Introduction – Sémantique concrète

TAS : Typage et analyse statique
M2, Master STL, UPMC

Antoine Miné

Année 2016–2017

Cours 8
1er décembre 2016

Objectifs du cours : 2ème partie

La **vérification** des programmes

par **analyse statique sûre**

basée sur la théorie de l'**interprétation abstraite**.

- Maîtriser les principes et l'état de l'art.
  (à quoi cela peut servir)

- Comprendre les justifications théoriques.
  (pourquoi est-ce que cela marche, quelles règles à respecter)

  Quelques théorèmes et quelques preuves,
  mais sans entrer dans les aspects théoriques les plus techniques.

- Savoir programmer une analyse statique sûre (simple).
  (comment cela marche en pratique)

  Implantation en TME et en projet.

Organisation générale de la 2ème partie

Plan :

- **Séance 8**, 2h de cours + 2h de TME
  Introduction, sémantique concrète
- **Séance 9**, 3h de cours + 3h de TME
  Abstractions de sémantiques
- **Séance 10**, 2h de cours + 2h de TME
  Analyse de boucles, produits réduits
- **Séance 11**, 3h de cours + 3h de TME
  Interrogation
  Domaines disjonctifs, domaines relationnels
- **Séance 12**, 2h de cours + 2h de TME
  Analyse de pointeurs, analyse de tableaux, analyse inter-procédurale
- **Séance 13**, 2h de cours + 2h de TME
  Application : vérification de logiciels embarqués avioniques

Projet d'analyseur, TME

But : écrire un petit analyseur statique

- pour un langage jouet
  variables entières, affectations, tests if-then-else, boucles
- en OCaml
- analyse des valeurs numériques des variables : analyse de constantes, d'intervalles, produit réduit intervalles / parité ;

  plus une extension au choix (analyse relationnelle, entiers machine, etc.)

Fournis :

- sources du *front-end* (analyse syntaxique, AST)
- squelette de l'analyseur et interfaces
- banque de tests minimale

À rendre :

- sources compilables de l'analyseur (OCaml, `Makefile`)
- `README` (liste de fonctionnalités, guide d'utilisation)
- banque de tests enrichie et fichiers de résultats

Organisation des TME

Séances de TME : développement du projet d'analyseur

- sujet de projet donné dès le 1er TME
- séance de TME = l'occasion de mettre en pratique le cours pour avancer le projet
- développement à votre rythme

  exemple de programme :

  - Séance 8 : prise en main, sémantique concrète
  - Séance 9 : domaines des constantes, des intervalles
  - Séance 10 : analyse des boucles
  - Séance 11 : produit réduit
  - Séance 12 : début de développement d'une extension avancée
  - Séance 13 : finalisation du projet

Évaluation

2ème partie du cours TAS $=$ 50% de la note de l'UE

dont :

- Projet : 20%
- Présentation d'un article : 20%
- Interrogation écrite courte ($\simeq$ 30 mn) en début de séance 11 : 10%
  sauf les parties hors programme, notées d'un $_\star{}^\star{}_\star$

Ressources :

les transparents sont mis sur le site du Master au fur et à mesure :
https://www-master.ufr-info-p6.jussieu.fr/2016/TAS

Principes de l'analyse statique

Analyse statique

source

```
int search(int* t, int n) {
  int i;
  for (i=0; i<n; i++) {
    if (t[i]) break;
  }
  return t[i];
}
```

$\Longrightarrow$

résultat de l'analyse

```
int search(int* t, int n) {
  int i;
  for (i=0; i<n; i++) {
    // 0 ≤ i < n
    if (t[i]) break;                ✓
  }
  // 0 ≤ i ≤ n ∨ n < 0
  return t[i];                      ✗
}
```

- analyse le code source original (pas de modèle, pas d'instrumentation)
- infère des propriétés pour déduire la correction des programmes
  au delà du typage : propriétés sensibles au flot de contrôle, sur les valeurs
  application à la preuve d'absence de RTE (*overflows*, tableaux, pointeurs, etc.)
- statiquement (à la compilation)
- automatiquement (sans interaction avec l'utilisateur)
- de manière efficace

$\Longrightarrow$ particulièrement adapté à une utilisation industrielle

L'inférence des propriétés les plus précises est un problème indécidable !

$\Longrightarrow$ nous utilisons des approximations sûres.

Sémantique

Principe : définir un sens mathématique à un programme.

Applications :

- comprendre ce qu'est, fondamentalement, un programme
  ce qu'est un calcul ;
- raisonner sur les programmes ;
  (en raisonnant sur leur interprétation mathématique)
- justifier la validité des méthodes de vérification.

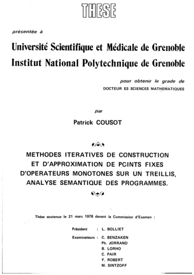
Nombreuses variantes : sémantiques opérationnelles, axiomatiques, dénotationnelles, etc.

Validité d'une analyse statique :

- en partant d'une sémantique concrète d'intérêt d'un langage ;
  base de confiance décrivant très précisément le comportement des programmes
  nous utilisons un modèle opérationnel
- par abstractions successives des opérations du langage
  nous oublions les informations *a priori* inutiles à la preuve
  jusqu'à obtenir une sémantiques calculable

Interprétation abstraite

Patrick Cousot [1]



THESE

présentée à

Université Scientifique et Médicale de Grenoble

Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR ES SCIENCES MATHEMATIQUES

par

Patrick COUSOT

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

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

Président : L. BOLLIET

Examinateurs : C. BENZAKEN, Ph. JORRAND, B. LORHO, C. PAIR, F. ROBERT, M. SINTZOFF

Théorie générale de l'approximation
et de la comparaison des sémantiques :

- permet d'unifier des sémantiques disparates ;
- guide la conception de nouvelles analyses statiques qui sont sûres par construction ;
  $\Longrightarrow$ nous l'utiliserons pour concevoir nos analyses.

1. P. Cousot. "Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes." Thèse És Sciences Mathématiques, 1978.

Langage numérique

Objectifs

Proposer un langage simple :

- dont la sémantique est facile à écrire ;
- qui va nous servir à illustrer l'analyse de propriétés numériques ;
- proche du langage cible du projet d'analyseur.

Nous présentons dans ce cours :

- la syntaxe du langage ;
- une sémantique concrète d'états, non calculable ;
- quelques outils mathématiques et théorèmes
  utiles pour justifier que la sémantique est bien fondée ;
- le code calculant la sémantique concrète
  qui sert de base au projet d'analyseur
  commencé aujourd'hui en TME.

