top \cup \{\omega\} \]

as required

We have shown the abstract interpretation of arithmetic expressions to be sound \( \text{Faexp}^b[A] \supseteq \alpha^b(\text{Faexp}[A]) \) or equivalently \( \text{Faexp}[A] \subseteq \gamma^b(\text{Faexp}^b[A]) \)

- We define
  \[ \hat{\gamma}(\top) \overset{\text{def}}{=} \lambda R.\top \]
  \[ \hat{\gamma}(\text{INI}) \overset{\text{def}}{=} \lambda R.\text{INI} \]

so that

\[ \hat{\gamma}(A) = \gamma^b \circ \gamma_A \]

Why choosing abstract specifications?

- The objective is to check the conformance of a semantics to a specification:
  \[ \text{Sem} \sqsubseteq \text{Spec} \] (a)

- We want to perform the check in the abstract:
  \[ \text{Sem}^b \sqsubseteq \text{Spec}^b \] (b)

so that it implies in the concrete:

\[ \gamma(\text{Sem}^b) \subseteq \gamma(\text{Spec}^b) \] (c)

- For (c) to imply (a) we need both:
Sem ⊆ τ(Sem^b) and τ(Spec^b) ⊆ Spec    (1)

- Sem ⊆ τ(Sem^b) is an approximation from above, which is pretty well studied
- τ(Spec^b) ⊆ Spec is an approximation from below for which only finite abstractions are known to be automatable, while specifications are most often infinite!

Why choosing an abstract semantics more refined than an abstract specifications?

- The fact that the abstract semantics should be more refined than the abstract specification is similar to the proof of theorem requiring stronger arguments in the proof
- For example, with Floyd’s method

\[ \text{lfp}_{\mathcal{D}} F \subseteq P \iff \exists I : F(I) \subseteq I \land I \subseteq P \]

\( P \) is invariant while the proof requires to find a stronger inductive invariant (while, in general \( F(P) \not\subseteq P \))

- By choosing abstract specifications only, we solve the problem by choosing

\[ \text{Spec} = \tau(\text{Spec}^b) \]

but we are left with the problem of finding adequate machine representations of the specifications as abstract domain

- Progress is necessary in the abstraction of specifications from below!

- Similarly we can always choose the abstraction \( \mathcal{D}_A = \mathcal{D}_A \) as a starting point, but in general refinements are needed

- While in Floyd’s method or abstract model checking this refinement is done for a particular program, the difficulty in this refinement must be done for a language
Principle of a structural static analyzer/verifier

Example of abstract domain: Error analysis

Error abstraction

- Takes initialization and arithmetic errors into account

- The abstract properties are \( \langle E, \subseteq E \rangle \) where \( E \equiv \{\text{NER}, \text{AER}, \text{IER}, \text{ERR}\} \) and the partial order is defined by the Hasse diagram
- Concretization:

\[
\begin{align*}
\gamma_E(\text{NER}) & \overset{\text{def}}{=} \mathbb{I} \\
\gamma_E(\text{AER}) & \overset{\text{def}}{=} \mathbb{I} \cup \{\Omega_a\} \\
\gamma_E(\text{IER}) & \overset{\text{def}}{=} \mathbb{I} \cup \{\Omega_1\} \\
\gamma_E(\text{ERR}) & \overset{\text{def}}{=} \mathbb{I} \cup \{\Omega_1, \Omega_a\} = \mathbb{I}_\Omega
\end{align*}
\]
The error complete lattice

The finite lattice

is obviously a complete lattice, with

- Partial ordering: NER \sqsubseteq E AER \sqsubseteq E AER ERR \sqsubseteq E ERR and NER \sqsubseteq E IER \sqsubseteq E IER \sqsubseteq E ERR
- lub: x \sqcup_E x = x AER \sqcup_E IER = ERR NER \sqcup_E x = x x \sqcup_E y = y \sqcup_E x ERR \sqcup_E x = ERR
- glb: x \sqcap_E x = x AER \sqcap_E IER = NER NER \sqcap_E x = NER x \sqcap_E y = y \sqcap_E x ERR \sqcap_E x = x
- \infimum: NER
- \supremum: ERR

Proof. To prove \( \alpha(x) \subseteq E y \iff x \subseteq \gamma_E(y) \), we consider 4 cases for \( y = NER, y = AER, y = IER \) and \( y = ERR \). Since all cases are very similar and the proof is tedious, we consider only the case \( y = AER \) and prove \( \alpha(x) \subseteq E AER \iff x \subseteq \gamma_E(AER) \).

- If \( \alpha(x) \subseteq E AER \) then either \( \alpha(x) = NER \) or \( \alpha(x) = AER \)
  - If \( \alpha(x) = NER \) then \( x \subseteq I = \gamma_E(NER) \)
  - Else \( \alpha(x) = AER \) and then \( x \subseteq I \cup \{\Omega_a\} = \gamma_E(AER) \)
- Reciprocally, if \( x \subseteq \gamma_E(AER) \) then \( x \subseteq \sqcup \{\Omega_a\} \).
  - If \( x \subseteq I \) then \( \alpha(x) = NER \subseteq E AER \)
  - Otherwise \( \alpha(x) = AER \subseteq E AER \)

\[ \square \]

The error analysis abstract domain

We have defined

\[ \gamma_E(NER) \overset{\text{def}}{=} \mathbb{I} \]
\[ \gamma_E(AER) \overset{\text{def}}{=} \mathbb{I} \cup \{\Omega_a\} \]
\[ \gamma_E(IER) \overset{\text{def}}{=} \mathbb{I} \cup \{\Omega_i\} \]
\[ \gamma_E(ERR) \overset{\text{def}}{=} \mathbb{I} \cup \{\Omega_1, \Omega_2\} = \mathbb{I}_E \]

we let

\[ \alpha_E(X) \overset{\text{def}}{=} \begin{cases} \mathbb{I} & \text{NER} \\ \mathbb{I} \cup \{\Omega_a\} & \text{AER} \\ \mathbb{I} \cup \{\Omega_i\} & \text{IER} \\ \mathbb{I} \cup \{\Omega_1, \Omega_2\} = \mathbb{I}_E & \text{ERR} \end{cases} \]

and we have

\[ \langle \alpha_E(\mathbb{I}_E), \subseteq \rangle \overset{\gamma_E}{\xrightarrow{\alpha_E}} \langle E, \subseteq_E \rangle \]
let initerr () = IER
let top () = ERR
let nat_of_lat u =
  match u with
  | NER -> 0
  | AER -> 1
  | IER -> 2
  | ERR -> 3
let select t u v = t.(nat_of_lat u).(nat_of_lat v)
let join_table =
  (* NER AER IER ERR *)
  (*NER*)
  [ [ NER ; AER ; IER ; ERR ; | ];
  (*AER*)
  [ | AER ; AER ; ERR ; ERR ; | ];
  (*IER*)
  [ | IER ; ERR ; IER ; ERR ; | ];
  (*ERR*)
  [ | ERR ; ERR ; ERR ; ERR ; | ]
let join u v = select join_table u v
let meet_table =
  (* NER AER IER ERR *)
  (*NER*)
  [ [ NER ; NER ; NER ; NER ; | ];
  (*AER*)
  [ | NER ; AER ; NER ; AER ; | ];
  (*IER*)
  [ | NER ; NER ; IER ; IER ; | ];
  (*ERR*)
  [ | NER ; AER ; IER ; ERR ; | ]
let meet u v = select join_table u v
let leq_table =
  (* NER AER IER ERR *)
  (*NER*)
  [ [ true ; true ; true ; true ; | ];
  (*AER*)
  [ | false ; true ; false ; true ; | ];
  (*IER*)
  [ | false ; false ; true ; true ; | ];
  (*ERR*)
  [ | false ; false ; false ; true ; | ]
let leq u v = select leq_table u v
let in_errors v = (leq v ERR)
let leq u v = (u = v)

let f_NAT s = (machine_int_of_string s)
let f_RANDOM () = NER
let f_UMINUS a = match a with
  | NER -> AER (* a can be min_int *)
  | AER -> AER
  | IER -> IER
  | ERR -> ERR
let f_UPLUS a = a
let f_BINARITH a b = (machine_binary_binarith i j) | i in gamma(a) \ j in gamma(b) | *}

let leq u v = (u = v)

let f_PLUS = f_BINARITH
let f_MINUS = f_BINARITH
let f_TIMES = f_BINARITH
let f_DIV = f_BINARITH

(* forward abstract semantics of boolean expressions *)
let f_EQ u v = true
let f_LT u v = true

(* widening *)
let widen v w = w

(* narrowing *)
let narrow v w = w

(* backward abstract semantics of arithmetic expressions *)

let b_NAT s p = (machine_int_of_string s) in gamma(v) cap I?
let b_RANDOM p = gamma(p) cap I <> emptyset
let b_UMINUS q p = NER
let b_UPLUS q p = NER
let b_TIMES q1 q2 p = NER
let b_DIV q1 q2 p = NER
let b_MOD q1 q2 p = NER

(* backward abstract interpretation of boolean expressions *)
let a_EQ p1 p2 = let p = p1 cap p2 cap [min_int, max_int] in <p, p>
let a_LT p1 p2 = alpha {{i1, i2} | i1 in gamma(p1) cap [min_int, max_int] \ i1 <= i2}

Example of abstract domain: Parity analysis
The parity analysis abstract domain

(* _values.ml _*)
open Values
(* abstraction of sets of machine integers by parity *)
type t = BOT | ODD | EVEN | TOP
(* complete lattice *)
let bot () = BOT
let isbotempty () = false
let initerr () = TOP
(* supremum *)
let top () = TOP
(* least upper bound *)
(* greatest lower bound *)
(* approximation ordering *)
let leq v w =
let eq u v = (u = v)
(* included in errors? *)
let in_errors u = (u = BOT)
(* printing *)
let print u =
match u with
| BOT -> print_string "_|_
| ODD -> print_string "O"
| EVEN -> print_string "e"
| TOP -> print_string "T"

(* forward abstract semantics of arithmetic expressions *)
let pry_of_intstring i s =
let l = (String.length s) in
if l = 0 then
  if (i mod 2) = 0 then EVEN else ODD
else
  let v = (10 * i) + (int_of_string (String.sub s 0 1)) in
  if v < i then (* overflow *)
    BOT (* = alpha({_O_(a)}) *)
  else
    pry_of_intstring v (String.sub s 1 (l-1))
let parity_of_intstring i = pry_of_intstring 0 i
let f_NAT i = parity_of_intstring i
let f_RANDOM () = TOP
let pry_of_intstring v (String.sub s 1 (l-1))
let parity_of_intstring i = pry_of_intstring 0 i
let f_NAT i = parity_of_intstring i
let f_RANDOM () = TOP
(* _uminus a = alpha((machine_unary_minus x) | x \in gamma(a)) *)
69  let f_MINUS u = u
70  (* f_IPLUS a = alpha(gamma(a)) *)
71  let f_UPLUS a = a
72  (* f_BINARITH a b = alpha({ (machine_binary_binarith i j)| } | *)
73  (* i in gamma(a) \ j \in gamma(b)) *)
74  let nat_of_lat u =
75     match u with
76     | BOT -> 0
77     | ODD -> 1
78     | EVEN -> 2
79     | TOP -> 3
80  let select t u v = t.(nat_of_lat u). (nat_of_lat v)
81  let f_PLUS_table =
82     (* + BOT ODD EVEN TOP *)
83     (*BOT*)[| | BOT ; BOT ; BOT ; BOT |];
84     (*ODD*)[| | BOT ; EVEN ; ODD ; TOP |];
85     (*EVEN*)[| | BOT ; ODD ; EVEN ; TOP |];
86     (*TOP*)[| | BOT ; TOP ; TOP ; TOP |]]
87  let f_PLUS u v = select f_PLUS_table u v
88  let f_MINUS = f_PLUS
89  let f_TIMES_table =
90     (* * BOT ODD EVEN TOP *)
91     (*BOT*)[| | BOT ; BOT ; BOT ; BOT |];
92     (*ODD*)[| | BOT ; EVEN ; EVEN ; TOP |];
93     (*EVEN*)[| | BOT ; EVEN ; EVEN ; TOP |];
94     (*TOP*)[| | BOT ; TOP ; TOP ; TOP |]]
95  let f_TIMES u v = select f_TIMES_table u v
96  let f_DIV_table =
97     (* / BOT ODD EVEN TOP *)
98     (*BOT*)[| | BOT ; BOT ; BOT ; BOT |];
99     (*ODD*)[| | BOT ; TOP ; TOP ; TOP |];
100    (*EVEN*)[| | BOT ; TOP ; TOP ; TOP |];
101    (*TOP*)[| | BOT ; TOP ; TOP ; TOP |]]
102  let f_DIV u v = select f_DIV_table u v
103  let f_MOD = f_DIV
104  (* forward abstract semantics of boolean expressions *)
105  (* Are there integer values in gamma(u) equal to values in gamma(v)? *)
106  let f_EQ u v = (u = TOP) || (v = TOP) || ((u = v) & (u != BOT))
107  (* Are there integer values in gamma(u) less than or equal to (<=) *)
108  (* integer values in gamma(v)? *)
109  let f_LT u v = ((u != BOT) & (v != BOT))
110  (* widening *)
111  let widen v w = w
112  (* narrowing *)
113  let narrow v w = w
114  (* backward abstract semantics of arithmetic expressions *)
115  (* b_NAT s v = (machine_int_of_string s) in gamma(v) cap I? *)
116  exception Error_b_NAT of string
117  let b_NAT n p =
118     match (String.get n (String.length n - 1)) with
119     | '0' -> leq EVEN p
120     | '1' -> leq ODD p
121     | '2' -> leq EVEN p
122     | '3' -> leq ODD p
123     | '4' -> leq EVEN p
124     | '5' -> leq ODD p
125     | '6' -> leq EVEN p
126     | '7' -> leq ODD p
127     | '8' -> leq EVEN p
128     | '9' -> leq ODD p
129     | _ -> raise (Error_b_NAT "not a digit")
130  (* b_RANDOM p = gamma(p) cap \[min_int, max_int\] *)
131  let b_RANDOM p =
132     match p with
133     | BOT -> false
134     | _ -> true
135  (* backward abstract semantics of arithmetic expressions *)
136  (* b_NAT s v = (machine_int_of_string s) in gamma(v) cap *)
137  (* [min_int, max_int]? *)
138  let b_UMINUS q p = meet q p
139  let b_UPLUS q p = meet q p
140  (* b_BOP q1 q2 p = alpha2({<i1,i2> in gamma2(<q1,q2>) | *)
exception Error\_b\_PLUS of string

