INF889A

Analyse de programmes pour la sécurité logicielle

Chapitre 4 - Analyse de flot de données

Quentin Stiévenart

Hiver 2024

Ressources

- Static program analysis, Møller, A., & Schwartzbach, M. I.
  - Chapitres 4 et 5
- Program Analysis, cours de Michael Pradel de 2020-2021.
  - Chapitre “Data flow analysis”

Fondements théoriques de l'analyse statique

Problème de l'arrêt

Pour un modèle de calcul Turing complet : on veut une méthode pour **décider** pour tout programme, s'il s'arrête ou boucle indéfiniment.

- Est-ce que `while (true) { }` s'arrête ?

Problème de l'arrêt

Pour un modèle de calcul Turing complet : on veut une méthode pour **décider** pour tout programme, s'il s'arrête ou boucle indéfiniment.

- Est-ce que `while (true) { }` s'arrête ?
- Est-ce que `printf("hello\n");` s'arrête ?

Problème de l'arrêt

Pour un modèle de calcul Turing complet : on veut une méthode pour **décider** pour tout programme, s'il s'arrête ou boucle indéfiniment.

- Est-ce que `while (true) { }` s'arrête ?
- Est-ce que `printf("hello\n");` s'arrête ?
- Est-ce que `printf(input());` s'arrête ?

Problème de l'arrêt

Pour un modèle de calcul Turing complet : on veut une méthode pour **décider** pour tout programme, s'il s'arrête ou boucle indéfiniment.

- Est-ce que `while (true) { }` s'arrête ?
- Est-ce que `printf("hello\n");` s'arrête ?
- Est-ce que `printf(input());` s'arrête ?
- Est-ce que le programme suivant s'arrête ?

```
int f(int n) {
  while (n > 1) {
    if (n % 2 == 0) n = n/2;
    else n = 3*n + 1;
  }
}
```

Problème de l'arrêt

Pour un modèle de calcul Turing complet : on veut une méthode pour **décider** pour tout programme, s'il s'arrête ou boucle indéfiniment.

- Est-ce que `while (true) { }` s'arrête ?
- Est-ce que `printf("hello\n");` s'arrête ?
- Est-ce que `printf(input());` s'arrête ?
- Est-ce que le programme suivant s'arrête ?

```
int f(int n) {
  while (n > 1) {
    if (n % 2 == 0) n = n/2;
    else n = 3*n + 1;
  }
}
```

$\rightarrow$ conjecture de Collatz, non prouvée depuis 1937

Problème de l'arrêt : brouillon de preuve

Hypothèse : une telle méthode P existe

On pose $Q =$ `if(P(Q)) { while (true) {} }`

- Soit $P(Q)$ est vrai : Q termine
  - on exécute `while (true) {}`
  - Q ne termine pas $\rightarrow$ contradiction
- Soit $P(Q)$ est faux : Q ne termine pas
  - on n'exécute **pas** `while (true) {}`
  - Q termine $\rightarrow$ contradiction

Donc P n'existe pas : on ne peut avoir une méthode P qui décide si un programme s'arrête.

Turing. (1936). On computable numbers, with an application to the Entscheidungsproblem. J. of Math.

Théorème de Rice

Théorème de Rice

Toute propriété sémantique non triviale des programmes est indécidable.

- sémantique : qui n'est pas limitée à la syntaxe, par exemple :
  - syntaxique = le programme contient un appel à `gets`
  - sémantique = le programme exécute un appel à `gets`
- non-triviale : qui n'est pas toujours vraie ou fausse

Rice. (1953). Classes of Recursively Enumerable Sets and their Decision Problems. Transactions of the American Mathematical society.

Théorème de Rice : brouillon de preuve

Idée de la preuve : réduction au problème de l'arrêt

On pose $P(x) =$ vrai si x est un programme qui retourne toujours 0, faux sinon.

On construit $H(a, i) = P(\texttt{a(i); return 0})$

- Si a(i) s'arrête, H retourne toujours vrai
- Si a(i) ne s'arrête pas, H retourne toujours faux

Donc H décide de l'arrêt de a sur $i \rightarrow$ impossible

Autre argument

On pose $P(x) =$ vrai si x est un programme qui retourne toujours 0, faux sinon.

$Q() =$ `return hash(input()) == 0x12345678`

Trouver P revient à briser n'importe quelle fonction de hachage.

Conséquences pour l'analyse de programme

N'importe quelle propriété « intéressante » est indécidable

On change les règles du jeu : on va donner des P tels que, pour un programme x :

- soit P prouve la propriété est vraie
- soit P prouve que la propriété est fausse
- soit P répond "peut-être" : la propriété peut être vraie ou fausse

L'analyseur statique *trivial* répond toujours "peut-être"

Approximation de l'analyse : problème de l'arrêt

Soit $P =$ exécuter le programme sur 10 étapes et voir s'il s'est arrêté

- Est-ce que `while (true) { }` s'arrête ?