Prochains cours : dérivation de sémantiques approchées calculables

Syntaxe des instructions

Instructions

$$
\begin{array}{rcll}
\textcolor{blue}{\mathit{stat}} & ::= & X \leftarrow \mathit{expr} & \text{(affectation, } X \in \mathbb{V}\text{)} \\
& | & \mathit{stat}; \mathit{stat} & \text{(séquence)} \\
& | & \textbf{if } \mathit{cond} \textbf{ then } \mathit{stat} \textbf{ else } \mathit{stat} & \text{(test)} \\
& | & \textbf{while } \mathit{cond} \textbf{ do } \mathit{stat} & \text{(boucle)} \\
& | & \textbf{skip} & \text{(ne rien faire)} \\
& | & \textbf{assert } \mathit{cond} & \text{(vérification de condition)} \\
& | & \textbf{halt} & \text{(arrêt du programme)}
\end{array}
$$

- présentation sous forme de grammaire BNF
- l'ensemble des variables $\mathbb{V}$ est fini et fixé (pas d'allocation dynamique)
  (dans le projet, les blocs déclareront des variables locales)
- les expressions *expr* des affectations sont arithmétiques,
  les conditions *cond* des tests, boucles, assertions sont booléennes
  (voir transparents suivants)
- **skip** est utile pour programmer le **if** sans **else**
- **assert** permet de spécifier les propriétés à prouver

Syntaxe des expressions arithmétiques

expressions arithmétiques

expr	$::=$	X	(variable, $X \in \mathbb{V}$)
	$\mid$	c	(constante, $c \in \mathbb{Z}$)
	$\mid$	$\diamond\ expr$	(opéartion unaire)
	$\mid$	$expr \diamond expr$	(opération binaire)
	$\mid$	$\textbf{rand}(c_1, c_2)$	(valeur aléatoire entre deux constantes)

- les constantes et variables sont à valeur dans $\mathbb{Z}$
  (les entiers mathématiques illimités, pas les entiers machine !)
- $\diamond$ dénote les opérations arithmétiques classiques :
  - opération unaire : $-$
  - opérations binaires : $+$, $-$, $\times$, $/$
- $\textbf{rand}(c_1, c_2)$ modélise un choix non-déterministe
  entre deux valeurs constantes : $c_1 \in \mathbb{Z} \cup \{-\infty\}$, $c_2 \in \mathbb{Z} \cup \{+\infty\}$

Syntaxe des expressions booléennes

expressions booléennes (conditions)

$\color{blue}{cond}$	$::=$	$expr \bowtie expr$	(comparaison)
	$\mid$	c	(constantes logiques, $c \in \mathbb{B}$)
	$\mid$	$\neg cond$	(négation logique)
	$\mid$	$cond \wedge cond$	(et logique)
	$\mid$	$cond \vee cond$	(ou logique)

- comparaison d'expressions entières : $\bowtie \in \{\leq, \geq, =, \neq, <, >\}$
- constantes logiques : $c \in \mathbb{B}$ où $\mathbb{B} \stackrel{\text{def}}{=} \text{true}, \text{false}$
- opérations logiques classiques : $\vee$, $\wedge$, $\neg$
- les variables ne peuvent pas contenir de booléen,
  les variables n'apparaissent donc pas directement dans les conditions

Justification du non-déterminisme

Le non-déterminisme dans un programme permet :

- de modéliser un environnement partiellement inconnu
  (entrées de l'utilisateur, fichiers, communications réseau)
- d'abstraire des parties inconnues du programme (bibliothèques)
- d'abstraire des parties trop complexes du programme (arrondi flottant)
- d'analyser en une fois une famille de programmes (programme paramétré)

Syntaxe :

Non-déterminisme sur les données : $expr ::= \textbf{rand}(c_1, c_2)$.

Le non-déterminisme sur le contrôle peut s'écrire :

"**if** $\textbf{rand}(\mathbf{0}, \mathbf{1}) = 0$ **then** s_1 **else** s_2"

Impact sur la sémantique et la vérification

il nous faut vérifier **toutes** les exécutions possibles
(quelles que soient les entrées, l'implantation des bibliothèques, les paramètres du programme, ...)

$\Longrightarrow$ la sémantique doit donc énumérer **tous** les cas.

Exemple de programme non-déterministe

Exemple

$Y \leftarrow \textbf{rand}(0, 100);$
$X \leftarrow 0;$
$\textbf{while } X \leq Y \textbf{ do}$
$\quad X \leftarrow X + \textbf{rand}(0, 10)$
$\textbf{done};$
$\textbf{assert } X \leq 100$

- l'assertion est-elle vérifiée ?
- quelle est la valeur de X à la fin du programme ?
  (après l'assertion)

Quizz : $\textbf{rand}(0, 1) = \textbf{rand}(0, 1)$ est-il vrai ou faux ?

Choix d'une sémantique concrète collectrice

Sémantique concrète collectrice :

- définition mathématique précise du comportement du programme, servant de base à la construction de l'analyse
  la sûreté de l'analyse abstraite est prouvée vis à vis de cette sémantique
- choisie pour exprimer les propriétés (non décidables) d'intérêt de l'analyse, mais pas plus !

Exemple

$X \leftarrow 100;$
$I \leftarrow 1;$
while $I \leq X$ **do**
    $I \leftarrow I + 3$
done

A priori, plusieurs choix sont possibles :

- sémantique des traces d'exécution :
  $(X = 0, I = 0) \to (X = 100, I = 0) \to (X = 100, I = 1) \to (X = 100, I = 0) \to (X = 100, I = 3) \to (X = 100, I = 6) \to \cdots$
- sémantique des états accessibles :
  $\{(X = 0, I = 0), (X = 100, I = 0), (X = 100, I = 1), (X = 100, I = 0), (X = 100, I = 3), (X = 100, I = 6)\}$
  abstraction de la sémantique de traces...

Pour prouver la sûreté de fonctionnement, les états accessibles suffisent.
(les traces seraient nécessaires pour prouver la terminaison)

Sémantique fonctionnelle

Notre choix : sémantique fonctionnelle d'états

- fonction qui aux états mémoires en entrée du programme
- associe les états mémoires possibles en sortie du programme

C'est donc une abstraction de la sémantique d'états accessibles.

Avantages :

- plus efficace en mémoire ;
  inutile de se souvenir des états aux points de contrôle intermédiaires
- programmable facilement par induction sur la syntaxe
  $\Longrightarrow$ l'analyseur ressemble à un interprète classique ;
- sémantique des triplets de Hoare.

Exemple

$X \leftarrow 100;$
$I \leftarrow 1;$
while $I \leq X$ **do**
    $I \leftarrow I + 3$
done

- sémantique fonctionnelle :
  $(X = x, I = i) \mapsto (X = 100, I = 103)$

  Quelles que soient les valeurs de X et I en entrée, nous avons $X = 100$ et $I = 103$ en sortie.

  Les états intermédiaires sont oubliés.
  En pratique nous gardons quand même une trace des erreurs possibles à l'exécution, par exemple en les affichant au fur et à mesure de l'analyse.