let nat\_of\_lat' u =
  match u with
  | ODD -> 0
  | EVEN -> 1
  | TOP -> 2
  | _ -> raise (Error\_b\_PLUS "impossible selection")

let select' t u v =
  t.(nat\_of\_lat' u).(nat\_of\_lat' v)

let b\_PLUS\_ODD\_table =
  (* ODD EVEN TOP *)
  (* ODD *)
  | (BOT,BOT) ; (ODD,EVEN) ; (ODD,EVEN) |
  (* EVEN *)
  | (EVEN,ODD) ; (BOT,BOT) ; (EVEN,ODD) |
  (* TOP *)
  | (EVEN,ODD) ; (ODD,EVEN) ; (TOP,TOP) |

let b\_PLUS\_EVEN\_table =
  (* ODD EVEN TOP *)
  (* ODD *)
  | (ODD,ODD) ; (BOT,BOT) ; (ODD,ODD) |
  (* EVEN *)
  | (BOT,BOT) ; (EVEN,EVEN) ; (EVEN,EVEN) |
  (* TOP *)
  | (ODD,ODD) ; (BOT,BOT) ; (ODD,ODD) |

let b\_PLUS \_q1 \_q2 \_p =
  if (q1=BOT)||(q2=BOT)||(p=BOT) then
    (BOT,BOT)
  else if (p=TOP) then
    (q1,q2)
  else if p = ODD then select' b\_PLUS\_ODD\_table q1 q2
  else if p = EVEN then select' b\_PLUS\_EVEN\_table q1 q2
  else raise (Error\_b\_PLUS "impossible case")

let b\_MINUS = b\_PLUS

let b\_PLUS\_ODD\_table =
  (* ODD EVEN TOP *)
  (* ODD *)
  | (BOT,BOT) ; (ODD,EVEN) ; (ODD,EVEN) |
  (* EVEN *)
  | (EVEN,ODD) ; (BOT,BOT) ; (EVEN,ODD) |
  (* TOP *)
  | (EVEN,ODD) ; (ODD,EVEN) ; (TOP,TOP) |

let b\_PLUS\_EVEN\_table =
  (* ODD EVEN TOP *)
  (* ODD *)
  | (ODD,ODD) ; (BOT,BOT) ; (ODD,ODD) |
  (* EVEN *)
  | (BOT,BOT) ; (BOT,BOT) ; (BOT,BOT) |
  (* TOP *)
  | (ODD,ODD) ; (BOT,BOT) ; (ODD,ODD) |

exception Error\_b\_TIMES of string

let b\_TIMES\_q1 \_q2 \_p =
  if (q1=BOT)||(q2=BOT)||(p=BOT) then
    (BOT,BOT)
  else if (p=TOP) then
    (q1,q2)
  else if p = ODD then select' b\_TIMES\_ODD\_table q1 q2
  else if p = EVEN then select' b\_TIMES\_EVEN\_table q1 q2
  else raise (Error\_b\_TIMES "impossible case")

let b\_DIV q1 q2 p =
  if (q1=BOT)||(q2=BOT)||(p=BOT) then
    (BOT,BOT)
  else if (p=TOP) then
    (q1,q2)
  else if p = ODD then select' b\_TIMES\_ODD\_table q1 q2
  else if p = EVEN then select' b\_TIMES\_EVEN\_table q1 q2
  else raise (Error\_b\_TIMES "impossible case")

let b\_MOD = b\_DIV

let b\_TIMES\_ODD\_table =
  (* ODD EVEN TOP *)
  (* ODD *)
  | (ODD,ODD) ; (BOT,BOT) ; (ODD,ODD) |
  (* EVEN *)
  | (BOT,BOT) ; (BOT,BOT) ; (BOT,BOT) |
  (* TOP *)
  | (ODD,ODD) ; (BOT,BOT) ; (ODD,ODD) |

let b\_TIMES\_EVEN\_table =
  (* ODD EVEN TOP *)
  (* ODD *)
  | (ODD,ODD) ; (BOT,BOT) ; (ODD,ODD) |
  (* EVEN *)
  | (BOT,BOT) ; (EVEN,EVEN) ; (EVEN,TOP) |
  (* TOP *)
  | (EVEN,ODD) ; (TOP,EVEN) ; (TOP,TOP) |

exception Error\_b\_TIMES of string

let a\_EQ p1 p2 =
  if p1 = p2 cap (min\_int, max\_int) then
    (p1,p1)
  else raise (Error\_a\_EQ "impossible case")

let a\_LT_table =
  (* < BOT ODD EVEN TOP *)
  (* BOT *)
  | (BOT,BOT); (BOT,BOT) ; (BOT,BOT) ; (BOT,BOT) |
  (* ODD *)
  | (ODD,ODD) ; (ODD,EVEN) ; (ODD,EVEN) |
  (* EVEN *)
  | (EVEN,ODD) ; (EVEN,EVEN) ; (EVEN,EVEN) |
  (* TOP *)
  | (EVEN,ODD) ; (EVEN,EVEN) ; (EVEN,TOP) |

let a\_LT u v = select a\_LT\_table u v
Parity analysis example

** Input file:

```
x := -1073741823 -1;
y := x - 1;
{ x:T; y:T }
0:
x := (-1073741823 - 1);
1:
y := (x - 1)
2:
{ x:e; y:o }
```

Example of abstract domain: Interval analysis

Design of the abstract properties

Interval abstractions
– In the traditional lattice for interval analysis [1], a supremum $+\infty$ and an infimum $-\infty$ are added to reason on the complete lattice $\langle \mathbb{Z} \cup \{-\infty, +\infty\}, \leq \rangle$ [1]
– This is appropriate for mathematical, machine-independent reasoning only
– In practice we have $+\infty = \max\_\text{int}$ and $-\infty = \min\_\text{int}$ to take the finite machine representation of integers into account: $\langle \{z \in \mathbb{Z} | -\infty \leq z \leq +\infty\}, \leq \rangle$ that is $\langle \mathbb{I}, \leq \rangle$
– The abstract properties are $I \overset{\text{def}}{=} \{\bot\} \cup \{[a, b] \mid a, b \in \mathbb{I} \land -\infty \leq a \leq b \leq +\infty\}$

### Error and interval abstraction

– Combine interval and error information
– The lattice of program properties is $I \times E$
– The concretization is $\gamma((i, e)) \overset{\text{def}}{=} (\gamma_i(i) \cup \{\Omega_1, \Omega_2\}) \cap \gamma_E(e)$
– Intervals bring no information of errors
– Errors bring no range information
– The combination provide both range and error information
– This is an example of reduced product

### The partial order of intervals

– For intervals we let $+\infty = \max\_\text{int}$ and $-\infty = \min\_\text{int}$ so that $\forall i \in \mathbb{I} : -\infty \leq i \leq +\infty$
– The Hasse diagram defines the interval abstract properties is $I \overset{\text{def}}{=} \{\bot\} \cup \{[a, b] \mid a, b \in \mathbb{I} \land -\infty \leq a \leq b \leq +\infty\}$
– The Hasse diagram defines the interval partial order $\sqsubseteq_I$ as

\[
\forall x \in I : \bot \sqsubseteq_I i \\
\forall [a, b], [c, d] \in I : ([a, b] \sqsubseteq_I [c, d]) \iff (a \leq c \leq d \leq b)
\]
Theorem. \( \langle I, \subseteq I \rangle \) is a partial order.

Proof. By def. of \( \subseteq I \) and reflexivity of \( \leq I \), \( \subseteq I \) is reflexive
- If \( i \subseteq I j \) and \( j \subseteq I k \) then
- If \( i = \bot \) then \( i = \bot \subseteq I k \)
- Else \( i = [a, b] \) so \( j = [c, d] \) so \( k = [e, f] \). By def. of \( \subseteq I \), \( i \subseteq I j \) implies \( c \leq a \leq b \leq d \) and \( j \subseteq I k \) implies \( e \leq c \leq d \leq f \) so that by transitivity of \( \leq I \), we get \( c \leq a \leq b \leq f \) proving \( i \subseteq I k \)
In both cases \( i \subseteq I k \) proving transitivity
- If \( i \subseteq I j \) and \( j \subseteq I i \) then
- If \( i = \bot \) then \( j = \bot \) by def. of \( \subseteq I \) so \( i = j \)
- If \( i = [a, b] \) then \( j = [c, d] \) since \( i \subseteq I j \) by def. of \( \subseteq I \). This implies \( c \leq a \leq b \leq d \). \( j \subseteq I i \) implies \( a \leq b \leq d \leq b \) so by antisymmetry of \( \leq I \), we get \( a = c \subseteq I b = d \) so \( i = j \).
In both cases \( i = j \) proving antisymmetry.
- We conclude that \( \subseteq I \) is a partial order on \( I \)

The complete lattice of intervals
- \( \langle I, \subseteq I, \bot, [-\infty, \infty], \cup_I, \cap_I \rangle \) is a complete lattice, where:
  - \( \bigcup_I i \_J \_J = \bigcup \{i_j \mid j \in \Delta \land i_j \neq \bot\} \)
  - \( \bigcup_I 0 = \bot \)
  - \( \bigcup_I [a_j, b_j] = [\min \_J a_j, \max \_J b_j] \) where min and max are extended on \( \cup_I \) to \(-\infty\) and \(+\infty\) in the natural way

Theorem. \( \cup_I \) is the lub.
Interval abstraction

- We have defined $\gamma_i \in I \mapsto \wp(\mathbb{I})$ as
  \[
  \gamma_i(\bot) \overset{\text{def}}{=} \emptyset \\
  \gamma_i([a, b]) \overset{\text{def}}{=} \{ z \in \mathbb{I} | a \leq z \leq b \}
  \]

- Given $X \subseteq \mathbb{I}$, we define
  \[
  \alpha_i(\emptyset) \overset{\text{def}}{=} \bot \\
  \alpha_i(X) \overset{\text{def}}{=} [\min_X x, \min_X z], \quad X \neq \emptyset
  \]

- We have the Galois connection:
  \[
  \langle \wp(\mathbb{I}), \subseteq \rangle \sqsubseteq \alpha_i \quad \alpha_i^{-1} \quad \langle I, \subseteq \rangle
  \]

Proof. We prove that $\alpha_i(x) \subseteq y \iff x \subseteq \gamma_i(y)$.

- If $x = \emptyset$ then $\alpha_i(\emptyset) \overset{\text{def}}{=} \bot \subseteq y$ and $x = \emptyset \subseteq \gamma_i(y)$ is true for all $y \in I$.

- If $y$ is $\bot$ then $\alpha_i(x) \subseteq y$ implies $\alpha_i(x) = \bot$ whence $x = \emptyset$ by def. $\alpha_i$, and so $x = \emptyset \subseteq \emptyset = \gamma_i(\bot)$.