Approximation de l'analyse : problème de l'arrêt

Soit $P =$ exécuter le programme sur 10 étapes et voir s'il s'est arrêté

- Est-ce que `while (true) { }` s'arrête ?
  - non $\rightarrow$ vrai négatif
- Est-ce que `printf("hello\n");` s'arrête ?

Approximation de l'analyse : problème de l'arrêt

Soit $P =$ exécuter le programme sur 10 étapes et voir s'il s'est arrêté

- Est-ce que `while (true) { }` s'arrête ?
  - non $\rightarrow$ vrai négatif
- Est-ce que `printf("hello\n");` s'arrête ?
  - oui $\rightarrow$ vrai positif
- Est-ce que le programme suivant s'arrête ?

```
int f(int n) {
  while (n > 1) {
    if (n % 2 == 0) n = n/2;
    else n = 3*n + 1;
  }
}
```

non $\rightarrow$???

Approximation de l'analyse : antivirus

- $P =$ exécuter le programme lancera-t-il un virus ?

Approximation de l'analyse : antivirus

- $P =$ exécuter le programme lancera-t-il un virus ?
  propriété non triviale $\rightarrow$ indécidable
- $P =$ le code du programme contient-il la signature d'un virus connu ?

Approximation de l'analyse : antivirus

- $P =$ exécuter le programme lancera-t-il un virus ?
  propriété non triviale $\rightarrow$ indécidable
- $P =$ le code du programme contient-il la signature d'un virus connu ?
  propriété "finie" $\rightarrow$ décidable (statiquement)

Question : l'une propriété implique-t-elle l'autre ?

Approximation de l'analyse : fuzzer

- $P =$ le programme contient-il un bug ?

Approximation de l'analyse : fuzzer

- $P =$ le programme contient-il un bug ?
  $\rightarrow$ indécidable
- $P =$ le programme plante-t-il avec l'entrée x ?

Approximation de l'analyse : fuzzer

- $P =$ le programme contient-il un bug ?
  $\rightarrow$ indécidable
- $P =$ le programme plante-t-il avec l'entrée x ?
  $\rightarrow$ décidable

Question : l'une propriété implique-t-elle l'autre ?

Analyse de flot de données : intuition

$P =$ le programme ne retourne que des valeurs positives

```
int foo(int x) {
  int y = 0;
  if (x > 0) {
      y = x;
  }
  if (x < 0) {
      y = -x;
  }
  return y;
}
```

Analyse de flot de données : intuition

On va “simuler” les exécutions possibles, avec des états abstraits

État = pour chaque point de programme, pour chaque variable, on a un signe $(+, 0, -)$

```
int foo(int x) { // [x -> inconnu]
  int y = 0;     // [x -> inconnu, y -> 0]
  if (x > 0) {   // [x -> +, y -> 0]
      y = x;     // [x -> +, y -> +]
  }
  if (x < 0) {   // [x -> -, y -> 0]
      y = -x;    // [x -> -, y -> +]
  }
                 // [x -> inconnu, y -> +]
  return y;      // résultat : +
}
```

Rappel : Graphe de flot de contrôle

Graphe

Un graphe (N, E) est composé de :

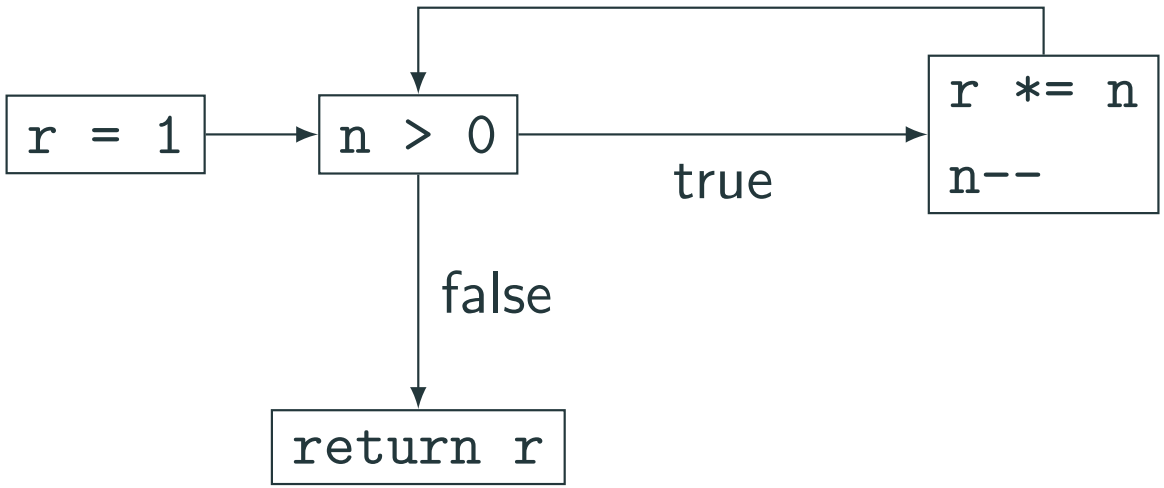
- un ensemble N de nœuds (*nodes*)
- un ensemble E d'arcs (*edges*) reliant deux nœuds : $E \subseteq N \times N$