Sémantique

Environnements, sémantique des expressions

$\mathsf{E}[\![\, expr \,]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{Z})$

- les environnements $\mathcal{E} \stackrel{\text{def}}{=} \mathbb{V} \to \mathbb{Z}$
  donnent une valeur dans $\mathbb{Z}$ à chaque variable
  (c'est l'état mémoire du programme)

- $\mathsf{E}[\![\, expr \,]\!]\, \rho$ est l'évaluation de l'expression *expr*
  dans l'environnement $\rho \in \mathcal{E}$.
  Elle renvoie un ensemble de valeurs dans $\mathbb{Z}$
  à cause du non-déterminisme de **rand**.

$$
\begin{array}{lcl}
\mathsf{E}[\![\, V \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{\rho(V)\} \\
\mathsf{E}[\![\, c \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{c\} \\
\mathsf{E}[\![\, \mathbf{rand}(a,b) \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{\, x \in \mathbb{Z} \,|\, a \leq x \leq b \,\}
\end{array}
$$

Notation : la notation de type $\mathsf{X}[\![\, y \,]\!]$ sera souvent utilisée ; y dénote un objet syntaxique ; X permet de distinguer des fonctions sémantiques avec des signatures différentes, dépendant de y (expression, instruction, etc.) ; $\mathsf{X}[\![\, y \,]\!]\, z$ dénote l'application de la fonction $\mathsf{X}[\![\, y \,]\!]$ à la valeur z)

Sémantique des expressions (suite)

$\mathsf{E}[\![\, expr \,]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{Z})$

- définie par induction structurelle sur la syntaxe de l'expression
- nous pouvons avoir $\mathsf{E}[\![\, e \,]\!]\, \rho = \emptyset$ à cause d'une division par zéro
- la valeur $\emptyset$ se propage ;
  ainsi $\mathsf{E}[\![\, 1 + (2/0) \,]\!]\, \rho = \emptyset$ car $\mathsf{E}[\![\, 2/0 \,]\!]\, \rho = \emptyset$.

$$
\begin{array}{lcl}
\mathsf{E}[\![\, -e \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{\, -v \mid v \in \mathsf{E}[\![\, e \,]\!]\, \rho \,\} \\
\mathsf{E}[\![\, e_1 + e_2 \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{\, v_1 + v_2 \mid v_1 \in \mathsf{E}[\![\, e_1 \,]\!]\, \rho,\ v_2 \in \mathsf{E}[\![\, e_2 \,]\!]\, \rho \,\} \\
\mathsf{E}[\![\, e_1 - e_2 \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{\, v_1 - v_2 \mid v_1 \in \mathsf{E}[\![\, e_1 \,]\!]\, \rho,\ v_2 \in \mathsf{E}[\![\, e_2 \,]\!]\, \rho \,\} \\
\mathsf{E}[\![\, e_1 \times e_2 \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{\, v_1 \times v_2 \mid v_1 \in \mathsf{E}[\![\, e_1 \,]\!]\, \rho,\ v_2 \in \mathsf{E}[\![\, e_2 \,]\!]\, \rho \,\} \\
\mathsf{E}[\![\, e_1 / e_2 \,]\!]\, \rho & \stackrel{\text{def}}{=} & \{\, v_1 / v_2 \mid v_1 \in \mathsf{E}[\![\, e_1 \,]\!]\, \rho,\ v_2 \in \mathsf{E}[\![\, e_2 \,]\!]\, \rho \setminus \{0\} \,\}
\end{array}
$$

Quizz : que vaut $\mathsf{E}[\![\, \textbf{rand}(-1,1)/0 \,]\!]\, \rho$? et $\mathsf{E}[\![\, 1/\textbf{rand}(-1,1) \,]\!]\, \rho$?

Sémantique des conditions booléennes

$\mathsf{C}[\![\, cond \,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

- Notre sémantique manipule des ensembles d'environnements.
- Les condition agissent comme des filtres ;
  elles sélectionnent les états mémoires qui satisfont la condition
  et peuvent continuer l'exécution du programme.

$$
\begin{array}{lcl}
\mathsf{C}[\![\, \mathrm{true} \,]\!]\, R & \stackrel{\text{def}}{=} & R \\
\mathsf{C}[\![\, \mathrm{false} \,]\!]\, R & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{C}[\![\, c_1 \wedge c_2 \,]\!]\, R & \stackrel{\text{def}}{=} & \mathsf{C}[\![\, c_1 \,]\!]\, R \cap \mathsf{C}[\![\, c_2 \,]\!]\, R \\
\mathsf{C}[\![\, c_1 \vee c_2 \,]\!]\, R & \stackrel{\text{def}}{=} & \mathsf{C}[\![\, c_1 \,]\!]\, R \cup \mathsf{C}[\![\, c_2 \,]\!]\, R \\
\mathsf{C}[\![\, e_1 = e_2 \,]\!]\, R & \stackrel{\text{def}}{=} & \{\, \rho \in R \mid \exists v_1 \in \mathsf{E}[\![\, e_1 \,]\!]\, \rho, v_2 \in \mathsf{E}[\![\, e_2 \,]\!]\, \rho \colon v_1 = v_2 \,\} \\
\mathsf{C}[\![\, e_1 < e_2 \,]\!]\, R & \stackrel{\text{def}}{=} & \{\, \rho \in R \mid \exists v_1 \in \mathsf{E}[\![\, e_1 \,]\!]\, \rho, v_2 \in \mathsf{E}[\![\, e_2 \,]\!]\, \rho \colon v_1 < v_2 \,\} \\
\ldots & &
\end{array}
$$

Sémantique des conditions booléennes (suite)

- La négation $\neg$ peut être gérée par transformation d'expression :

$$\begin{array}{ll} \neg(e_1 = e_2) \to e_1 \neq e_2 & \neg(c_1 \vee c_2) \to (\neg c_1) \wedge (\neg c_2) \\ \neg(e_1 < e_2) \to e_1 \geq e_2 & \neg(c_1 \wedge c_2) \to (\neg c_1) \vee (\neg c_2) \\ \ldots & \end{array}$$

$\neg$ est éliminé par lois de De Morgan

- On peut vérifier que :
  - $\mathsf{C}[\![\, cond \,]\!]\, \{\rho\}$ est soit $\{\rho\}$, soit $\emptyset$;
  - $\mathsf{C}[\![\, cond \,]\!]\, R = \cup_{\rho \in R}\, \mathsf{C}[\![\, cond \,]\!]\, \{\rho\}$ (morphisme complet pour l'union)

  $\Longrightarrow$ chaque environnement est filtré indépendamment des autres.

Quizz : que se passe-t-il en cas d'erreur dans l'expression ?