  Reciprocally, if $x \subseteq \gamma_i(y)$ then $x \subseteq \emptyset$ so $x = \emptyset$ proving $\alpha_i(x) = \bot \subseteq y$ by def. $\subseteq_I$.

- If $x$ is not $\bot$ and $y$ is not $\bot$ then $y = [a, b]$ with $-\infty \leq a \leq b \leq +\infty$ by def. $I$. If $\alpha_i(x) \subseteq y$ then $[\min_{\mathbb{I}} x, \max_{\mathbb{I}} x] \subseteq [a, b]$ so $a \leq \min_{\mathbb{I}} x \leq \max_{\mathbb{I}} x \leq b$ by def. $\subseteq_I$, proving $x \subseteq \{ z \in I | a \leq z \leq b \} = \gamma_i([a, b]) = \gamma_i(y)$.

  Reciprocally, $x \subseteq_I y$ implies $x \subseteq \{ z \in I | a \leq z \leq b \}$ which implies $a \leq \min_{\mathbb{I}} x \leq \max_{\mathbb{I}} x \leq b$ that is $[\min_{\mathbb{I}} x, \max_{\mathbb{I}} x] \subseteq [a, b]$ i.e. $\alpha_i(x) \subseteq_I y$.

\[\square\]
- By definition, we have immediately:

\[ \gamma_I(\perp) = \{\Omega_a, \Omega_1\} \]
\[ \gamma_I([a,b]) = \{x \in I \mid a \leq x \leq b\} \cup \{\Omega_a, \Omega_1\} \]
\[ \alpha_I(X) = \{X \subseteq \{\Omega_a, \Omega_1\} \mid \perp \in [\min_I x \setminus \{\Omega_a, \Omega_1\}, \max_I x \setminus \{\Omega_a, \Omega_1\}] \} \]

The reduced product of abstractions

If
- \( \langle L, \sqsubseteq, \sqcup \rangle \) is a meet semilattice,
- \( \langle L, M \rangle \xrightarrow{\gamma_1} \langle M_1, \leq_1 \rangle \)
- \( \langle L, M \rangle \xrightarrow{\gamma_2} \langle M_2, \leq_2 \rangle \)

then their reduced product \([4]\) is

\[ \langle L, M \rangle \xrightarrow{\gamma} \langle M, \leq \rangle \]

where

\[ \gamma((x, y)) = \gamma_1(x) \cap \gamma_2(y) \]
\[ \langle x, y \rangle \equiv \langle x', y' \rangle \iff \gamma((x, y)) = \gamma((x', y')) \]

\[ \mathcal{M} \overset{\text{def.}}{=} (M_1 \times M_2) \]

\[ \alpha(x) \overset{\text{def.}}{=} [(\alpha_1(x), \alpha_2(x))] \]

- \([\langle x, y \rangle]) \subseteq \langle \langle x', y' \rangle \rangle \iff \exists \langle x_1, y_1 \rangle \in \langle \langle x, y \rangle \rangle \Rightarrow \exists \langle x_2, y_2 \rangle \in \langle \langle x', y' \rangle \rangle \Rightarrow x_1 \leq x_2 \land y_1 \leq y_2 \]

\[ \gamma([\langle x, y \rangle]) = \gamma((x, y)) = \gamma_1(x) \cap \gamma_2(y) \]

Proof.
- the definition of \(\gamma([\langle x, y \rangle])\) is obviously independent of the choice of the representant \(\langle x, y \rangle\) of the equivalence class \([\langle x, y \rangle] = \{(x_1, x_2) \mid \gamma((x, y)) = \gamma((x_1, x_2))\}\). This remark is also valid for the definition of \(\leq\).

\[ \alpha(X) \subseteq \langle [x, y] \rangle = \]
\[ \Rightarrow \exists \langle x_1, \alpha(X) \rangle \in \langle [x, y] \rangle \overset{\text{def. } \alpha}{=} \]
\[ \Rightarrow \exists \langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle \in \langle [x, y] \rangle \Rightarrow x_1 \leq x_2 \land y_1 \leq y_2 \]
The error/interval abstraction

We define the interval and error abstraction as the reduced product of the interval and error abstractions:

\[ \langle I, E \rangle \cong \frac{\gamma}{\alpha} \langle I \times E, \sqsubseteq \rangle \]

where

- \( \gamma(i, e) \) \( \overset{\text{def.}}{=} \gamma_I(i) \cup \gamma_E(e) \)
- \( \alpha(X) \) \( \overset{\text{def.}}{=} \langle \alpha_I(X), \alpha_E(X) \rangle \)
- \( \langle i_1, e_1 \rangle \sqsubseteq \langle i_2, e_2 \rangle \) \( \overset{\text{def.}}{=} i_1 \sqsubseteq_I i_2 \land e_1 \sqsubseteq_E e_2 \)

\[ \langle i, e \rangle \equiv \langle i', e' \rangle \]

Proof.

It follows that \( \equiv \) is equality and so \( \langle i, e \rangle = \{ (i, e) \} \) whence \( (I \times E)/\equiv \) is \( I \times E \) up to the isomorphism \( \{ (i, e) \} \mapsto \langle i, e \rangle \). The definition of \( \alpha \) and of the ordering \( \sqsubseteq \) follows immediately from this remark. \( \square \)

Reference

The interval abstraction as the reduced product of the minimum and maximum abstractions

- We have seen that if \( \langle S, \leq, -\infty, +\infty, \max, \min \rangle \) is a complete lattice, \( \langle \wp(S), \subseteq \rangle \overset{\alpha_M}{\rightarrow} \langle S, \leq \rangle \) where \( \alpha_M(X) = \max X \) and \( \gamma_M(s) = \{ x \in S \mid x \leq s \} \)
- By duality, \( \langle \wp(S), \subseteq \rangle \overset{\gamma_m}{\rightarrow} \langle S, \geq \rangle \) where \( \alpha_m(X) = \min X \) and \( \gamma_m(s) = \{ x \in S \mid x \geq s \} \)
- Let us consider the reduced product of these two abstractions

\[
\langle \wp(S), \subseteq \rangle \overset{\gamma}{\rightarrow} \langle \{ (a, b) \mid a \leq b \} \cup \{ \bot \}, \sqsubseteq \rangle
\]

**Proof.** Trivial. \( \square \)

---

Implementation of the complete lattice of intervals — (1) Abstract properties

(* avvalues.ml *)

open Values
(* abstraction of sets of machine integers by intervals *)
(* complete lattice *)
(* ABSTRACT VALUES *)
(* )

**The classes \([\langle a, b \rangle] \equiv \) where \( a \leq b \) is \{ \( a, b \) \} whence can be represented as \( \langle a, b \rangle \in S \times S \)**

- The classes \([\langle a, b \rangle] \equiv \) where \( a > b \) are \{0\} whence can be represented by some new element \( \bot \not\in S \times S \)
- The reduced product is now, up to an isomorphism:

\[
\langle \wp(S), \subseteq \rangle \overset{\gamma}{\rightarrow} \langle \{ (a, b) \mid a \leq b \} \cup \{ \bot \}, \sqsubseteq \rangle
\]

(* infimum: bot () = alpha({}) *)
let bot () = bottom
(* isbottom a = (a = bot ()) *)
let isbottom (x, y) = y < x
(* isbotempty () = gamma(bot ()) = {} *)
let isbotempty () = false
(* gamma([min_int, max_int]) = *)
let meet (v,w) (x,y) = ((max v x), (min w y))

(* supremum: top () = alpha(_0_i) *)
let top () = (min_int, max_int)
(* least upper bound join: p q = alpha(gamma(p) U gamma(q)) *)
let min x y = if (x <= y) then x else y
let max x y = if (x < y) then y else x
let join (v,w) (x,y) = ((min v x), (max w y))
(* greatest lower bound meet: p q = alpha(gamma(p) cap gamma(q)) *)
let meet (v,w) (x,y) = ((max v x), (min w y))
(* approximation ordering: leq p q = gamma(p) subseteq gamma(q) *)
let leq (v,w) (x,y) = (isbottom (v, w)) || ((x <= v) && (w <= y))
(* equality: eq p q = gamma(p) = gamma(q) *)
let eq (v,w) = (v = w)
(* errors = alpha(\{0,1,0_a\} *)
let errors = bottom
(* included in errors?: in_errors p = gamma(p) subseteq \{0,1,0_a\} *)
let in_errors (x, y) = isbottom (x, y)
(* printing *)
let print_int x =
if x = min_int then print_string "min_int"
else if x = - max_int then print_string "-max_int"
else if x = max_int then print_string "max_int"
else print_int x
let print (x, y) = if (isbottom (x, y)) then print_string "[]" else
(print_string "["; print_int x; print_string ","; print_int y; print_string "]")

Design of the abstract transformers: forward integer constant

f_NAT s = a(\{machine_int_of_strings\})

where:

(* values.ml *)
type error_type = INITIALIZATION | ARITHMETIC
type machine_int = ERROR_NAT of error_type | NAT of int
type machine_bool = ERROR_BOOL of error_type | BOOLEAN of bool
exception Incorrect_Nat of string
let machine_int_of_string s =
    int_of_intstring 0 s

The lexer (lexer.mll) is:

rule token = parse
    [' ' '	' '
' ''] { token lexbuf }
| ( '%' [^'%'])* '%' ) { token lexbuf }
| ['0'-'9']* % { (T_NAT (Lexing.lexeme lexbuf)) }
| '\' { T_LPAR }
...

The parser (parser.mly) is:

let machine_int_of_string s =
    int_of_intstring 0 s

The lexer (lexer.mll) is:

rule token = parse
    [' ' '	' '
' ''] { token lexbuf }
| ( '%' [^'%'])* '%' ) { token lexbuf }
| ['0'-'9']* % { (T_NAT (Lexing.lexeme lexbuf)) }
| '\' { T_LPAR }
...

%token <string> T_NAT
...

...
Proof. By recurrence on $n$:
- if $n = -1$ that is $|s| = 0$ whence $(\text{String.length} s) = 0$, then by symbolic execution, we get $(\text{NAT} i)$ as requested.
- if $n = 0$ then, by symbolic execution
  \[
  \text{int.of.instr.string } i "d_0"
  \]
  \[
  \text{let } v = (10 \otimes i) + (\text{int.of.instr.string } "d_0") \text{ in }
  \{
  v < i ? \text{ERROR_NAT_ARITHMETIC : int.of.instr.string } v ""
  \}
  \]
  \[
  = (10 \otimes i) + d_0 < i ? \text{ERROR_NAT_ARITHMETIC : (NAT} (10 \otimes i) + d_0) \}
  \]
  \[
  \begin{array}{l}
  \{ \text{Notice that } \otimes \text{ and } \oplus \text{ are modulo arithmetic in } [-\text{max.int} \ldotp 1, \text{max.int}] \text{ where max.int} > 9 \text{ and so } ((10 \otimes i) + d_0) < i \iff ((10 \otimes i) + d_0) > \text{max_int since } i, d_0 > 0. \text{ Moreover } (10 \otimes i) + d_0 = (10 \times i) + d_0 \text{ when } (10 \times i) + d_0 \leq \text{max_int}. \text{ Finally } d_0 = "d_0" =
  \end{array}
  \]
  \[
  = ((10 \times i) + d_0) > \text{max_int} ? \text{ERROR_NAT_ARITHMETIC : (NAT} (10^1 \times i) + d_0) \}
  \]
  \[
  \text{Q.E.D.}
  \]

Recall that we have defined (in decimal notation):
\[
\begin{align*}
\mathfrak{s} & \overset{\text{def}}{=} d_n \cdot 10^n + d_{n-1} \cdot 10^{n-1} + \ldots + d_1 \cdot 10^1 + d_0 \cdot 10^0 \\
& = d_n \cdot 10^n + d_{n-1} \cdot 10^{n-1} + \ldots + d_1 \cdot 10 + d_0
\end{align*}
\]

Lemma. If $i$ is a non-negative integer and $s$ a string of digits, "$d_n \ldots d_0" \text{ (which may be empty)}$ then
\[
\begin{align*}
\text{int.of.instr.string } i \mathfrak{s} & = (\text{NAT} i) \quad \text{if } n = -1 \text{ (}|s| = 0) \\
& = (\text{NAT} 10^n + 1 \times i + \mathfrak{s}) \quad \text{if } 10^n + 1 \times i + \mathfrak{s} \leq \text{max.int} \\
& = (\text{ERROR_NAT_ARITHMETIC}) \quad \text{otherwise}
\end{align*}
\]
LEMMA. Let \( s = d_n \ldots d_0 \) where \( n \geq 0 \). Then

\[
\begin{align*}
\text{machine\_int\_of\_string } s &= (\text{NAT } s) \quad \text{if } s \leq \text{max\_int} \\
&= (\text{ERROR\_NAT ARITHMETIC}) \quad \text{otherwise}
\end{align*}
\]