Graphe de flot de contrôle (CFG)

Un graphe de flot de contrôle est un graphe (N, E) où :

- N représente des éléments du programme
- E représente l'ordre d'exécution potentiel

Graphe de flot de contrôle (CFG)



Treillis

Ensemble partiellement ordonné

Ensemble partiellement ordonné (*poset*)

Un ensemble partiellement ordonné $(S, \sqsubseteq)$ est un ensemble S avec une relation d'ordre partielle $\sqsubseteq$ qui est :

- refléxive : $\forall x \in S : x \sqsubseteq x$
- transitive : $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$
- antisymétrique : $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$

Par exemple :

- $(\mathbb{N}, \leq)$
- $(\mathcal{P}(S), \subseteq)$ pour tout ensemble S
- $(T, <:)$ où T sont des types de lambda-calcul typé, et $<:$ est la relation de sous-type

Ensemble partiellement ordonné (*poset*)

Note : un *poset* n'est pas *totalement* ordonné. Pour qu'il le soit, il faut une relation d'ordre totale :

$$\forall x, y \in S, x \sqsubseteq y \vee y \sqsubseteq x$$

Mais en pratique on n'utilise que des *poset*

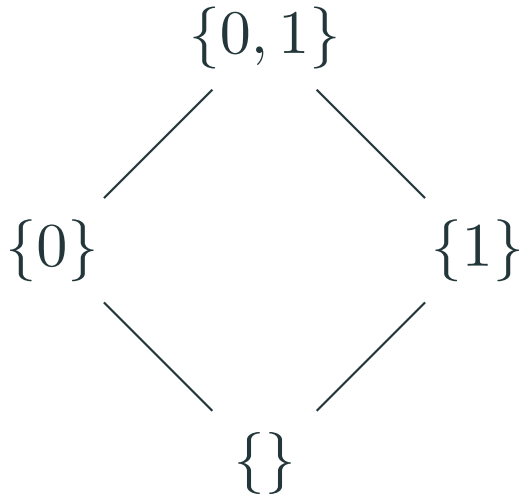
Diagramme de Hasse

On peut représenter un ensemble ordonné par un *diagramme de Hasse* :

- chaque élément est un nœud du diagramme
- l'élément le plus petit est en bas
- un arc vers le haut entre deux nœuds représente l'ordre entre ces deux éléments

Diagramme de Hasse de $(\mathbb{N}, \leq)$

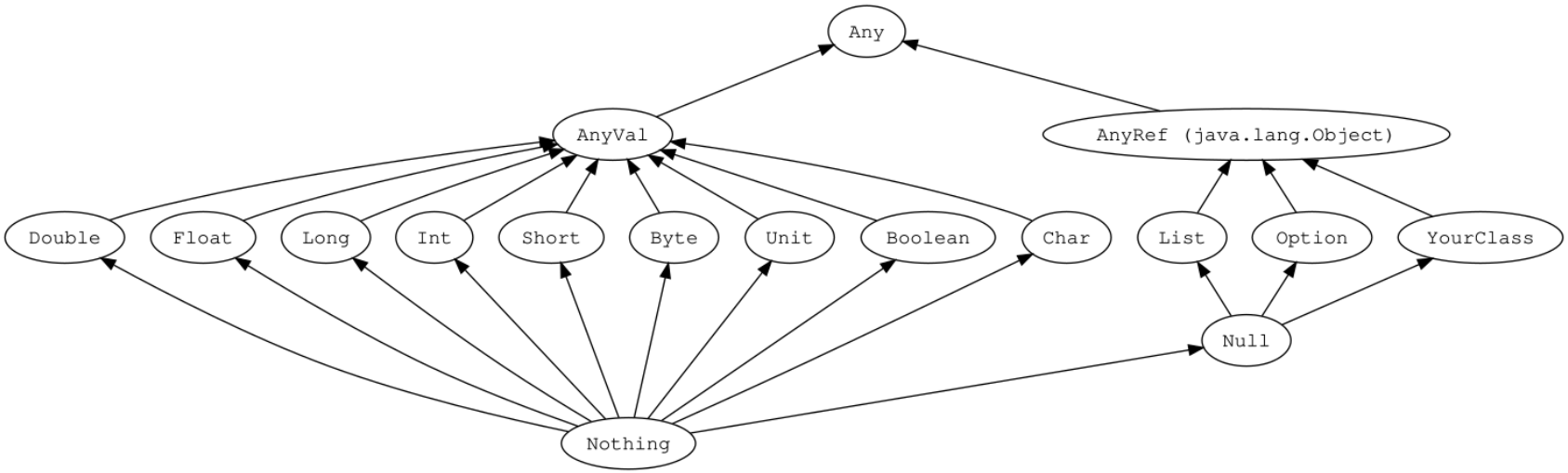
Diagramme de Hasse de $(\mathcal{P}(\{0,1\}), \subseteq)$