- $\mathsf{C}[\![\, 1 = (1/0) \,]\!]\, R$
- $\mathsf{C}[\![\, \mathrm{true} \vee (1 = 1/0) \,]\!]\, R$

Domaine sémantique des instructions

$\mathsf{S}[\![\, stat \,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

- Notre sémantique manipule des ensembles d'environnements.
- $Q = \mathsf{S}[\![\, stat \,]\!]\, P$ signifie :
  - si l'état mémoire est dans P avant d'exécuter *stat*
  - alors l'état mémoire est dans Q après avoir exécuté *stat*
- $\mathsf{S}[\![\, stat \,]\!]\, R = \cup_{\rho \in R}\, \mathsf{S}[\![\, stat \,]\!]\, \{\rho\}$

  la signature $\mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ permet de composer facilement les $\mathsf{S}[\![\, stat \,]\!]$
  $\Longrightarrow$ simplifie l'écriture des fonctions
- $\mathsf{S}[\![\, stat \,]\!]\, \{\rho\} = \emptyset$ si l'exécution de *stat* sur ρ :
  - rencontre un erreur, ou
  - boucle infiniment.

Quizz : quelle est la sémantique de $X \leftarrow 0;$ **while** $\mathbf{rand}(\mathbf{0}, \mathbf{1}) = \mathbf{0}$ **do done** ?

Sémantique des instructions

$\mathsf{S}[\![\, stat \,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

- skip : ne fait rien

  $$\mathsf{S}[\![\, \mathbf{skip} \,]\!]\, R \stackrel{\text{def}}{=} R$$

- halt : arrête le programme

  $$\mathsf{S}[\![\, \mathbf{halt} \,]\!]\, R \stackrel{\text{def}}{=} \emptyset$$

- affectation : évalue l'expression et met à jour l'environnement

  $$\mathsf{S}[\![\, X \leftarrow e \,]\!]\, R \stackrel{\text{def}}{=} \{\, \rho[X \mapsto v] \mid \rho \in R,\ v \in \mathsf{E}[\![\, e \,]\!]\, \rho \,\}$$

  - pour chaque environnement ρ
  - pour chaque valeur v de l'expression, dans $\mathsf{E}[\![\, e \,]\!]\, \rho$ (non-déterminisme)
  - génère un nouvel environnement $\rho[X \mapsto v]$ où X est mis à jour

$f[x \mapsto y]$ dénote la fonction qui associe y à x, et $f(z)$ à tout $z \neq x$

Sémantique des instructions (suite)

$\mathsf{S}[\![\, stat \,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

- séquence : composition de fonctions

$$\mathsf{S}[\![\, s_1 ; s_2 \,]\!] \stackrel{\text{def}}{=} \mathsf{S}[\![\, s_2 \,]\!] \circ \mathsf{S}[\![\, s_1 \,]\!]$$

- test : filtrage par la condition et exécution des branches

$$\begin{array}{l} \mathsf{S}[\![\, \textbf{if } c \textbf{ then } s_1 \textbf{ else } s_2 \,]\!]\, R \stackrel{\text{def}}{=} \\ \qquad \mathsf{S}[\![\, s_1 \,]\!]\, (\mathsf{C}[\![\, c \,]\!]\, R) \cup \\ \qquad \mathsf{S}[\![\, s_2 \,]\!]\, (\mathsf{C}[\![\, \neg c \,]\!]\, R) \end{array}$$

  - filtre les environnements par e
  - exécute les deux branches indépendamment
  - union des comportements des deux branches, avec $\cup$

- assert : filtrage par une condition

$$\mathsf{S}[\![\, \textbf{assert } c \,]\!]\, R \stackrel{\text{def}}{=} \mathsf{C}[\![\, c \,]\!]\, R$$

et affiche une erreur à l'écran si la condition n'est pas toujours vraie ; i.e., si $\mathsf{C}[\![\, \neg c \,]\!]\, R \neq \emptyset$.

Sémantique des boucles

Calcul de : $\mathsf{S}[\![\, \textbf{while}\ c\ \textbf{do}\ s \,]\!]\, R$

1. calculer un invariant de boucle I
   i.e, les environnements atteignables après zéro, un ou plusieurs tours de boucle, au point : **while** • c **do** s
2. nous avons ensuite, $\mathsf{S}[\![\, \textbf{while}\ c\ \textbf{do}\ s \,]\!]\, R = \mathsf{C}[\![\, \neg c \,]\!]\, I$

Le calcul de I peut se faire par un principe d'induction
si $\rho \in I$, alors :

- soit il s'agit du premier tour de boucle : $\rho \in R$;
- sinon, ρ est dans l'image de I par un tour de boucle :
  $\rho \in \mathsf{S}[\![\, s \,]\!]\, (\mathsf{C}[\![\, c \,]\!]\, I)$

Soit donc à résoudre $I = F(I)$ où $F(X) = R \cup \mathsf{S}[\![\, s \,]\!]\, (\mathsf{C}[\![\, c \,]\!]\, X)$.

La **théorie de l'ordre** justifie que :

- cette équation a au moins une solution ;
- cette équation a une plus petite solution ;
  $\mathrm{lfp}\, F$: le plus petit point fixe de F
  $\Longrightarrow$ c'est l'invariant optimal recherché.

Éléments de théorie de l'ordre

Ordres partiels

Ordres partiels

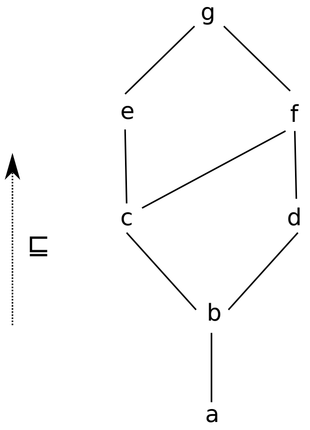
Étant donné un ensemble X,
la relation $\sqsubseteq \in X \times X$ est un ordre partiel si elle est :

1. réflexive : $\forall x \in X,\ x \sqsubseteq x$
2. anti-symmetrique : $\forall x, y \in X,\ x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$
3. transitive : $\forall x, y, z \in X,\ x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$

$(X, \sqsubseteq)$ est un poset *(partially ordered set)*.