□

PROOF. By symbolic execution:

\[
\begin{align*}
machine\_int\_of\_string s &= \text{int\_of\_intstring } 0 \ s \\
&= (\text{NAT } s) \quad \text{if } s \leq \text{max\_int} \\
&= (\text{ERROR\_NAT ARITHMETIC}) \quad \text{otherwise}
\end{align*}
\]

We now have:

THEOREM.

\[
\begin{align*}
f\_\text{NAT } s &= \begin{cases} [s], s & \text{if } s \leq \text{max\_int} \\
&= \bot & \text{otherwise}
\end{cases}
\end{align*}
\]

□

PROOF. If \( s \leq \text{max\_int} \) then

\[
\begin{align*}
f\_\text{NAT } s &= \begin{cases} [s], s & \text{if } s \leq \text{max\_int} \\
&= \bot & \text{otherwise}
\end{cases}
\end{align*}
\]

The implementation follows (the impossible case (ERROR\_NAT INITIALIZATION) could have been signalled as a design error by the analyzer):

\[
\begin{align*}
\text{let } f\_\text{NAT } s &= \alpha(\{\text{machine\_int\_of\_string } s\}) \\
&= \langle \& \rangle \\
\text{Otherwise } s > \text{max\_int} \text{ and then }
\end{align*}
\]

\[
\begin{align*}
f\_\text{NAT } s &= \alpha(\{\text{ERROR\_NAT ARITHMETIC}\}) \\
&= \bot
\end{align*}
\]

□
Design of the abstract transformers: backward integer constant

- The backward collecting semantics of arithmetic expressions was defined in lecture (17) as:

\[ \text{Baexp}(A)(R)P \overset{\text{def}}{=} \{ \rho \in R \mid \exists i \in P \cap \mathbb{I} : \rho \vdash A \Rightarrow i \} \hspace{1cm} (2) \]

and their backward abstract interpretation was defined as:

\[ \text{Baexp}^\gamma[A] \ni \alpha^\gamma(\text{Baexp}[A]) \hspace{1cm} (3) \]

and we have proved that:

\[ \text{Baexp}^\gamma[A](\lambda Y. \bot)p \overset{\text{def}}{=} \lambda Y. \bot \quad \text{if } \gamma(\bot) = \emptyset \hspace{1cm} (4) \]

\[ \text{Baexp}^\gamma[n]((r)p) \overset{\text{def}}{=} (n_0(p) ? r : \lambda Y. \bot) \hspace{1cm} (5) \]

where:

\[ n_0(p) \overset{\text{def}}{=} (n \in \gamma(p) \cap \mathbb{I}) \hspace{1cm} (6) \]

- Therefore, for the implementation, we define \(^6\)

\[ b\_\text{NAT} s v \overset{\text{def}}{=} (\text{machine\_int\_of\_string } s) \in \gamma(v) \cap [\text{min\_int}, \text{max\_int}] \]

\[ \begin{align*}
\text{let } v \in [a, b] \text{ in } (s > \text{max\_int} \iff a \leq s \leq b) \\
\end{align*} \]

PROOF. Assume that \( v = [a, b] \) where \( b < a \) for bottom. We have:

\[ \begin{align*}
& b\_\text{NAT} s [a, b] \\
& = (\text{machine\_int\_of\_string } s) \in \gamma([a, b]) \cap [\text{min\_int}, \text{max\_int}] \\
& \quad \text{by lemma on machine\_int\_of\_string} \\
& = (\hat{s} \leq \text{max\_int} \iff \text{INT } s) : (\text{ERROR\_NAT ARITHMETIC}) \in \gamma([a, b]) \cap [\text{min\_int}, \text{max\_int}] \\
& = (\hat{s} \leq \text{max\_int} \iff \text{INT } s) : (\text{ERROR\_NAT ARITHMETIC}) \in \{ (a \leq b \iff [a, b] \cap \{\Omega_1, \Omega_2\} \cap [\text{min\_int}, \text{max\_int}] = \{\Omega_1, \Omega_2\} \cap [\text{min\_int}, \text{max\_int}] \}) \\
\end{align*} \]

which directly yields the implementation:

\[ \begin{align*}
\& (* b\_\text{NAT} s v = (\text{machine\_int\_of\_string } s) \in \gamma(v) \cap [\text{min\_int}, \text{max\_int}] \iff *) \\
& \text{let } b\_\text{NAT} s (a, b) = \\
& \text{match (machine\_int\_of\_string } s) \text{ with} \\
& | (\text{ERROR\_NAT INITIALIZATION}) \rightarrow \text{false} \\
& | (\text{ERROR\_NAT ARITHMETIC}) \rightarrow \text{false} \\
& | (\text{NAT } i) \rightarrow (a \leq i) \& (i \leq b) \\
\end{align*} \]

\(^6\) For short, up to a machine representation (NAT i) for \( i \), (ERROR\_NAT INITIALIZATION) for \( \Omega_1 \), and (ERROR\_NAT ARITHMETIC) for \( \Omega_2 \).
Design of the abstract transformers: forward integer addition

- The forward abstract semantics of a binary operator is (from lecture 16):

\[
F_{\text{exp}}[A_1 \circ A_2]_r \overset{\text{def}}{=} b^\circ(F_{\text{exp}}[A_1]_r, F_{\text{exp}}[A_2]_r)
\]

where:

\[
b^\circ(p_1, p_2) \equiv \alpha(\{v_1 \circ v_2 \mid v_1 \in \gamma(p_1) \land v_2 \in \gamma(p_2)\})
\]  

(7)

Theorem. \(\gamma([u, v]) = \{\Omega_1, \Omega_3\}\) if \(v < u\)
\[
\gamma([u, v]) = [u, v] \cup \{\Omega_1, \Omega_3\}\) if \(v \geq u\)

and

let add_int x y =
\[
\begin{align*}
& \text{if } (x > 0) \& \& (y > 0) \text{ then } \\
& \quad \text{if } (x <= (\text{max_int} - y) \text{ then } (\text{NAT}(x+y)) \text{ else } (\text{ERROR}_{\text{NAT ARITHMETIC}})) \\
& \quad \text{else if } (x < 0) \& \& (y <= 0) \text{ then } \\
& \quad \quad \text{if } (\text{min_int} - x) <= y \text{ then } (\text{NAT}(x+y)) \text{ else } (\text{ERROR}_{\text{NAT ARITHMETIC}}) \\
& \quad \quad \text{else } (\text{NAT}(x+y))
\end{align*}
\]

let machine_binary_plus a b = match a with
\[
\begin{align*}
& \text{ERROR}_{\text{NAT } e} \rightarrow (\text{ERROR}_{\text{NAT } e}) \\
& \text{NAT } a' \rightarrow \text{match } b \text{ with } \\
& \quad \text{ERROR}_{\text{NAT } e'} \rightarrow (\text{ERROR}_{\text{NAT } e'}) \\
& \quad \text{NAT } b' \rightarrow (\text{add_int } a' \ b')
\end{align*}
\]

Therefore, up to the computer representation

\[
\begin{align*}
\bot & \rightarrow (\text{max_int, min_int}) \\
\Omega_1 & \rightarrow (\text{ERROR}_{\text{NAT INITIALIZATION}}) \\
\Omega_3 & \rightarrow (\text{ERROR}_{\text{NAT ARITHMETIC}}) \\
[a, b] & \rightarrow (a, b)
\end{align*}
\]

we define

\[
f_{\text{PLUS}} x y \equiv \alpha(\{\text{machine_binary_plus } i \ j \mid i \in \gamma(x) \land j \in \gamma(y)\})
\]

where

Lemma.
add_int x y = \(x + y\) if \(\text{min_int} \leq x + y \leq \text{max_int}\)
\[
= \Omega_3 \quad \text{otherwise}
\]

Proof. By cases
- if \(x \geq 0 \& \& y \geq 0\) then, by symbolic execution, if \(O \leq x + y \leq \text{max_int}\) (or equivalently \(x \leq \text{max_int} - y\), which avoids overflows) then \(\text{add_int } x y = x + y\) else \(x + y > \text{max_int}\) and then \(\text{add_int } x y = \Omega_3\)
- if \(x \leq 0 \& \& y \leq 0\) then, by symbolic execution, if \(\text{min_int} \leq x + y \leq 0\) (or equivalently \(\text{min_int} - x \leq y\), which avoids overflows) then \(\text{add_int } x y = x + y\) else \(x + y < \text{min_int}\) and \(\text{add_int } x y = \Omega_3\)
- Otherwise \(x\) and \(y\) are of opposite signs. Assume \(x \in [\text{max_int} - 1, 0]\) and \(y \in [0, \text{max_int}]\) (the other case being symmetric). We have \(x + y \in [-\text{max_int} - 1, \text{max_int}] = [\text{min_int}, \text{max_int}]\) and \(\text{add_int } x y = x + y\) as required.
Notice that the proof implies the absence of overflows when computing `add_int x y` and so the modulo arithmetic of OCaml can be used in place of the mathematical arithmetic operations.

\[ \text{THEOREM.} \]
\[
f_{\_PLUS} \perp b = \perp \\
f_{\_PLUS} a \perp = \perp \\
f_{\_PLUS} (u, v) (w, x) = \perp \text{ if } u + w > \text{max_int} \\
\text{ } \perp \text{ if } v + x < \text{min_int} \\
= (\text{min_int, max_int}) \text{ if } u + w < \text{min_int} \land v + x > \text{max_int} \\
= (\text{min_int, } u + w) \text{ if } u + w < \text{min_int} \land v + x < \text{max_int} \\
= (u + w, \text{max_int}) \text{ if } u + w \geq \text{min_int} \land v + x > \text{max_int} \\
= (u + w, v + x) \text{ if } u + w \geq \text{min_int} \land v + x \leq \text{max_int} \\
\]

\[ \text{PROOF.} \] For the definition of `f_{\_PLUS} a b`, we proceed by cases

- if `a` is bottom, that is `a = (u, v)` with `v < u` so that `(\text{isbottom } (u, v))` holds, we have

\[ f_{\_PLUS} \perp b \]
\[ = \alpha(\{\text{machine_binary_plus } i j \mid i \in \{\Omega_1, \Omega_2\} \land j \in \gamma(b)\}) \]

- otherwise `a = (u, v) \neq \perp \land b = (w, x) \neq \perp` in which case, we have `u \leq v` and `w \leq x`. We calculate

\[ f_{\_PLUS} (u, v) (w, x) \]
\[ = \alpha(\{\text{machine_binary_plus } i j \mid i \in \gamma((u, v)) \land j \in \gamma((w, x)))\}) \]
\[ = \alpha(\{\text{machine_binary_plus } i j \mid i \in \{i' \mid u \leq i' \leq v\} \land \{\Omega_1, \Omega_2\} \land j \in \{j' \mid w \leq j' \leq x\} \}) \]

We proceed by cases:

- if `u + w > \text{max_int}`, then `\text{min}(\text{min_int}, u + w) = u + w` and `\text{min}(\text{max_int}, v + x) = \text{max_int}` if `v + x > u + w` so in this case

\[ = \alpha(\{\Omega_1, \Omega_2\} \cup \{i + j \mid \text{min_int} \leq i + j \leq \text{max_int} \land u \leq i \land v \leq w \leq j \leq x\}) \]
\[ = \alpha(\{\Omega_1, \Omega_2\} \cup \{i + j \mid \text{max}(\text{min_int}, u + w) \leq i + j \leq \text{min}(\text{max_int}, v + x)\}) \]

- otherwise `\text{min}(\text{min_int}, u + w) = u + w` and `\text{min}(\text{max_int}, v + x) = \text{max_int}` so

\[ = \alpha(\{\Omega_1, \Omega_2\}) \]
\[ = \perp \]

- if `v + x < \text{min_int}`, then `u + w \leq v + x < \text{min_int} < \text{max_int}` so

\[ = \alpha(\{\Omega_1, \Omega_2\}) \]
\[ = \perp \]

- if `v + x \geq \text{min_int}`, then `\text{max}(\text{min_int}, u + w) = \text{min_int}` and `\text{max}(\text{max_int}, v + x) = v + x`, so that in this case

\[ = \alpha(\{\Omega_1, \Omega_2\}) \]
\[ = \perp \]

\[ \square \]
\[
= \alpha(\{\Omega_i, \Omega_3\} \cup \{i + j \mid \min_int \leq i + j \leq v + x < \min_int\})
\]

Observe that all sums are in the \([\min_int, \max_int]\) interval whence produce no overflow and can be computed with OCaml modulo arithmetic.

\[
= \alpha(\{\Omega_i, \Omega_3\})
\]

- Otherwise, we have \(u + w \leq v + x\), \(u + w \leq \max_int\) and \(\min_int \leq v + x\).