On voit visuellement que :

- $\{0\} \subseteq \{0,1\}$ car ils sont directement connectés
- $\{\} \subseteq \{0,1\}$ car ils sont connectés transitivement
- $\{0\} \not\subseteq \{1\}$ car il n'y a pas de chemin montant de l'un à l'autre

Diagramme de Hasse de la hiérarchie de classe de Scala



Source

Bornes

Borne supérieure la plus petite (*least upper bound, lub*)

Dans un *poset*, $x \sqcup y$ est la borne supérieure la plus petite entre x et y si :

- elle est supérieure à x : $\forall x, y \in S : x \sqsubseteq (x \sqcup y)$
- elle est supérieure à y : $\forall x, y \in S : y \sqsubseteq (x \sqcup y)$
- il n'existe pas d'autre élément plus petit qui soit supérieure à x et y : $\forall x, y, z \in S : x \sqsubseteq z \wedge y \sqsubseteq z \implies (x \sqcup y) \sqsubseteq z$

Par exemple :

- Sur $(\mathbb{N}, \leq)$:

Bornes

Borne supérieure la plus petite (*least upper bound, lub*)

Dans un *poset*, $x \sqcup y$ est la borne supérieure la plus petite entre x et y si :

- elle est supérieure à x : $\forall x, y \in S : x \sqsubseteq (x \sqcup y)$
- elle est supérieure à y : $\forall x, y \in S : y \sqsubseteq (x \sqcup y)$
- il n'existe pas d'autre élément plus petit qui soit supérieure à x et y : $\forall x, y, z \in S : x \sqsubseteq z \wedge y \sqsubseteq z \implies (x \sqcup y) \sqsubseteq z$

Par exemple :

- Sur $(\mathbb{N}, \leq)$: $\max$
- Sur $(\mathcal{P}(S), \subseteq)$:

Bornes

Borne supérieure la plus petite (*least upper bound, lub*)

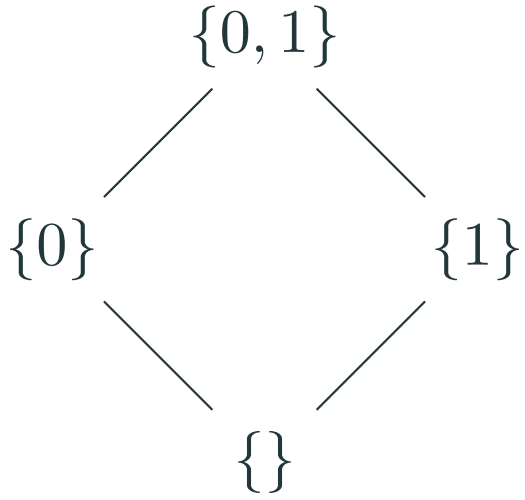
Dans un *poset*, $x \sqcup y$ est la borne supérieure la plus petite entre x et y si :

- elle est supérieure à x : $\forall x, y \in S : x \sqsubseteq (x \sqcup y)$
- elle est supérieure à y : $\forall x, y \in S : y \sqsubseteq (x \sqcup y)$
- il n'existe pas d'autre élément plus petit qui soit supérieure à x et y : $\forall x, y, z \in S : x \sqsubseteq z \wedge y \sqsubseteq z \implies (x \sqcup y) \sqsubseteq z$

Par exemple :

- Sur $(\mathbb{N}, \leq)$: $\max$
- Sur $(\mathcal{P}(S), \subseteq)$: $\cup$

Bornes



Visuellement, on prend le premier élément commun en montant :

- $\{\} \cup \{1\} = \{1\}$
- $\{0\} \cup \{1\} = \{0,1\}$

Bornes

Borne inférieure la plus grande (*greatest lower bound, glb*)

Dans un *poset*, $x \sqcap y$ est la borne supérieure la plus petite entre x et y si :

- elle est inférieure à x : $\forall x, y \in S : (x \sqcap y) \sqsubseteq x$
- elle est supérieure à y : $\forall x, y \in S : (x \sqcap y) \sqsubseteq y$
- il n'existe pas d'autre élément plus grand qui soit inférieur à x et y : $\forall x, y, z \in S : z \sqsubseteq x \land z \sqsubseteq y \implies z \sqsubseteq (x \sqcap y)$