Exemples d'ordres partiels

- Donné par un diagramme de Hasse, e.g. :



$g \sqsubseteq g$
$f \sqsubseteq f, g$
$e \sqsubseteq e, g$
$d \sqsubseteq d, f, g$
$c \sqsubseteq c, e, f, g$
$b \sqsubseteq b, c, d, e, f, g$
$a \sqsubseteq a, b, c, d, e, f, g$

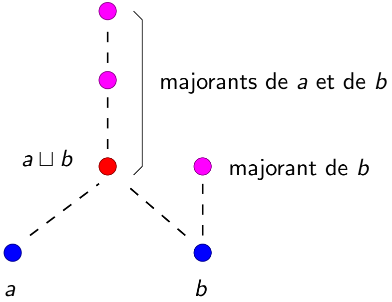
Exemples d'ordres partiels

Ordres partiels :

- $(\mathbb{Z}, \leq)$
  (ordre complet)
- $(\mathcal{P}(X), \subseteq)$
  (ordre non complet : $\{1\} \not\subseteq \{2\}$, $\{2\} \not\subseteq \{1\}$)
- $(\mathbb{Z}^2, \sqsubseteq)$, où $(a, b) \sqsubseteq (a', b') \iff a \geq a' \wedge b \leq b'$
  (ordre des bornes d'intervalles pour avoir l'inclusion)

(Plus petit) majorant

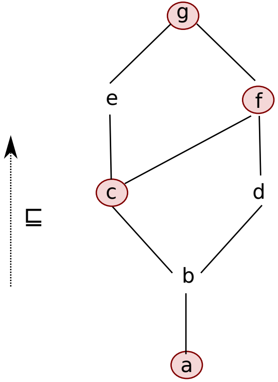
- c est un majorant (upper bound) de a et de b si : $a \sqsubseteq c$ et $b \sqsubseteq c$
- c est un plus petit majorant (least upper bound, lub ou join) de a et b si
  - c est un majorant de a et de b
  - pour tout majorant d de a et b, $c \sqsubseteq d$



- si elle existe, la lub est unique
- généralisable aux lubs d'ensembles arbitraires $\sqcup Y$, $Y \subseteq X$
- notion duale de (plus grand) minorant $\sqcap$ (glb, meet)

Chaînes

$C \subseteq X$ est une chaine de $(X, \sqsubseteq)$
si elle est totalement ordonnée par $\sqsubseteq$:
$\forall x, y \in C,\ x \sqsubseteq y \vee y \sqsubseteq x$.



$a \sqsubseteq c \sqsubseteq f \sqsubseteq g$

Ordre partiel complet (CPO)

Un ordre partiel $(X, \sqsubseteq)$ est complet (CPO)
si toute chaîne C a un plus petit majorant $\sqcup C$.

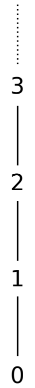
Exemples :

- $(X, \sqsubseteq)$ est complet si X est fini.
- $(\mathcal{P}(Y), \subseteq)$ est complet pour tout Y (même infini).

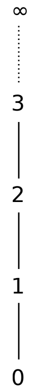
Ce plus petit majorant correspond à la notion de limite :
dans un CPO, toute séquence croissante a une limite dans le CPO.

Un CPO a un plus petit élément $\sqcup \emptyset$, denoté $\perp$.

Ordres complets et incomplets



$(\mathbb{N}, \leq)$
non complet



$(\mathbb{N} \cup \{ \infty \}, \leq)$
complet

Par exemple, $\max \{ 2x \mid x \in \mathbb{N} \}$ existe dans $\mathbb{N} \cup \{ \infty \}$, mais pas dans $\mathbb{N}$.

Treillis

Treillis

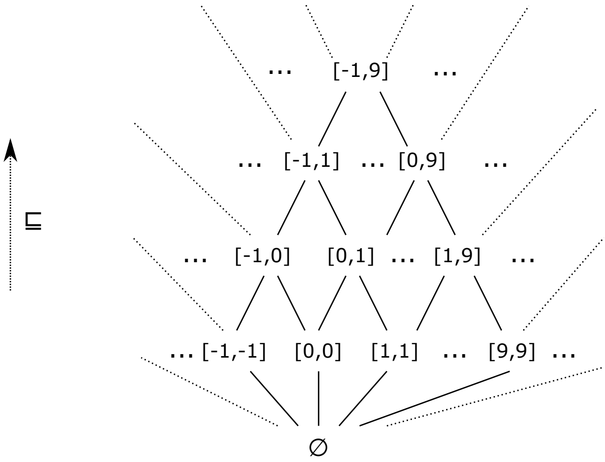
Un treillis $(X, \sqsubseteq, \sqcup, \sqcap)$ est un ordre partiel avec

1. une lub $a \sqcup b$ pour toute paire d'éléments a, b ;
2. une glb $a \sqcap b$ pour toute paire d'éléments a, b.

Exemples :

- les entiers $(\mathbb{Z}, \leq, \max, \min)$
- les intervalles d'entiers (transparent suivant)

Exemple : les intervalles d'entiers



Intervalles d'entiers : $(\{\,[a,b] \mid a,b \in \mathbb{Z},\ a \leq b\,\} \cup \{\,\emptyset\,\}, \subseteq, \sqcup, \cap)$

où $[a,b] \sqcup [a',b'] \stackrel{\text{def}}{=} [\min(a,a'), \max(b,b')]$.

Treillis complets

Un treillis complet $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ est un ordre partiel ayant

1. une lub $\sqcup S$ pour tout ensemble $S \subseteq X$
2. une glb $\sqcap S$ pour tout ensemble $S \subseteq X$
3. un plus petit élément $\bot$
4. un plus grand élément $\top$

Notes : $\star\star\star$

- 1 implique 2 car $\sqcap S = \sqcup \{\, y \mid \forall x \in S,\ y \sqsubseteq x \,\}$
  (et 2 implique 1 de même),
- 1 et 2 impliquent 3 et 4 : $\bot = \sqcup \emptyset = \sqcap X$, $\top = \sqcap \emptyset = \sqcup X$,

Tout treillis complet est aussi un CPO ;
l'inverse n'est pas vrai.

Exemples de treillis complets

- tout treillis fini est complet
- les intervalles entiers avec des bornes finies ou infinies :

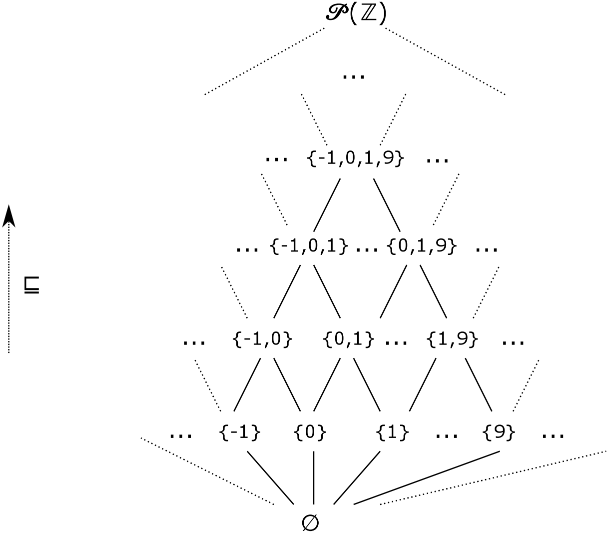
  $(\{\,[a,b] \mid a \in \mathbb{Z} \cup \{\,-\infty\,\},\ b \in \mathbb{Z} \cup \{\,+\infty\,\},\ a \leq b\,\} \cup \{\,\emptyset\,\},$
  $\subseteq,\ \sqcup,\ \cap,\ \emptyset,\ [-\infty,+\infty])$

  où $\sqcup_{i \in I} [a_i, b_i] \stackrel{\text{def}}{=} [\min_{i \in I} a_i,\ \max_{i \in I} b_i]$.
- $(\mathcal{P}(S), \subseteq, \cup, \cap, \emptyset, S)$ est complet