There remain four cases:
- if \(u + w < \min_int\) then \(\max(\min_int, u + w) = \min_int\) with two sub-cases:
  - if \(v + x > \max_int\) then \(\min(\max_int, v + x) = \max_int\) so that in that case:
    \[
    = \alpha(\{\Omega_i, \Omega_3\} \cup \{i + j \mid \min_int \leq i + j \leq \max_int\})
    \]
    \[
    = (\min_int, \max_int)
    \]
  - otherwise \(v + x \leq \max_int\) and then \(\min(\max_int, v + x) = v + x\) so that in that case:
    \[
    = \alpha(\{\Omega_i, \Omega_3\} \cup \{i + j \mid \min_int \leq i + j \leq u + w \leq \max_int\})
    \]

The only potential problem are the test \(x + y < \min_int \wedge x + y > \max_int\) which can be easily proved to be equivalent to the following functions which produce no overflow whence can be implemented with modulo arithmetic:

```ocaml
let is_sum_lt_min_int x y = (* x + y < min_int *)
  if (x < 0) && (y < 0) then (x < min_int - y) else false
let is_sum_gt_max_int x y = (* x + y > max_int *)
  if (x > 0) && (y > 0) then (x > max_int - y) else false
```

\[
= (u + w, u + w)
\]

- otherwise \(u + w \geq \min_int\) and so \(\max(\min_int, u + w) = u + w\), with two sub-cases, as above:
  - if \(v + x > \max_int\) then \(\min(\max_int, v + x) = \max_int\) so that in that case:
    \[
    = \alpha(\{\Omega_i, \Omega_3\} \cup \{i + j \mid \min_int \leq u + w \leq i + j \leq \max_int\})
    \]
    \[
    = (u + w, \max_int)
    \]
  - otherwise \(v + x \leq \max_int\) then \(\min(\max_int, v + x) = v + x\) so that in that case:
    \[
    = \alpha(\{\Omega_i, \Omega_3\} \cup \{i + j \mid \min_int \leq u + w \leq i + j \leq v + x \leq \max_int\})
    \]
    \[
    = (u + w, v + x)
    \]

\(\square\)
From the calculational derivation of the definition of \( f_{\text{PLUS}} \) as shown above, we immediately obtain the following implementation, by just considering all possible cases:

\[
(* \ f_{\text{BINARITH}} \ a \ b = \alpha(\{(\text{machine\_binary\_binarith} \ i \ j) \mid i \in \gamma(a) \land j \in \gamma(b)\} \ast) *)
\]

let \( f_{\text{PLUS}} \ (a, b) \ (c, d) = \)
if (isbottom (a, b)) || (isbottom (c, d)) then bottom
else if (is_sum_gt_max_int a c) then bottom
else if (is_sum_lt_min_int b d) then bottom
else let lb = if (is_sum_lt_min_int a c) then min_int else a + c
and ub = if (is_sum_gt_max_int b d) then max_int else b + d
in (lb, ub)

Design of the abstract transformers: backward integer addition

- The generic backward/bottom-up non-relational abstract semantics of arithmetic expressions was shown to be of the form

\[
\begin{align*}
\text{Baexp}^b[A_1 \ b \ A_2](r)p & \overset{\text{def}}{=} \ b^\ast(q_1, q_2, p) \\
\text{let} \ (p_1, p_2) & = b^\ast(\text{Faexp}^b[A_1]r, \text{Faexp}^b[A_2]r, p) \ \\
& \text{in} \ \\
\text{Baexp}^b[A_1](r)p_1 & \cap \text{Baexp}^b[A_2](r)p_2
\end{align*}
\]

where

\[
b^\ast(q_1, q_2, p) \ \\
\alpha^2(\{\langle i_1, i_2 \rangle \in \gamma^2(q_1, q_2) \mid i_1 \ b \ i_2 \in \gamma(p) \cap \mathbb{I}\})
\]

- We consider the case of the binary addition \(+\), up to the encoding

\[
\begin{align*}
\bot & \rightarrow (\text{max\_int, min\_int}) \\
\Omega_1 & \rightarrow (\text{ERROR\_NAT\ INITIALIALIZATION}) \\
\Omega_2 & \rightarrow (\text{ERROR\_NAT\ ARITHMETIC}) \\
[a, b] & \rightarrow (a, b)
\end{align*}
\]

- Recall that we have

\[
\begin{align*}
\gamma(a, b) & \ast \\
\gamma(\{\_0(\_a), \_0(\_i)\} \mid \text{when} \ min\_int \leq a \leq b \leq \text{max\_int} \ast) \\
\gamma(\{\_0(\_a), \_0(\_i)\} \mid \text{when} \ a = \text{max\_int} > \text{min\_int} = b \ast) \\
\text{bottom} & \ast \\
\text{bot} & \ast \\
\text{isbottom} & \ast \\
\text{isbotempty} & \ast \\
\text{errors} & \ast \\
\text{let} & \ast
\end{align*}
\]
We define
\[
\text{b\_PLUS } q_1 \ q_2 \ p \overset{\text{def}}{=} \alpha^2(\{(i_1, i_2) \mid i_1 \in \gamma(q_1) \land i_2 \in \gamma(q_2) \land (\text{machine\_binary\_plus } i_1 \ i_2) \in \gamma(p) \cap \mathbb{I}\})
\]

We have \( q_1 = (a, b), q_2 = (c, d) \) and \( p = (e, f) \) with \( (x, y) = \bot \) (bottom) whenever \( y < x \).

**Theorem.**
\[
b\_PLUS \ q_1 \ q_2 \ p \ = \ \bot \quad \text{if } q_1, q_2 \ \text{or } p \ = \ \bot
\]
\[
b\_PLUS \ (a, b) \ (c, d) \ (e, f) =
\]
\[
\text{let } \ell_1 = \max(a, (e - d < \min \ _\text{int} \ ? \ min \ _\text{int} \ ? \ e - d))\text{in}
\]
\[
\quad \quad \text{and } u_1 = \min(b, (f - c > \max \ _\text{int} \ ? \ max \ _\text{int} \ ? \ f - c))\text{in}
\]
\[
\quad \quad \text{and } \ell_2 = \max(c, (e - b < \min \ _\text{int} \ ? \ min \ _\text{int} \ ? \ e - b))\text{in}
\]
\[
\quad \quad \text{and } u_2 = \min(d, (f - a > \max \ _\text{int} \ ? \ max \ _\text{int} \ ? \ f - a))\text{in}
\]
\[
\quad \quad \text{and } q_1' = (\ell_1 \leq u_1 ? [\ell_1, u_1] \ \bot)\text{in}
\]
\[
\quad \quad \text{and } q_2' = (\ell_2 \leq u_2 ? [\ell_2, u_2] \ \bot)\text{in}(q_1', q_2')
\]

**Proof.** We first consider the cases of bottom arguments:
- If \( q_1 = \bot \), then \( i_1 \in \gamma(q_1) = \{\Omega_1, \Omega_2\} \) so, by definition of (machine\_binary\_plus \ (in values.ml):

\[
\text{let machine\_binary\_plus a b = match a with}
\]
\[
| \text{ERROR\_NAT e } \rightarrow (\text{ERROR\_NAT e})
\]
\[
| \text{NAT a' } \rightarrow \text{match b with}
\]
\[
| \text{ERROR\_NAT e' } \rightarrow (\text{ERROR\_NAT e'})
\]
\[
| \text{NAT b' } \rightarrow (\text{add\_int a' b'})
\]

we have (machine\_binary\_plus i_1 i_2 = i_1 \not\in \mathbb{I} \text{ so } i_1 \not\in \gamma(p) \cap \mathbb{I} ). In that case the result is therefore \( \alpha^2(\emptyset) = (\bot, \bot) \).
- If \( q_2 = \bot \), then the same reasoning yields \( (\bot, \bot) \).
- If \( p = \bot \) then \( \gamma(p) \cap \mathbb{I} = \{\Omega_1, \Omega_2\} \cap \mathbb{I} = \emptyset \) and so, once again, the result is \( \alpha^2(\emptyset) = (\bot, \bot) \).
- In the remaining cases, none of \( q_1 = (a, b), q_2 = (c, d) \) and \( p = (e, f) \) is \( \bot \) so that we can assume \( a \leq b, c \leq d \) and \( e \leq f \). In that case we have:
The test can be implemented using the following function which can easily be shown to be respectively equivalent to \((e - d < \min\_\text{int})\) and \(f - c > \max\_\text{int}\) while avoiding overflows:

```plaintext
let is_difference_lt_min_int x y = (* x - y < min_int *)
  if (x < 0) && (y > 0) then
    (x < min_int + y)
  else false

let is_difference_gt_max_int x y = (* x - y > max_int *)
  if (x > 0) && (y < 0) then
    (x > max_int + y)
  else false
```

#### Design of the abstract transformers: forward integer comparison

- For the generic forward/top-down nonrelational abstract semantics of boolean expressions, we have defined (in lecture 16):

\[
\text{Abexp}[A_1 \land A_2] = \tilde{c}(\text{Faexp}^b[A_1]r, \text{Faexp}^b[A_2])
\]

where

\[
\tilde{c}(p_1, p_2)r \supseteq (\exists v_1 \in \gamma(p_1) : \exists v_2 \in \gamma(p_2) \cap \mathbb{II} : v_1 \triangleleft v_2 = \text{tt} \uparrow \tau : \downarrow)
\]

- Therefore, we define \(f\_\text{LT} p q\) such that

\[
(\exists i \in \gamma(p) \cap \mathbb{II} : \exists j \in \gamma(q) \cap \mathbb{II} : \text{machine}\_\text{lt} i j) \implies (f\_\text{LT} p q)
\]

where (from values.ml)

```plaintext
let machine\_\text{lt} a b = match a with
  | ERROR\_\text{NAT} e -> (ERROR\_\text{BOOL} e)
  | NAT a' -> match b with
    | ERROR\_\text{NAT} e' -> (ERROR\_\text{BOOL} e')
    | NAT b' -> (BOOLEAN (a' < b'))
```

\[f\_\text{LT} q = \perp\]
\[f\_\text{LT} p = \perp\]
\[f\_\text{LT} (x, y) (x', y') = (x < y')\]

### The symbolic execution of \(b\_\text{PLUS} q_1 q_2 p\) yields the expected result as defined above:

\[
\begin{align*}
(*\text{ b\_BOP q1 q2 p} = \alpha\text{alpha2}({\langle i_1,i_2\rangle} \in \gamma\text{gamma2}({q1,q2}) | \cdot)\newline (*\text{ BOP(i1, i2) \in gamma(p) cap [min_int, max_int]} | \cdot)\newline \text{let b\_PLUS} (a, b) (c, d) (e, f) = \newline \quad \text{if } (\text{in\_errors} (a, b)) || (\text{in\_errors} (c, d)) \text{ then errors, errors} \newline \quad \text{else if } (\text{in\_errors} (e, f)) \text{ then bottom, bottom} \newline \quad \text{else let } lq1 = \max\_\text{int} (\text{is\_difference\_lt\_min\_int} e d) \newline \quad \quad \text{then min\_int else } (e - d)) \newline \quad \text{and uq1 = min\_int} (\text{is\_difference\_gt\_max\_int} f c) \\
\quad \text{then max\_int else } (f - c)) \newline \quad \text{and lp2 = max\_int} (\text{is\_difference\_lt\_min\_int} e b) \\
\quad \text{then min\_int else } (e - b)) \newline \quad \text{and up2 = min\_int} (\text{is\_difference\_gt\_max\_int} f a) \\
\quad \text{then max\_int else } (f - a))
\end{align*}
\]
Proof. – Observe that if \( p = \bot \) or \( q = \bot \) then \( \gamma(p) \cap I = \emptyset \) or \( \gamma(q) \cap I = \emptyset \) so that we have \( f_{\text{LT}} p = \bot \) and \( f_{\text{LT}} p = \bot \) – Otherwise we let \( p = (x, y) \) and \( q = (x', y') \) where \( x \leq y \) and \( x' \leq y' \). We have

\[
\exists i \in \gamma((x, y)) \cap I. \exists j \in \gamma((x', y')) \cap I. \text{machine}_{lt} i j
\]

so that we have \( \gamma _{LT} p q = \bot \) and \( \gamma _{LT} p q = \bot \) – Otherwise we let \( p = (x, y) \) and \( q = (x', y') \) where \( x \leq y \) and \( x' \leq y' \). We have

\[
\exists i \in \gamma((x, y)) \cap I. \exists j \in \gamma((x', y')) \cap I. \text{machine}_{lt} i j
\]

so when \( p = (x, y) \neq \bot \) and \( q = (x', y') \neq \bot \), we define \( f_{\text{LT}} (x, y) (x', y') = (x < y') \). \( \square \)