Par exemple :

- Sur $(\mathbb{N}, \leq)$:

Bornes

Borne inférieure la plus grande (*greatest lower bound, glb*)

Dans un *poset*, $x \sqcap y$ est la borne supérieure la plus petite entre x et y si :

- elle est inférieure à x : $\forall x, y \in S : (x \sqcap y) \sqsubseteq x$
- elle est supérieure à y : $\forall x, y \in S : (x \sqcap y) \sqsubseteq y$
- il n'existe pas d'autre élément plus grand qui soit inférieur à x et y : $\forall x, y, z \in S : z \sqsubseteq x \land z \sqsubseteq y \implies z \sqsubseteq (x \sqcap y)$

Par exemple :

- Sur $(\mathbb{N}, \leq)$: $\min$
- Sur $(\mathcal{P}(S), \subseteq)$:

Bornes

Borne inférieure la plus grande (*greatest lower bound, glb*)

Dans un *poset*, $x \sqcap y$ est la borne supérieure la plus petite entre x et y si :

- elle est inférieure à x : $\forall x, y \in S : (x \sqcap y) \sqsubseteq x$
- elle est supérieure à y : $\forall x, y \in S : (x \sqcap y) \sqsubseteq y$
- il n'existe pas d'autre élément plus grand qui soit inférieur à x et y : $\forall x, y, z \in S : z \sqsubseteq x \land z \sqsubseteq y \implies z \sqsubseteq (x \sqcap y)$

Par exemple :

- Sur $(\mathbb{N}, \leq)$: $\min$
- Sur $(\mathcal{P}(S), \subseteq)$: $\cap$

Infimum (*bottom*)

Infimum

Dans un *poset* $(S, \sqsubseteq)$, l'infimum (dénoté $\bot$) est l'élément qui est plus petit que tous les autres :

$$\bot = \bigsqcup \{\} = \bigsqcap S$$

Un infimum n'existe pas toujours.

Par exemple :

- $(\mathbb{N}, \leq) : 0$
- $(\mathcal{P}(S), \subseteq) : \{\}$
- Hiérarchie de classe de Scala : `Nothing`

Supremum (*top*)

Supremum

Dans un *poset* $(S, \sqsubseteq)$, le supremum (dénoté $\top$) est l'élément qui est plus grand que tous les autres :

$$\top = \bigsqcap \{\} = \bigsqcup S$$

Un supremum n'existe pas toujours.

Treillis (*lattice*)

Treillis

Un treillis $(S, \sqsubseteq, \bot, \sqcup, \top, \sqcap)$ est un *poset* $(S, \sqsubseteq)$ avec :

- un infimum unique $\bot \in S$
- un supremum unique $\top \in S$
- pour chaque paire d'éléments $x, y \in S$, une borne supérieure $x \sqcup y$ et inférieure $x \sqcap y$

Exemples :

- $(\mathbb{N}, \leq, \max)$ n'est pas un treillis : pas de supremum
- $(\mathcal{P}(S), \subseteq, \cup)$ est un treillis
- la hiérarchie de classe de Scala forme un treillis

Treillis (*lattice*)

Treillis complet (*complete lattice*)

Un treillis complet est un treillis $(S, \sqsubseteq, \sqcup)$ si tout sous-ensemble d'éléments E a une borne supérieure $\bigsqcup E$.

Treillis (*lattice*)

Demi-treillis avec borne supérieure (*join semi-lattice*)

Un demi-treillis avec borne supérieure $(S, \sqsubseteq, \bot, \sqcup)$ est un *poset* $(S, \sqsubseteq)$ avec :

- un infimum unique $\bot \in S$
- pour chaque paire d'élément $x, y \in S$, une borne supérieure $x \sqcup y$

Demi-treillis avec borne inférieure (*meet semi-lattice*)

Un demi-treillis avec inférieure supérieure $(S, \sqsubseteq, \top, \sqcap)$ est un *poset* $(S, \sqsubseteq)$ avec :

- un supremum unique $\top \in S$
- pour chaque paire d'élément $x, y \in S$, une borne inférieure $x \sqcap y$

En pratique, on utilisera généralement des demi-treillis

Treillis : implémentation

code/Lattice.h

```
template<typename T>
struct SemiLattice {
    static T bottom();
    static bool leq(const T &x, const T &y);
    static T join(const T &x, const T &y);
};
```

Treillis communs : booléen

code/Lattice.h

Le treillis $(\{\top, \bot\}, \sqsubseteq)$ où $\bot \sqsubseteq \top$:

Parfois utilisé pour représenter des niveaux de sécurité (public/privé)