  (voir transparent suivant)

Exemple : les parties d'un ensemble

Exemple : $(\mathcal{P}(\mathbb{Z}), \subseteq, \cup, \cap, \emptyset, \mathbb{Z})$



Fonctions

Une fonction $f : (X_1, \sqsubseteq_1, \sqcup_1, \perp_1) \to (X_2, \sqsubseteq_2, \sqcup_2, \perp_2)$ est

- croissante si : $\forall x, x', \, x \sqsubseteq_1 x' \implies f(x) \sqsubseteq_2 f(x')$

  (en anglais : *monotonic*, qui veut bien dire croissante et non monotone)

- stricte si : $f(\perp_1) = \perp_2$

- continue entre des CPO si :

  pour toute chaîne C de X_1
  son image $\{ f(c) \mid c \in C \}$ est une chaîne dans X_2
  et $f(\sqcup_1 C) = \sqcup_2 \{ f(c) \mid c \in C \}$

  l'image d'une limite est la limite des images,
  donc pas de surprise lors du passage à la limite

Points fixes

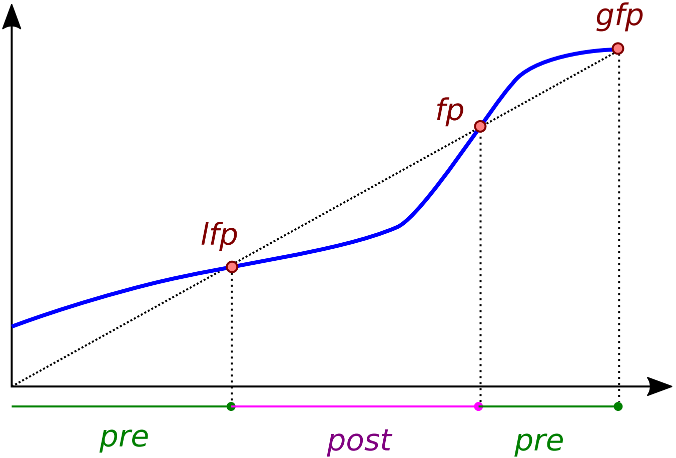
Étant donnée $f : (X, \sqsubseteq) \to (X, \sqsubseteq)$

- x est un point fixe de f si $f(x) = x$
- x est un pre point fixe de f si $x \sqsubseteq f(x)$
- x est un post point fixe de f si $f(x) \sqsubseteq x$

Il peut y avoir plusieurs points fixes, ou aucun.
Nous notons $\mathrm{lfp}\, f$ le plus petit point fixe de f.

Beaucoup de problèmes de mathématique, de physique, mais aussi d'informatique peuvent se réduire à un calcul de point fixe.

Points fixes : illustration

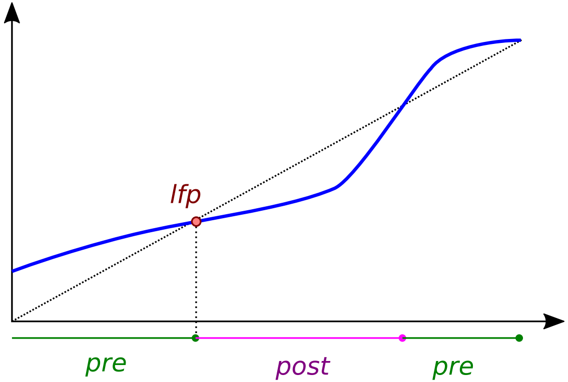


Théorème de point fixe de Tarski ⁂

Théorème de Tarksi

Si $f : X \to X$ est croissante dans un treillis complet X alors $\mathrm{lfp}\, f$ existe
(en fait, l'ensemble des points fixes de f forme un treillis)

Le théorème précise que :
$\mathrm{lfp}\, f = \sqcap \{\, x \mid f(x) \sqsubseteq x \,\}$ (i.e., le plus petit des post points fixes)

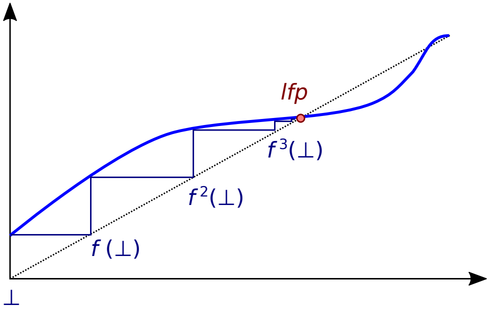


Théorème de point fixe de "Kleene"

Théorème de "Kleene"

Si $f : X \to X$ est continue dans un CPO X alors $\mathrm{lfp}\, f$ existe.

Le théorème précise que : $\{ f^n(\bot) \mid n \in \mathbb{N} \}$ est une chaîne
et que $\mathrm{lfp}\, f = \sqcup \{ f^n(\bot) \mid n \in \mathbb{N} \}$.



Théorème de point fixe de "Kleene" : preuve ⋆⋆⋆

Preuve d'une petite généralisation du théorème...

Hypothèse : $f : X \to X$ est continue et $a \sqsubseteq f(a)$.

À prouver : $\mathrm{lfp}_a\, f = \sqcup \{\, f^n(a) \mid n \in \mathbb{N} \,\}$. ($\mathrm{lfp}_a$ est le plus petit point fixe, plus grand que a)

$a \sqsubseteq f(a)$ par hypothèse.
$f(a) \sqsubseteq f(f(a))$ par croissance de f. (toute fonction continue est croissante)
Par récurrence, $\forall n,\ f^n(a) \sqsubseteq f^{n+1}(a)$.
Donc, $\{\, f^n(a) \mid n \in \mathbb{N} \,\}$ est une chaîne et donc $\sqcup \{\, f^n(a) \mid n \in \mathbb{N} \,\}$ existe.

$f(\sqcup \{\, f^n(a) \mid n \in \mathbb{N} \,\})$
$= \sqcup \{\, f^{n+1}(a) \mid n \in \mathbb{N} \,\})$ (par continuité)
$= a \sqcup (\sqcup \{\, f^{n+1}(a) \mid n \in \mathbb{N} \,\})$ (tous les $f^{n+1}(a)$ sont plus grands que a)
$= \sqcup \{\, f^n(a) \mid n \in \mathbb{N} \,\}$.
Donc, $\sqcup \{\, f^n(a) \mid n \in \mathbb{N} \,\}$ est un point fixe de f.

De plus, tout point fixe plus grand que a doit être plus grand que tous les $f^n(a)$, $n \in \mathbb{N}$.
Donc, $\sqcup \{\, f^n(a) \mid n \in \mathbb{N} \,\} = \mathrm{lfp}_a\, f$.