This immediately leads to the following implementation:

\begin{verbatim}
(* Are there integer values in gamma(u) equal to values in gamma(v)? *)
(* f_LT p q = exists i in gamma(p) cap [min_int,max_int]: *)
(* exists j in gamma(q) cap [min_int,max_int]: machine_eq i j *)
let f_EQ (x, y) (x', y') =
  if (isbottom (x, y)) || (isbottom (x', y')) then false
  else (min y y') <= (max x x')
\end{verbatim}

Design of the abstract transformers: forward integer comparison, revisited version

– When considering the improved abstract interpretation of boolean expressions using the backward abstract interpretation of arithmetic subexpressions (course 17), we have defined:

\[
\text{Abexp}^\alpha[A_1 \cup A_2] r \equiv
\text{let } (p_1, p_2) = \tilde{c}(\text{Faexp}^\alpha[A_1] r, \text{Faexp}^\alpha[A_2] r) \text{ in }
\text{Baexp}^\alpha[A_1] (r) p_1 \cap \text{Baexp}^\alpha[A_2] (r) p_2
\]

where

\[
\tilde{c}(p_1, p_2) \equiv^2 \alpha^2(\{<i_1, i_2> | i_1 \in \gamma(p_1) \cap I \land i_2 \in \gamma(p_2) \cap I \land i_1 < i_2 \})
\]

– Up to the machine representation of abstract values, we define:

\[
a_{\text{LT}} p q = \alpha^2(\{<i_1, i_2> | i_1 \in \gamma(p_1) \cap I \land i_2 \in \gamma(p_2) \cap I \land i_1 < i_2 \})
\]
**Theorem.**

\[
a \_ \_ \LT q = (\bot, \bot) \\
a \_ \_ \LT p = (\bot, \bot) \\
a \_ \_ \LT (a, b) \ (c, d) = (\bot, \bot) \text{ if } a \geq d \\
= ([a, \min(b, d - 1)], [\max(a + 1, c), d]) \text{ if } a < d
\]

\[\square\]

**Proof.**  
- If \(p = \bot \lor q = \bot\) then \(\gamma(p) \cap \bot \lor \gamma(q) \cap \bot = \emptyset\) so \(a \_ \_ \_ \LT q = (\bot, \bot)\) and \(a \_ \_ \_ \LT p = (\bot, \bot)\)
- Otherwise \(p = [a, b]\) and \(q = (c, d)\) with \(\min \_ \_ \_ \min(a, b) \leq c \leq d \leq \max \_ \_ \_ \max(a, b)\)
- We have

\[\alpha^2((i_1, i_2) \mid i_1 \in \gamma(a, b)) \cap I \wedge i_2 \in \gamma(c, d) \cap I \wedge i_2 < i_2]\]
\[\alpha^2((i_1, i_2) \mid a \leq i_1 \leq b \wedge c \leq i_2 \leq d \wedge i_2 < i_2)\]

Now, we consider three cases:
- If \(d \leq a\), then we get \(\alpha^2(\emptyset) = (\bot, \bot)\). In this case
  - \((a, \min(b, d - 1)) = (a, a - 1)\) since \(d \leq a \leq b\)
    - \(= \bot\) since \(d - 1 < a\)
  - \((\max((a + 1, c), d) = (a + 1, d)\) since \(c \leq d \leq a + 1\)
    - \(= \bot\) since \(d < a + 1\)
- Otherwise \(a < d\), in which case:
  \[\alpha^2((i_1, i_2) \mid a \leq i_1 \leq b \wedge c \leq i_2 \leq d \wedge i_1 < i_2)\]
  \[= \alpha^2((i_1, i_2) \mid a \leq i_1 \leq b \wedge i_1 < i_2 \leq d \wedge c \leq i_2 \leq d \wedge a < i_2 \wedge i_1 < i_2)\]
  \[\{\alpha^2 \text{ is monotone}\}
\[\subseteq \alpha^2((i_1, i_2) \mid a \leq i_1 \leq \min(b, d - 1) \wedge \max(a + 1, c) \leq i_2 \leq d \wedge i_1 < i_2)\]

\[\square\]

**Implementation of the abstract transformers**

1 (* avalues.ml *)
2 open Values
3 (* abstraction of sets of machine integers by intervals *)
4 (* complete lattice *)
5 ...
6 (*)
7 (* ABSTRACT TRANSFORMERS *)
8 (*)
9 (* forward abstract semantics of arithmetic expressions *)
10 (* f_NAT s = alpha({machine_int_of_string s})\) *)
11 let f_NAT s =
12 match (machine_int_of_string s) with
13 | (ERROR_NAT_INITIALIZATION) -> bottom
14 | (ERROR_NAT_ARITHMETIC) -> bottom
15  \| (\text{NAT} \ i) \rightarrow (i, i) \\
16 (* f\_RANDOM () = \alpha([\text{min\_int}, \text{max\_int}]) *) \\
17 \text{let } f\_RANDOM () = ([\text{min\_int}, \text{max\_int}]) \\
18 (* f\_UMINUS a = \alpha(\{ (\text{machine\_unary\_minus} x) | x \in \gamma(a) \} \} *) \\
19 \text{let } f\_UMINUS (x, y) = \text{if isbottom (x, y) then bottom} \\
20 \text{else if (x = \text{min\_int}) then (-y, \text{max\_int})} \\
21 \text{else (-y, -x)} \\
22 (* f\_UPLUS a = \alpha(\gamma(a)) *) \\
23 \text{let } f\_UPLUS x = x \\
24 (* f\_BINARITH a b = \alpha(\{ (\text{machine\_binary\_binarith} i j) | i \in \gamma(a) /\ j \in \gamma(b) \} \} \} *) \\
25 \text{let is\_sum\_lt\_min\_int x y =} \\
26 (* x + y < \text{min\_int} *) \\
27 \text{if (x < 0) \&\& (y < 0) then} \\
28 (* x + y < \text{min\_int} - y) \\
29 \text{else false} \\
30 \text{let is\_sum\_gt\_max\_int x y =} \\
31 (* x + y > \text{max\_int} *) 

51  \text{let } f\_MINUS (a, b) (c, d) = \\
52 \text{if (isbottom (a, b)) \&\& (isbottom (c, d)) then bottom} \\
53 \text{else if (is\_difference\_gt\_max\_int a d) then bottom} \\
54 \text{else if (is\_difference\_lt\_min\_int b c) then bottom} \\
55 \text{else let lb = (is\_difference\_lt\_min\_int a d) then \text{min\_int} \&\& \text{a - d}} \\
56 \text{and ub = (is\_difference\_gt\_max\_int b c) then \text{max\_int} \&\& \text{b - c}} \\
57 \text{in (lb, ub)} \\
58 \text{let } f\_MINUS (a, b) (c, d) = \\
59 \text{if (isbottom (a, b)) \&\& (isbottom (c, d)) then bottom} \\
60 \text{else if (is\_difference\_gt\_max\_int a d) then \text{bottom}} \\
61 \text{else if (is\_difference\_lt\_min\_int b c) then \text{bottom}} \\
62 \text{else (is\_difference\_lt\_min\_int a d) then \text{min\_int} \&\& \text{a - d}} \\
63 \text{and ub = (is\_difference\_gt\_max\_int b c) then \text{max\_int} \&\& \text{b - c}} \\
64 \text{in (lb, ub)} \\
65 \text{let } sign x = \text{if (x} \geq 0 \text{) then 1 else -1} \\
66 \text{let } \text{abs x} = \text{if (x} \geq 0 \text{) then x} \\
67 \text{else if (x} = \text{min\_int} \text{) then} \\
68 \text{raise (Error\_abs *Incoherence: abs(min\_int)*)} \\
69 \text{else (- x)} \\
70 \text{let times\_int x y =} \\
71 \text{if (x} = 0 \text{) or (y} = 0 \text{) then 0} \\
72 \text{else if x} = \text{min\_int} \text{then} \\
73 \text{if (y} = 1 \text{) then \text{min\_int} \&\& \text{if y} < 0 \text{then max\_int else min\_int \&\& \text{min\_int}}} \\
74 \text{else if (is\_sum\_gt\_max\_int a c) then bottom} \\
75 \text{else if (is\_sum\_lt\_min\_int b d) then bottom} \\
76 \text{else let lb = (is\_sum\_lt\_min\_int a c) then min\_int else a + c} \\
77 \text{and ub = (is\_sum\_gt\_max\_int b d) then max\_int else b + d} \\
78 \text{in (lb, ub)} \\
79 \text{let is\_difference\_lt\_min\_int x y =} \\
80 (* x - y < min\_int *) \\
81 \text{if (x} < 0 \text{) \&\& (y} > 0 \text{) then} \\
82 (* x - y < min\_int + y) \\
83 \text{else false} \\
84 \text{let is\_difference\_gt\_max\_int x y =} \\
85 (* x - y > max\_int *) \\
86 \text{if (x} > 0 \text{) \&\& (y} < 0 \text{) then}
and \( d = \text{times_int \, y \, y'} \) in
\[
\left( (\text{min} \, (\text{min} \, a \, b), \, (\text{min} \, c \, d)), \, (\text{max} \, (\text{max} \, a \, b), \, (\text{max} \, c \, d)) \right)
\]

let rec \( f_{DIV} \) (\( x, y \)) (\( x', y' \)) =
  if (isbottom (\( x, y \))) \lor \ (isbottom (\( x', y' \))) \lor \ (x' = \text{bottom})
  then bottom
  else if \( x' = 0 \) then \( f_{DIV} \) (\( x, y \)) (1, \( y' \))
  else if \( y' = 0 \) then \( f_{DIV} \) (\( x, y \)) (\( x', 1 \))
  else let \( a = \frac{x}{x'} \)
       and \( b = \frac{x}{y'} \)
       and \( c = \frac{y}{x'} \)
       and \( d = \frac{y}{y'} \) in
  \((\text{min} \, (\text{min} \, a \, b), \, (\text{min} \, c \, d)), \, (\text{max} \, (\text{max} \, a \, b), \, (\text{max} \, c \, d))\)\)

let rec \( f_{MOD} \) (\( x, y \)) (\( x', y' \)) =
  if (isbottom (\( x, y \))) \lor \ (isbottom (\( x', y' \))) \lor \ (y < 0)
  then bottom
  else if \( x' < 0 \) then \( f_{MOD} \) (\( x, y \)) (0, \( y' \))
  else if \( y' \leq 0 \) then \( f_{MOD} \) (\( x, y \)) (\( x', 1 \))
  else let \( a = x \bmod x' \)
       and \( b = x \bmod y' \)
       and \( c = y \bmod x' \)
       and \( d = y \bmod y' \) in
  \((\text{min} \, (\text{min} \, a \, b), \, (\text{min} \, c \, d)), \, (\text{max} \, (\text{max} \, a \, b), \, (\text{max} \, c \, d))\)\)

(*) backward abstract semantics of arithmetic expressions
(*)

let \( b_{\text{NAT}} \) s v = (\( \text{machine_int_of_string} \, s \)) \in \( \gamma(v) \) \cap [\text{min_int}, \text{max_int}]\) \(*\)

let \( b_{\text{RANDOM}} \) p = \( \gamma(p) \) \cap [\text{min_int}, \text{max_int}] \neq \emptyset \)
(*\)

let \( b_{-\text{UOP}} \) q p = \( \alpha(\{i \in \gamma(q) | \text{UOP}(i) \in \gamma(p) \cap [\text{min_int}, \text{max_int}]\}) \)
(*\)

let \( b_{\text{PLUS}} \) (\( a, b \)) (\( c, d \)) (\( e, f \)) =
  if (in_errors (\( a, b \))) \lor \ (in_errors (\( c, d \))) then errors, errors
  else if (in_errors (\( e, f \))) then bottom, bottom
  else let \( lq1 = \text{max} \, a \) \( \text{if} \ (\text{is_difference} \, \text{lt} \, \text{min_int} \, e \, d) \)
        and \( uq1 = \text{min} \, b \) \( \text{if} \ (\text{is_difference} \, \text{gt} \, \text{max_int} \, f \, c) \)
  in \( \text{if} \ (lq1 \leq uq1) \) then \( lq1, uq1 \) \else \( \text{bottom} \) \)

let \( b_{\text{MINUS}} \) (\( a, b \)) (\( c, d \)) (\( e, f \)) =
  if (in_errors (\( a, b \))) \lor \ (in_errors (\( c, d \))) then errors, errors
  else if (in_errors (\( e, f \))) then bottom, bottom
  else \( b_{\text{PLUS}} \) (\( a, b \)) (\( -d, -c \)) (\( e, f \))

let \( b_{\text{TIMES}} \) (\( a, b \)) (\( c, d \)) (\( e, f \)) =
  if (in_errors (\( a, b \))) \lor \ (in_errors (\( c, d \))) then errors, errors
  else if (in_errors (\( e, f \))) then bottom, bottom
  else \( b_{\text{PLUS}} \) (\( a, b \)) (\( c, d \)) (\( e, f \))
if (in_errors (a, b)) || (in_errors (c, d)) then errors, errors
else if (in_errors (e, f)) then bottom, bottom
else (a, b), (c, d)

let b_DIV (a, b) (c, d) (e, f) =
if (in_errors (a, b)) || (in_errors (c, d)) then errors, errors
else if (in_errors (e, f)) then bottom, bottom
else (a, b), (c, d)

let b_MOD (a, b) (c, d) (e, f) =
if (in_errors (a, b)) || (in_errors (c, d)) then errors, errors
else if (in_errors (e, f)) then bottom, bottom
else (a, b), (c, d)