Treillis communs : booléen

code/Lattice.h

```
template<>
struct SemiLattice<bool> {
    static bool bottom() { return false; }
    static bool leq(const bool &x, const bool &y) {
        if (x) return y;
        return true;
    }
    static bool join(const bool &x, const bool &y) {
        return x || y;
    }
};
```

Pour tout ensemble S, on peut définir un treillis en ajoutant deux éléments :

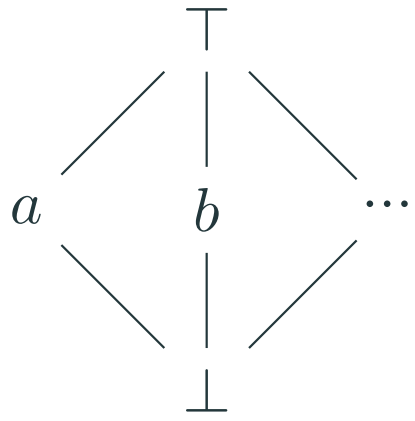
- $\top$ tel que $\forall x \in S, x \sqsubseteq \top$
- $\bot$ tel que $\forall x \in S, \bot \sqsubseteq x$

Pour tous les autres éléments : $\forall x, y \in S, x = y \iff x \sqsubseteq y$

Exemple : propagation de constante

Treillis communs : treillis plat

code/Lattice.h



Treillis communs : treillis plat

code/Lattice.h

```
template<typename T>
struct SemiLattice<FlatElement<T> *> {
    static FlatElement<T> *bottom() { return new FlatBottom<T>(); }
    static bool leq(const FlatElement<T> *x, const FlatElement<T> *y) {
        return x->height() <= y->height();
    }
    static FlatElement<T> *join(const FlatElement<T> *x, const FlatElem
        if (x->height() == FlatLatticeBottom ||
            y->height() == FlatLatticeTop)
            return const_cast<FlatElement<T> *>(y);
        if (y->height() == FlatLatticeBottom ||
            x->height() == FlatLatticeTop)
            return const_cast<FlatElement<T> *>(x);
        if (*x == *y)
            return const_cast<FlatElement<T> *>(x);
        return new FlatTop<T>();
    }
};
```

Treillis communs : ensemble des parties

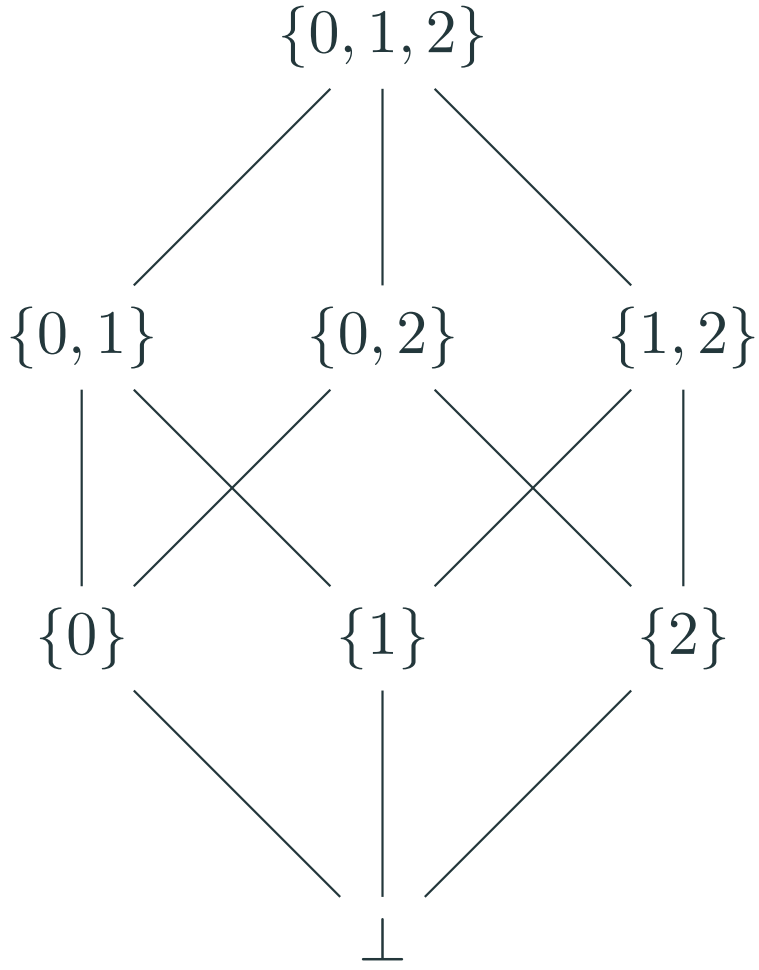
 `code/Lattice.h`

Pour tout ensemble S, on peut définir un treillis de l'ensemble des parties de S (*powerset lattice*) $(\mathcal{S}, \subseteq)$, avec :