Retour à la sémantique des boucles

Sémantique des boucles

$\mathsf{S}[\![\, \textbf{while}\ c\ \textbf{do}\ s \,]\!]\, R \stackrel{\text{def}}{=} \mathsf{C}[\![\, \neg c \,]\!]\, (\text{lfp}\ F)$

où $F(X) \stackrel{\text{def}}{=} R \cup \mathsf{S}[\![\, s \,]\!]\, (\mathsf{C}[\![\, c \,]\!]\, X)$

Justification : lfp F existe

Nous appliquons encore les théorèmes de points-fixes existants ; avec les hypothèses suivantes :

- $(\mathcal{P}(\mathcal{E}), \subseteq, \cup, \cap, \emptyset, \mathcal{E})$ est un treillis complet
  c'est donc aussi un CPO
  et toutes les séquences croissantes ont une limite
- toutes les fonctions sémantiques sont continues
  ce sont en fait des morphismes complets pour $\cup$
  cela se prouve par induction sur la syntaxe des expressions

Note :

en cas de boucles imbriquées,
la sémantique calcule des points fixes imbriqués. . .

Interprétation comme limite de chaîne croissante

$\mathsf{S}[\![\, \textbf{while}\ c\ \textbf{do}\ s \,]\!]\, R \stackrel{\text{def}}{=} \mathsf{C}[\![\, \neg c \,]\!]\,(\mathrm{lfp}\ F)$

où $F(X) \stackrel{\text{def}}{=} R \cup \mathsf{S}[\![\, s \,]\!]\,(\mathsf{C}[\![\, c \,]\!]\, X)$

F applique une itération de la boucle à X
décalant les itérés accumulées d'un cran
puis F ajoute les environnements R avant la boucle, i.e., les itérés au rang 0

Le théorème de Kleene précise que $\mathrm{lfp}\ F = \cup_{n \in \mathbb{N}}\ F^n(\emptyset)$

- $F^0(\emptyset) = \emptyset$
- $F^1(\emptyset) = R$
  environnements avant d'entrer dans la boucle
- $F^2(\emptyset) = R \cup \mathsf{S}[\![\, s \,]\!]\,(\mathsf{C}[\![\, c \,]\!]\, R)$
  environnements après zéro ou une itération de boucle
- $F^n(\emptyset)$: environnements après au plus $n - 1$ itérations de boucle
  juste avant de tester la condition de sortie,
  pour déterminer si une $n-$ième itération est nécessaire
- $\cup_{n \in \mathbb{N}}\ F^n(\emptyset)$ est bien l'invariant de boucle

Vous pourrez observer les itérations sur des exemples en TME...

Implantation de la sémantique concrète en OCaml

Analyseur

Les sources fournies pour le projet / TME comprennent :

- un *front-end* permettant de lire un source en texte et de le transformer en arbre syntaxique
- un itérateur générique permettant de parcourir l'arbre

L'itérateur est paramétré par un domaine d'interprétation :

- dans ce cours : le domaine concret des ensembles environnements
- dans le futur : des domaines abstraits d'analyse statique
- nous utilisons des foncteurs OCaml :
  - signature `DOMAIN` dans `domains/domain.ml`
  - implantation de la signature dans `domains/concrete_domain.ml`
  - utilisation dans `interpreter/interpreter.ml`

Exemple de programme

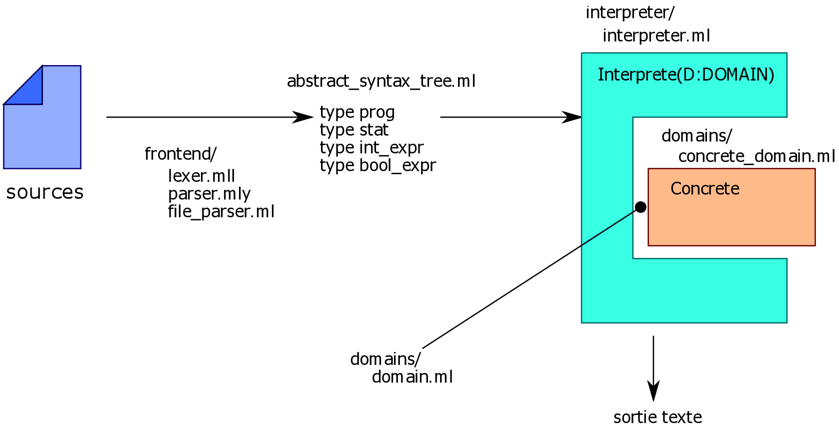
tests/0405_loop_rel.c

```
{
  int N;
  int x;
  N = rand(0,50);
  x = 0;
  while (x < N) {
    print(x,N);
    x = x + rand(0,3);
  }
  assert(x>=N && x<N+3);
}
```

- blocs introduisant des variables locales
  (pas de variablesglobale)
- type unique : entiers mathématiques $\mathbb{Z}$
- expressions arithmétiques et booléennes
- non-déterminisme avec `rand`
- `while`, `if`, `assert`
- affichage des variables avec `print`

De nombreux autres exemples dans `tests/`

Schéma de fonctionnement de l'interprète concret



Technologie du *front-end* : OCamlLex et Menhir ⋆⋆⋆

lexer.mll

```
{
open Lexing
open Abstract_syntax_tree
open Parser

(* table des mots-clés *)
let kwd_table = Hashtbl.create 10
let _ =
  List.iter (fun (a,b) ->
   Hashtbl.add kwd_table a b)
   [ "int",    TOK_INT;
     "true",   TOK_TRUE;
     "false",  TOK_FALSE;
     ...
(* utilitaires *)
let space = [' ' '\t' '\r']+
let newline = "\n" | "˚" | "\r\n"
...
(* règles lexicales *)
rule token = parse
| ['a'-'z' 'A'-'Z' '_']
  ['a'-'z' 'A'-'Z' '0'-'9' '_']* as id
{ try Hashtbl.find kwd_table id
   with Not_found -> TOK_id id }
| "("    { TOK_LPAREN }
| ")"    { TOK_RPAREN }
...
```

parser.mly

```
%{
open Abstract_syntax_tree
%}
(* liste des symboles *)
%token TOK_INT
%token TOK_TRUE
...
(* priorité et associativité des symboles *)
%left TOK_BAR_BAR
%left TOK_AND_AND
...
(* règle racine avec son type *)
%start<Abstract_syntax_tree.prog> file
file: t=list(ext(stat)) TOK_EOF { t }
...
(* règle pour les expressions entières *)
int_expr:
| TOK_LPAREN e=int_expr TOK_RPAREN    { e }
| e=ext(TOK_int)        { AST_int_const e }
| e=ext(TOK_id)        { AST_identifier e }
| o=int_unary_op e=ext(int_expr)
    { AST_int_unary (o,e) }
| e1=ext(int_expr) o=int_binary_op
  e2=ext(int_expr)
    { AST_int_binary (o,e1,e2) }
...
```