(* backward abstract interpretation of boolean expressions *)

let a_EQ p1 p2 = let p = p1 cap p2 cap [min_int, max_int] in <p, p>
let a_LT (a, b) (c, d) = if (isbottom (a, b)) || (isbottom (c, d)) || (a >= d) then
(bottom, bottom) else ((a, min b (d - 1)), (max (a + 1) c, d))

Design of the abstract convergence accelerators

Widening (with thresholds)

The widening is defined with thresholds (including min_int and max_int by default):

let thresholds = let data = [-1; 0; 1; ] in
(Array.sort cmp data; data)
let widen (x, y) (x', y') = if (isbottom (x, y)) then (x', y')
else if (isbottom (x', y')) then (x, y)
else let lastindex = (Array.length thresholds) - 1 in
let a = if x' >= x then x
else let i = ref lastindex in

let i = ref lastindex in
  while (!i >= 0) & (x' < thresholds.(!i)) do
    i := !i - 1
  done;
  if (!i < 0) then min_int else thresholds.(!i))

and b = if y' <= y then y
  else let j = ref 0 in
    while (!j <= lastindex) & (y' > thresholds.(!j)) do
      j := !j + 1
    done;
    if (!j > lastindex) then max_int else thresholds.(!j))

in a, b

---

**Theorem.** widen is a widening operator.

**Proof.** We first prove that \( p \sqsubseteq p \triangledown q \)

- if \( p = \bot \) then \( p = \bot \sqsubseteq q = p \triangledown q \)
- if \( q = \bot \) then \( p \sqsubseteq p = p \triangledown q \)
- Otherwise \( p = (x, y), q = (x', y') \) such that \( x \leq y \) and \( x' \leq y' \). Then \( p \triangledown q = (a, b) \). We must show that \( a \leq \min(x, x') \).
  - if \( x' \geq x \) then \( a = \min(x, x') = x \)
  - otherwise \( x' < x \) in which case we must prove that \( a \leq x' = \min(x, x') \).

We consider two cases

- if thresholds is \([i]\) is empty then \( \texttt{Array.length thresholds}) = 0 \) so lastindex = -1 whence \( i = -1 < 0 \) and \( a = \min \_ \text{int} \) which satisfies \( a \leq x' \)
- Otherwise thresholds is \([t_0, \ldots, t_n]\) with \( n \geq 0 \) is not empty. So lastindex = \( \texttt{Array.length thesholds}) - 1 = n \). Then the following loop is executed.

\[
\begin{align*}
y^0 &= x^0 \\
y^{i+1} &= y^i & \text{if } x^i \sqsubseteq y^i \\
\end{align*}
\]

is not strictly increasing. The proof is by reductio ad absurdum.

Assume that \( y^0 \sqsubseteq y^1 \sqsubseteq \ldots \sqsubseteq y^i \sqsubseteq \ldots \) then (b) is never used. It follows that \( \forall \delta \geq 0 : y^{i+1} = y^i \triangledown x^\delta \). The only \( \sqcup \) element can be \( y^i \), which can be eliminated by considering \( x^i, x^{i+1}, \ldots \) and \( y_i, y_{i+1}, \ldots \) with \( y_i \neq \bot \) and without changing the final result. Moreover the \( x^i \) cannot be \( \bot \) since in that case \( y^{i+1} = y^i \forall \delta = y^i \) in contradiction with \( y^i \sqsubseteq y^{i+1} \). Therefore we have \( x^i = (a^i, b^i), \delta \geq 1 \) with \( \min \_ \text{int} \leq a^i \leq b^i \leq \max \_ \text{int} \) such that the sequence \( y^i = (c^i, d^i), \delta \geq 1 \) with \( \min \_ \text{int} \leq c^i \leq d^i \leq \max \_ \text{int} \) is strictly increasing, with

\[
(c^{i+1}, d^{i+1}) = (c^i, d^i) \triangledown (a^i, b^i), \delta \geq 1
\]

Because

\[
(c^i, d^i) \sqsubseteq (c^{i+1}, d^{i+1})
\]

we have
\[ c^{i+1} < c^i \lor (d^i < d^i + 1) \]

by definition of \[ \sqsubseteq \]
- In case \[ c^{i+1} < c^i \], we have by definition of \[ \triangledown \] that

\[
\begin{align*}
    c^{i+1} &= a^i & \text{if } a^i \geq c^i \\
    &= t_i, 0 \leq i \leq n & \text{when thresholds } = \{ t_0; \ldots; t_n \}
\end{align*}
\]

Observe that the first case is indeed impossible since \[ c^{i+1} < c^i \] implies
- \[ (c^{i+1} = a^i) \geq c^i \] so \[ c^{i+1} = t_i \]
- In case \[ d^i < d^{i+1} \], a similar reasoning shows that \[ d^{i+1} = t'_j, j \in [0, n] \]
- So we have a decreasing chain of elements of “thresholds” for \( c^i, \delta \geq 2 \) and an increasing chain of elements of “thresholds” for \( d^i, \delta \geq 2 \), one of them strictly increasing for each \( \delta \), which is impossible since “thresholds” is finite, which provides the desired contradiction.

\[ \square \]

---

**Narrowing**

- The narrowing is defined as follows:

\[
\begin{align*}
    q^0 &= p^0 \\
    q^{i+1} &= q^i \bigtriangleup p^i & \text{if } p^i \sqsubseteq q^i \\
    &= q^i & \text{otherwise}
\end{align*}
\]

is not strictly increasing.

The proof is by reductio ad absurdum. Assume that \( q^i, \delta \geq 0 \) is strictly decreasing. The case (c) can never be chosen since we would have the contradiction that \( q^{i+1} = q^i \) for some \( \delta > 0 \). So the sequence \( (q^i, \delta \geq 0) \) is defined using (a) and (b) only that is (b) only for \( (q^i, \delta \geq 1) \). Let \( q^i = (a^i, b^i) \) and \( p^i = (c^i, d^i) \) for all \( \delta \geq 1 \). We have:

\[
\begin{align*}
    (c^i, d^i) &\sqsubseteq (a^i, b^i) \\
    \text{and} &\quad (a^{i+1}, b^{i+1}) = (a^i, b^i) \bigtriangleup (c^i, d^i) \\
    \text{and} &\quad (a^{i+1}, b^{i+1}) \sqsubseteq (a^i, b^i)
\end{align*}
\]

- otherwise, \( p = (x, y), q = (x', y') \) with \( \min_{\text{int}} \leq x \leq y \leq \max_{\text{int}} \) and \( \min_{\text{int}} \leq x' \leq y' \leq \max_{\text{int}} \) and \( (x', y') \sqsubseteq (x, y) \). By cases:

  - if \( x = \min_{\text{int}} \) then
    - if \( y = \max_{\text{int}} \) then, by symbolic execution, \( (x', y') \sqsubseteq (x', y') = (x, y) \bigtriangleup (x', y') \sqsubseteq (x, y) \)
    - else \( (x', y') \sqsubseteq (x, y) = (x, y) \bigtriangleup (x', y') \sqsubseteq (x, y) \) since \( x' \leq x \) by \( (x', y') \sqsubseteq (x, y) \)
  - otherwise \( x \neq \min_{\text{int}} \), and then
    - if \( y = \max_{\text{int}} \) then, by symbolic execution, \( (x', y') \sqsubseteq (x', \max_{\text{int}}) = (x, y) \bigtriangleup (x', y') \sqsubseteq (x, y) \) since \( x' \leq y \) by \( (x', y') \sqsubseteq (x, y) \) and \( y = \max_{\text{int}} \)
    - otherwise \( y \neq \max_{\text{int}} \) and then \( (x', y') = (x, y) \bigtriangleup (x', y') \sqsubseteq (x, y) \)
  - We must also show that for all sequences \( p^0, p^1, \ldots \) the sequence defined by

\[ y = \max_{\text{int}} \]

\[ \text{if } (x = \min_{\text{int}} \text{ then } x' \text{ else } x), \]

\[ \text{if } (y = \max_{\text{int}} \text{ then } y' \text{ else } y) \]
After $\delta \geq 1$, all elements of $(q^\delta, \delta \geq 1)$ are not $\bot$. We must have $a^{\delta+1} < a^\delta$ (or $b^{\delta+1} > b^\delta$ which is handled in the same way). By definition of $\Delta$, if $a^\delta = \text{min\_int}$ then $a^{\delta+1} = c^\delta \leq a^\delta$ else $a^{\delta+1} = a^\delta$ which is impossible. So $a^\delta = \text{min\_int}$ at the next step $a^{\delta+2} < a^{\delta+1} = c^\delta \leq a^\delta = \text{min\_int}$ which is impossible. This yields the contradiction proving that $\Delta$ enforces convergence.

\[ \square \]

Making the non-relational forward analyzer generic

- The global structure of the analyzer is the same whichever is the abstract domain chosen to approximate sets of values;
- Up to the use of widening/narrowing when no convergence acceleration is needed (e.g. finite domains, domains satisfying the ACC with rapid convergence)
- For non-relational analyzes, the structure of the abstract domain approximating sets of environment only depends on the abstract domain for sets of values

- The algebraic structure can be represented by the modular structure of OCaml programs (thanks to file aliases in this first implementation or better thanks to module functors)
- It is then easy to modify the static analyzer to perform experimentations on the abstract domains:
  → by changing the abstract domain of values
  → by changing the abstract interpretation of arithmetic/boolean expressions or commands
  → without having to change the global structure of the analyzer
Principle of a generic equational static analyzer/verifier

A first implementation using the modular structure of OCAML and aliases of module files

```
Generic-FW-Abstract-Interpreter % make
Forward non-relational static analysis:
make help : this help
1) reset:
make reset : erase all mode choices
2) choose tracing mode:
make trace  : tracing all
make traceaexp : tracing arithmetic expressions
make tracebexp : tracing boolean expressions
make tracecom : tracing commands
make tracered : tracing ternary reductions
make notrace : no tracing
3) choose abstract interpreter mode:
3a) relational/non-relational analysis:
make r  : relational abstract interpreter (not implemented)
make nr : non-relational abstract interpreter
3b) boolean expressions:
make fbool : forward analysis
make fbbool : forward/backward analysis
make fbrbool : forward/backward reductive analysis
3c) arithmetic expressions:
make fassign : forward analysis
make fbassign : forward/backward analysis
4) choose static analysis and compile analyzer:
make err : error analysis
make iss : initialization and simple sign analysis
make int : interval analysis
```

Principle of a generic structural static analyzer/verifier
5) analyze:
   ./a.out : analyze (the standard input)
   ./a.out file.sil : analyze (the file "file.sil")
make examples : analyze all examples
6) clean:
make clean : remove auxiliary files

---

File structure of the generic forward static analyzer

|-- Examples
| |-- example00.sil
| | ... |
| | example73.sil
| | `-- makefile
|-- Generic-FW-Abstract-Interpreter
| |-- Generic-FW-Abstract-Interpreter.tgz
| | |-- aaexp.ml
| | |-- aaexp.mli
| | |-- abexp.ml -> abexp_fbr.ml
| | | | `-- abexp.mli
| | | | `-- abexp_fb.ml
| | | `-- abexp_fbr.ml
| | |-- aaexp.ml
| | |-- aaexp.mli
| | |-- abexp.ml
| | |-- abexp_f.ml
| | | | `-- abexp_ml
| | | | `-- abexp_fbr.ml
| | | `-- abexp_fbr.ml
| | | `-- abexp_fbr.ml
| | | `-- abexp_fbr.ml
| | `-- abexp_fbr.ml
| `-- abexp_fbr.ml

|-- lexer.ml
| |-- main.ml
| |-- makefile
| |-- parser.mly
| |-- pretty_Print.ml
| |-- pretty_Print.mli
| |-- program_To_Abstract_Syntax.ml
| |-- program_To_Abstract_Syntax.mli
| | |-- red12.ml
| | |-- red12.mli
| | |-- red123.ml
| | |-- red123.mli
| | |-- red13.ml
| | |-- red13.mli
| | |-- red23.ml
| | |-- red23.mli
| | |-- symbol_Table.ml
| | |-- symbol_Table.mli
| | |-- trace.ml
| | |-- trace.mli
| `-- trace.mli