- $\bot = \{\}$
- $\top = S$
- $\sqsubseteq = \subseteq$
- $x \sqcup y = x \cup y$
- $x \sqcap y = x \cap y$

Treillis communs : ensemble des parties `code/Lattice.h`

Avec $S = \{0, 1, 2\}$



Treillis communs : ensemble des parties

code/Lattice.h

```
template<typename  T>
struct SemiLattice<std::set<T>> {
   static std::set<T> bottom() { return std::set<T>(); }
   static bool leq(const std::set<T> x, const std::set<T> y) {
       return std::includes(x.begin(), x.end(),
                            y.begin(), y.end());
   }
   static std::set<T> join(const std::set<T> x, const std::set<T> y) {
       std::set<T> r = std::set<T>();
       r.insert(x.begin(), x.end());
       r.insert(y.begin(), y.end());
       return r;
   }
};
```

Treillis communs : produit

 code/Lattice.h

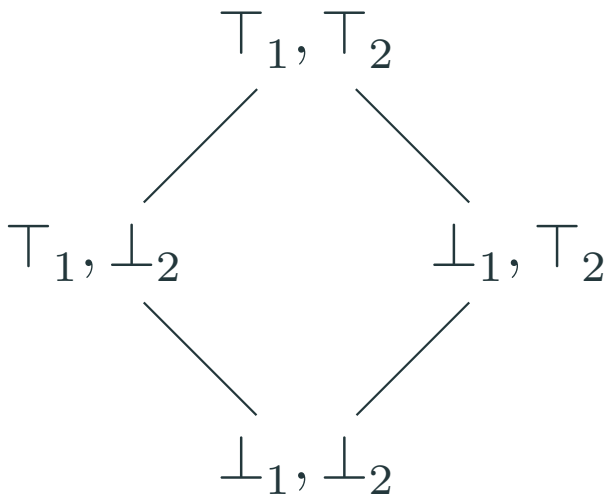
À partir de deux treillis $(S_1, \sqsubseteq_1)$ et $(S_2, \sqsubset_2)$, on peut définir un treillis de leur produit $(S_1 \times S_2, \sqsubseteq)$:

- $\bot = (\bot_1, \bot_2)$
- $\top = (\top_1, \top_2)$
- $(x, y) \sqsubseteq (x', y') \iff x \sqsubseteq_1 x' \wedge y \sqsubseteq_2 y'$
- $(x, y) \sqcup (x', y') = (x \sqcup_1 x', y \sqcup_2 y')$

Treillis communs : produit

Exemple : dessiner le treillis produit des deux treillis suivants :

Treillis communs : produit



Note : isomorphisme avec le treillis de $\mathcal{P}(\{a, b\})$ ($\top_1$ indique la présence de a, $\top_2$ indique la présence de b).

Treillis communs : produit

Exemple : inférence de type

$$Bool ::= \bot_{Bool} \mid \top_{Bool}$$
$$Num ::= \bot_{Num} \mid \top_{Num}$$
$$Str ::= \bot_{Str} \mid \top_{Str}$$
$$Val = Bool \times Num \times Str$$

```
foo(x) {
  return x;
}
foo(42);
foo("hello");
```

`x` a pour valeur $(\bot_{Bool}, \top_{Num}, \top_{Str})$

Treillis communs : produit

code/Lattice.h

```
template<typename T1, typename T2>
struct SemiLattice<std::pair<T1, T2>> {
  static std::pair<T1, T2> bottom() {
    return std::pair<T1, T2>(SemiLattice<T1>::bottom(),
                             SemiLattice<T2>::bottom());
  }
    static bool leq(const std::pair<T1, T2> x, const std::pair<T1, T2>
        return SemiLattice<T1>::leq(x.first, y.first) &&
            SemiLattice<T2>::leq(x.second, y.second);
    }
    static std::pair<T1, T2> join(const std::pair<T1, T2> x, const std:
        return new std::pair<T1, T2>(
            SemiLattice<T1>::join(x.first, y.first),
            SemiLattice<T2>::join(x.second, y.second));
    }
};
```

À partir d'un ensemble S et d'un treillis $(L, \sqsubseteq_L)$, on peut définir un treillis de leur association $(S \to L, \sqsubseteq)$, i.e., des fonctions $S \to L$:

- $\bot = \lambda s. \bot_L$
- $\top = \lambda s. \top_L$
- $x \sqsubseteq y \iff \forall s \in S : x(s) \sqsubseteq_L y(s)$
- $x \sqcup y = \lambda s. x(s) \sqcup_L y(s)$