Arbre syntaxique abstrait

abstract_syntax_tree.ml

```ocaml
type 'a ext = 'a * extent (* ajoute une information de position dans le source *)

type int_unary_op =  (* opérateurs unaires entiers *)
  | AST_UNARY_PLUS
  | AST_UNARY_MINUS

type int_binary_op = ... (* opérateurs binaires entiers *)

type int_expr = (* expressions entières *)
  | AST_int_unary  of int_unary_op * (int_expr ext)
  | AST_int_binary of int_binary_op * (int_expr ext) * (int_expr ext)
  | AST_identifier of var ext
  | AST_int_const  of string ext
  | AST_rand       of (string ext) * (string ext)

type bool_expr = ... (* expressions booléennes *)

type stat = (* instructions *)
  | AST_block    of (((typ * var) ext) list) * ((stat ext) list)
  | AST_assign   of (var ext) * (int_expr ext)
  | AST_if       of (bool_expr ext) * (stat ext) * (stat ext option)
  | AST_while    of (bool_expr ext) * (stat ext)
  | AST_HALT
  | AST_assert   of bool_expr ext
  | AST_print    of (var ext) list

type prog = stat ext list (* programmes *)
```

Bibliothèques : `Zarith` et `Map`

Représentation des entiers :

Les entiers d'OCaml font 31-bit ou 63 bits.
Notre langage utilise des entiers non bornés.
$\Longrightarrow$ utilisation d'une bibliothèque d'entiers longs

- module `Z`
- type des entiers : `Z.t`
- opérations classiques : `Z.add`, `Z.sub`, ...
- interface similaire à `Int64`

Représentation des environnements : (voir `domains/concrete_domain.ml`)

Nous avons besoin d'ensembles de fonctions : $\mathcal{P}(\mathbb{V} \to \mathbb{Z})$.

- représentations de fonctions dans $\mathbb{V} \to \mathbb{Z}$
  - ```
    module Env = Mapext.Make(String)
    type env = Z.t Env.t
    ```
  - utilisation du module `Mapext` généralisant `Map` d'OCaml
- pour un ensemble de fonctions, nous utilisons le module `Set`
  - ```
    module EnvSet = Set.Make (struct
        type t = env
        let compare = Env.compare Z.compare end)
    ```

Domaine d'interprétation : signature

domains/domain.ml

```
module type DOMAIN =
sig
    type t (* ensemble d'environnements *)

    (* initialisation *)
    val init: unit -> t     (* aucune variable *)
    val bottom: unit -> t   (* ∅ *)

    val add_var: t -> var -> t  (* ajout de variable *)
    val del_var: t -> var -> t  (* retrait de variable *)

    val assign: t -> var -> int_expr -> t (* V := e *)
    val compare: t -> int_expr -> compare_op -> int_expr -> t (* e1 cmp e2 *)

    val join: t -> t -> t       (* ∪ *)
    val meet: t -> t -> t       (* ∩ *)
    val widen: t -> t -> t
    val subset: t -> t -> bool (* ⊆ *)
    val is_bottom: t -> bool   (* = ∅? *)

    (* affichage *)
    val print: Format.formatter -> t -> var list -> unit
    val print_all: Format.formatter -> t -> unit
end
```

Domaine d'interprétation concrète

domains/concrete_domain.ml

```
module Concrete = (struct
  type t = EnvSet.t              (* 𝒫(ℰ) *)
  module ValSet = Set.Make(Z)   (* 𝒫(ℤ) *)

  let int_map (f:Z.t -> Z.t) (s:ValSet.t) : ValSet.t = (* { f(x) | x ∈ s } *)
    ValSet.fold (fun x acc -> ValSet.add (f x) acc) s ValSet.empty

  let rec eval_expr (e:int_expr) (m:env) : ValSet.t =  (* E⟦ e ⟧ m *)
    match e with
    | AST_int_unary (op,(e1,_)) ->
        let s1 = eval_expr e1 m in
        int_map
          (match op with
          | AST_UNARY_PLUS -> fun x -> x
          | AST_UNARY_MINUS -> Z.neg
          ) s1
    | ...
  (* S⟦ v ← e ⟧ m = { ρ[v ↦ x] | ρ ∈ m, x ∈ E⟦ e ⟧ ρ } *)
  let assign (m:t) (v:string) (e:int_expr) : t =
    EnvSet.fold
      (fun env acc ->
        let s = eval_expr e env in
        ValSet.fold (fun v acc -> EnvSet.add (Env.add v v env) acc) s acc
      ) m EnvSet.empty

  let join m1 m2 = EnvSet.union m1 m2 (* ∪ *)
end: DOMAIN)
```

Itérateur

interpreter/interpreter.ml

```
module Interprete(D : DOMAIN) = struct
  type t = D.t (* 𝒫(ℰ) *)

  (* C⟦ e ⟧ a si r=true,  C⟦ ¬ e ⟧ a si r=false *)
  let filter (a:t) (e:bool_expr ext) (r:bool) : t =
    let rec doit a (e,x) r = match e with
    | AST_bool_unary (AST_NOT, e) ->  doit a e (not r)
    | AST_bool_binary (AST_AND, e1, e2) ->
      (if r then D.meet else D.join) (doit a e1 r) (doit a e2 r)
    | ...
    | AST_compare (cmp, (e1,_), (e2,_)) ->
        let inv = function
        | AST_EQUAL         -> AST_NOT_EQUAL
        | AST_NOT_EQUAL     -> AST_EQUAL
        | AST_LESS          -> AST_GREATER_EQUAL
        | AST_LESS_EQUAL    -> AST_GREATER
        | AST_GREATER       -> AST_LESS_EQUAL
        | AST_GREATER_EQUAL -> AST_LESS
        in
        let cmp = if r then cmp else inv cmp in
        D.compare a e1 cmp e2
    in
    doit a e r
```

Itérateur (suite)

interpreter/interpreter.ml

```
let rec eval_stat (a:t) ((s,ext):stat ext) : t = (* S⟦ s ⟧ a *)
  match s with
  | AST_assign ((i,_),(e,_)) -> D.assign a i e

  | AST_if (e,s1,Some s2) ->
      let t = eval_stat (filter a e true ) s1 in
      let f = eval_stat (filter a e false) s2 in
      D.join t f

  | AST_assert e ->
      a

  | AST_block (decl,inst) ->
      let a = List.fold_left (fun a ((_,v),_) -> D.add_var a v) a decl in
      let a = List.fold_left eval_stat a inst in
      List.fold_left (fun a ((_,v),_) -> D.del_var a v) a decl

  | AST_while (e,s) ->
      let rec fix (f:t -> t) (x:t) : t =
        let fx = f x in
        if D.subset fx x then fx
        else fix f fx
      in
      let f x = D.join a (eval_stat (filter x e true) s) in
      let inv = fix f a in
      filter inv e false
```