---
Creating a specific instance of the generic analyzer

The creation of a specific instance of the analyzer consists in creating aliases of the specific instanciated files before recompiling.

```
% pwd
.../Generic-FW-Abstract-Interpreter
% make reset
Remove instanciated files
% make notrace
Tracing mode off
% make nr
"Non-relational" static analysis
% make fbrbool
Forward/backward analysis of boolean expressions with reduction
```

All files are shared by all instances, but:
- `aenv.ml` (and `avalues.ml` common to all non-relational abstractions)
- `avalues.ml` implementing each specific non-relational abstract domain (errors, intervals, ...)

```
% make fbassign
Forward/backward analysis of assignments
% make int
ocamlyacc parser.mly
ocamlex lexer.ml
62 states, 3001 transitions, table size 12376 bytes
```

"Interval" static analysis
For example for interval analysis, the aliases will be created as follows:

```%
  % tree
  ...
  |– abexp.ml -> abexp_fbr.ml
  ...
  |– acom.ml -> acom_fba.ml
  ...
  |– aenv.ml -> ../Non-Relational/aenv.ml
  ...
  |– avalues.ml -> ../Non-Relational/03-Intervals/avaleus.ml
  ...
  |– avalues.mli -> ../Non-Relational/avaleus.mli
  ...
```

Generating an instance of the generic forward static analyzer

```bash
1  # makefile
2  3  SHELL = tcsh
4  5  EXAMPLES = ../Examples
6  7  SOURCES_SINGLE_DOMAIN = \
8      trace.ml \
9      trace.mli \
10     symbol_Table.ml \
11     symbol_Table.mli \
12     variables.ml \
13     variables.mli \
14     abstract_Syntax.ml \
15     abstract_Syntax.mli \
16     concrete_To_Abstract_Syntax.ml \
17     labels.mli \
18     labels.ml \
19     parser.mli \
20     parser.ml \
21     lexer.ml \
22     program_To_Abstract_Syntax.mli \
23     program_To_Abstract_Syntax.ml \
24     pretty_Print.mli \
25     pretty_Print.ml \
26     values.mli \
27     values.ml \
28     avaleus.ml \
29     avaleus.mli \
30     aenv.mli \
31     aenv.ml \
32     aexp.mli \
33     aexp.ml \
34     basp.ml \
35     basp.mli \
36     fixpoint.mli \
37     fixpoint.ml \
38     abexp.mli \
39     abexp.ml \
40     acom.mli \
41     acom.ml \
42     main.ml \
43  44  SOURCES_BINARY_REDUCED_PRODUCT = \
45      trace.ml \
46      trace.ml \
47      symbol_Table.mli \
48      symbol_Table.ml \
49      variables.mli \
50      variables.ml \
51      abstract_Syntax.ml \
52      concrete_To_Abstract_Syntax.mli \
53      concrete_To_Abstract_Syntax.ml \
54      labels.mli \
55      labels.ml \
56
```
.PHONY : help
help :
  @echo ""
  @echo "Forward non-relational static analysis:"
  @echo "make help : this help"
  @echo "(1) reset:
  @echo "make reset : erase all mode choices"
  @echo "(2) choose tracing mode:
  @echo "make trace : tracing all"
  @echo "make traceaexp : tracing arithmetic expressions"
  @echo "make tracebexp : tracing boolean expressions"
  @echo "make tracecom : tracing commands"
  @echo "make traced : tracing ternary reductions"
  @echo "make notrace : no tracing"
  @echo "(3) choose abstract interpreter mode:
  @echo "(3a) relational/non-relational analysis:
  @echo "make r : relational abstract interpretor"
  @echo "make nr : non-relational abstract interpretor"
  @echo "(3b) boolean expressions:
  @echo "make fbool : forward analysis"
  @echo "make fboolb : forward/backward analysis"
  @echo "make fboolrb : forward/backward reductive analysis"
  @echo "(3c) arithmetic expressions:
  @echo "make famsign : forward analysis"
  @echo "make fbasisign : forward/backward analysis"
  @echo "(4) choose static analysis and compile analyzer:
  @echo "make err : error analysis"
  @echo "make ins : initialization and simple sign analysis"
  @echo "make int : interval analysis"
  @echo "make par : parity analysis"
  @echo "make err-int : error x interval analysis"
  @echo "make ins-int : initialization and simple sign x interval analysis"
  @echo "make par-int : parity x interval analysis"
  @echo "make par-iss-int : parity x initialization and simple sign analysis x"
  @echo "(5) analyze:
  @echo "make ./a.out : analyze (the standard input)"
  @echo "make examples : analyze all examples"
  @echo "make clean : remove auxiliary files"
  @echo ""
216  fbbool :
217     @/bin/rm -f abexp.ml || true
218     @ln -s abexp_f.ml abexp.ml
219     @echo "Forward analysis of boolean expressions"
220 .PHONY : fbbool
221 fbbool :
222     @/bin/rm -f abexp.ml || true
223     @ln -s abexp_fb.ml abexp.ml
224     @echo "Forward/backward analysis of boolean expressions"
225 .PHONY : fbrbool
226 .PHONY : fassign
227 fassign :
228     @/bin/rm -f acom.ml || true
229     @ln -s acom_fa.ml acom.ml
230     @echo "Forward analysis of assignments"
231 .PHONY : fbassign
232 .PHONY : r
233 r :
234     @echo "Relational" static analysis not implemented'
235 .PHONY : nr
236 nr :
237     @/bin/rm -f aenv.ml || true
238     @ln -s aenv_f.ml aenv.ml
239     @echo "Non-relational" static analysis'
240 .PHONY : err
241 err :
242     ocamlyacc parser.mly
243     ocamllex lexer.mll
244     @echo "Error" static analysis'
245 .PHONY : iss
246 iss :
247     ocamlyacc parser.mly
248     ocamllex lexer.mll
249     @/bin/rm -f avalues.ml || true
250     @ln -s ../Non-Relational/01-Initialization-Simple-Sign/avalues.ml avalues.ml
251     @echo "Initialization and simple sign" static analysis'
252 .PHONY : int
253 int :
254     ocamlyacc parser.mly
255     ocamllex lexer.mll
256     @/bin/rm -f avalues.ml || true
257     @ln -s ../Non-Relational/03-Intervals/avalues.ml avalues.ml
258     @echo "Interval" static analysis'
259 .PHONY : par
260 par :
261     ocamlyacc parser.mly
262     ocamllex lexer.mll
263     @/bin/rm -f avalues.ml || true
264     @ln -s ../Non-Relational/04-Parity/avalues.ml avalues.ml
265     @echo "Parity" static analysis'
296 .PHONY : err-int
297 err-int :
298  ocamlyacc parser.mly
299  ocamllex lexer.mll
300  @/bin/rm -f avalues1.ml || true
301  @/bin/rm -f avalues.ml avalues1.ml
302  @/bin/rm -f avalues.ml || true
303  @/bin/rm -f avalues.ml avalues2.ml
304  @/bin/rm -f avalues2.ml || true
305  @/bin/rm -f avalues2.ml avalues.ml
306  @/bin/rm -f red12.ml || true
307  @/bin/rm -f red12.ml red12.ml
308  # ocamli -i $(SOURCES_BINARY_REDUCED_PRODUCT) # to print types
309  ocamli $(SOURCES_BINARY_REDUCED_PRODUCT)

Course 16.399: "Abstract interpretation", Tuesday May 5th, 2005 — 269 — @ P. Cousot, 2005

336 @echo 'Reduced "initialization and simple sign" and "interval" static analysis'
337 .PHONY : par-int
338 par-int :
339  ocamlyacc parser.mly
340  ocamllex lexer.mll
341  @/bin/rm -f avalues1.ml || true
342  @/bin/rm -f avalues.ml avalues1.ml
343  @/bin/rm -f avalues.ml avalues2.ml
344  @/bin/rm -f avalues.ml || true
345  @/bin/rm -f avalues.ml avalues2.ml
346  @/bin/rm -f avalues2.ml || true
347  @/bin/rm -f avalues2.ml avalues.ml
348  @/bin/rm -f red12.ml || true
349  @/bin/rm -f red12.ml red12.ml
350  @/bin/rm -f red12.ml avalues.ml
351  @/bin/rm -f red12.ml avalues2.ml
352  # ocamli -i $(SOURCES_BINARY_REDUCED_PRODUCT) # to print types
353  ocamli $(SOURCES_BINARY_REDUCED_PRODUCT)
@echo 'Reduced "parity" and "Initialization and simple sign" static analysis'

.PHONY : par-iss-int
par-iss-int :
ocamlyacc parser.mly
ocamllex lexer.mll
@/bin/rm -f avalues1.mli || true
@ln -s avalues.ml avalues1.mli
@/bin/rm -f avalues1.ml || true
@ln -s ../Non-Relational/04-Parity/avalues.ml avalues1.ml
@/bin/rm -f avalues2.mli || true
@ln -s avalues.ml avalues2.mli
@/bin/rm -f avalues2.ml || true
@ln -s ../Non-Relational/01-Initialization-Simple-Sign/avalues.ml avalues2.ml
@/bin/rm -f avalues3.mli || true
@ln -s avalues.ml avalues3.mli
@/bin/rm -f avalues3.ml || true
@ln -s ../Non-Relational/03-Intervals/avalues.ml avalues3.ml
@/bin/rm -f red12.ml || true
@ln -s ../Non-Relational/05-Prod-Red/red-Parity-ISS12.ml red12.ml
@/bin/rm -f red23.ml || true
@ln -s ../Non-Relational/05-Prod-Red/red-ISS-Intervals23.ml red23.ml
@/bin/rm -f red13.ml || true
@ln -s ../Non-Relational/05-Prod-Red/red-Parity-Intervals13.ml red13.ml
@/bin/rm -f avalues.ml || true
@ln -s ../Non-Relational/05-Prod-Red/avalues123.ml avalues.ml
# ocamlc -i ${SOURCES_TERNARY_REDUCED_PRODUCT} # to print types
ocamlc ${SOURCES_TERNARY_REDUCED_PRODUCT}
@echo 'Reduced "parity", "initialization and simple sign" and "interval" static analysis'

.include ${EXAMPLES}/makefile

.PHONY : clean
clean :
-@/bin/rm -f *.cmi *.cmo *~ a.out lexer.ml parser.ml parser.mli
@/bin/rm -f red23.ml || true
@ln -s ../Non-Relational/04-Parity/red-Parity-ISS23.ml red23.ml
@/bin/rm -f red13.ml || true
@ln -s ../Non-Relational/05-Prod-Red/red-Parity-Intervals13.ml red13.ml
@/bin/rm -f avalues.ml || true
@ln -s ../Non-Relational/05-Prod-Red/avalues123.ml avalues.ml
# ocamlc -i $(SOURCES_TERNARY_REDUCED_PRODUCT) # to print types
ocamlc $(SOURCES_TERNARY_REDUCED_PRODUCT)
@echo 'Reduced "parity", "initialization and simple sign" and "interval" static analysis'

Examples of instantiation of the generic forward static analyzer

% pwd
./Generic-FW-Abstract-Interpreter
% make reset
Remove instanciated files
% make notrace
Tracing mode off
% make nr
"Non-relational" static analysis
% make fbbool
Forward/backward analysis of boolean expressions with reduction
% make fbassign
Forward/backward analysis of assignments

@echo "Remove instanciated files"
% make err
"Error" static analysis
% ./a.out ../Examples/example25.sil
{x:{0_i}; y:{0_i}; z:{0_i}}
0:
  x := 0;
  y := ?;
  if ((x + y) = 0) then
    z := (x + y)
  else {(((x + y) < 0) | (0 < (x + y)))}
    z := 0
  fi
7:
{x:{ }; y:{ }; z:{0_a} }

% make int
"Interval" static analysis
% ./a.out ../Examples/example25.sil
{x:{}; y:{}; z:{}}
0:
  x := 0;
  y := ?;
  if ((x + y) = 0) then
    z := (x + y)
  else {(((x + y) < 0) | (0 < (x + y)))}
    z := 0
  fi
7:
{x:[0,0]; y:[min_int,max_int]; z:[0,0] }
% make clean

\[\text{\LARGE Bibliography}\]


\[\text{\LARGE THE END}\]

My MIT website is \url{http://www.mit.edu/~cousot/}
The course website is \url{http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/}.