Treillis associatif : exemple

Pour représenter la mémoire d'un programme :

$$\sigma \in \mathit{Store} = \mathit{Var} \to \mathit{Val}$$

Si on a un treillis pour *Val* :

$$\begin{aligned}
\bot_{\mathit{Store}} &= \lambda v.\bot_{\mathit{Val}} \\
\top_{\mathit{Store}} &= \lambda v.\top_{\mathit{Val}} \\
\sigma_1 \sqsubseteq \sigma_2 &\iff \forall v \in \mathit{Var} : \sigma_1(v) \sqsubseteq_{\mathit{Val}} \sigma_2(v) \\
\sigma_1 \sqcup \sigma_2 &= \lambda v.\sigma_1(v) \sqcup_{\mathit{Val}} \sigma_2(v)
\end{aligned}$$

Treillis associatif : exemple

Par exemple, avec :

$$\sigma_1 = [\mathrm{x} \mapsto +, \mathrm{y} \mapsto -]$$
$$\sigma_2 = [\mathrm{x} \mapsto +, \mathrm{y} \mapsto \bot]$$
$$\sigma_3 = [\mathrm{x} \mapsto -, \mathrm{y} \mapsto \bot]$$

On a :

$$\sigma_1 \sqcup \sigma_2 = \sigma_1$$
$$\sigma_2 \sqsubseteq \sigma_1$$
$$\sigma_1 \sqcup \sigma_3 = [\mathrm{x} \mapsto \top, \mathrm{y} \mapsto -]$$
$$\sigma_1 \not\sqsubseteq \sigma_3$$
$$\sigma_3 \not\sqsubseteq \sigma_1$$

Treillis communs : treillis associatif

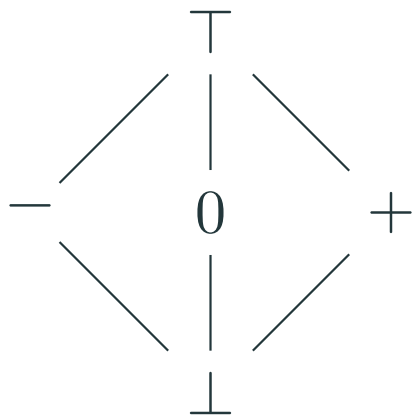
code/Lattice.h

```
template<typename T1, typename T2>
struct SemiLattice<std::map<T1, T2>> {
    static std::map<T1, T2> bottom() {
        return std::map<T1, T2>();
    }
    static bool leq(const std::map<T1, T2> x, const std::pair<T1, T2> y
        for (auto iter = x.begin(); iter != x.end(); iter++) {
            if (y.count(iter->first) == 0) return false;
            if (SemiLattice<T2>::leq(iter->second, y[iter->first]) == f
                return false;
            }
        }
        return true;
    }
    static const std::map<T1, T2> join(const std::map<T1, T2> x, const
        std::map<T1, T2> result(x);
        for (auto iter = y.begin(); iter != y.end(); iter++) {
            if (result.count(iter->first) == 0) {
```

Treillis de signe

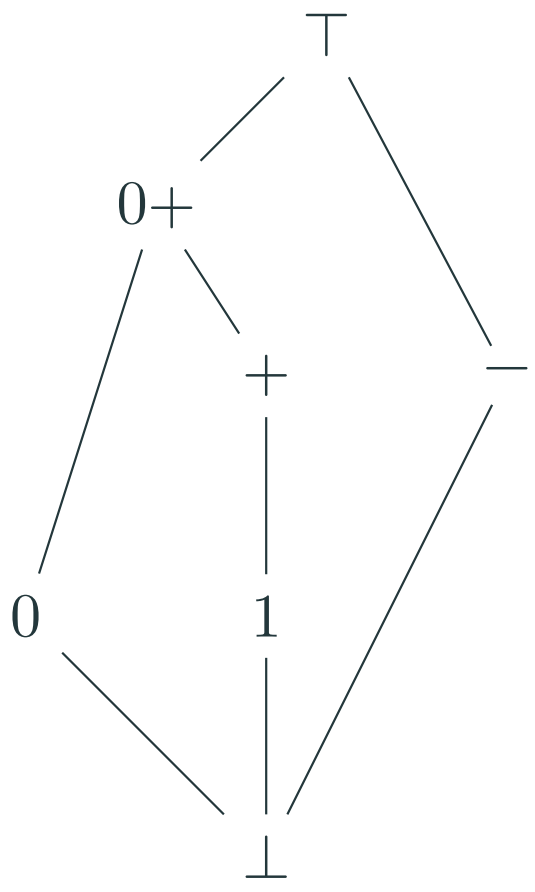
On souhaite définir un treillis capable de représenter le signe d'un nombre.

Version basique : treillis plat basé sur l'ensemble $\{-, 0, +\}$



Treillis de signe

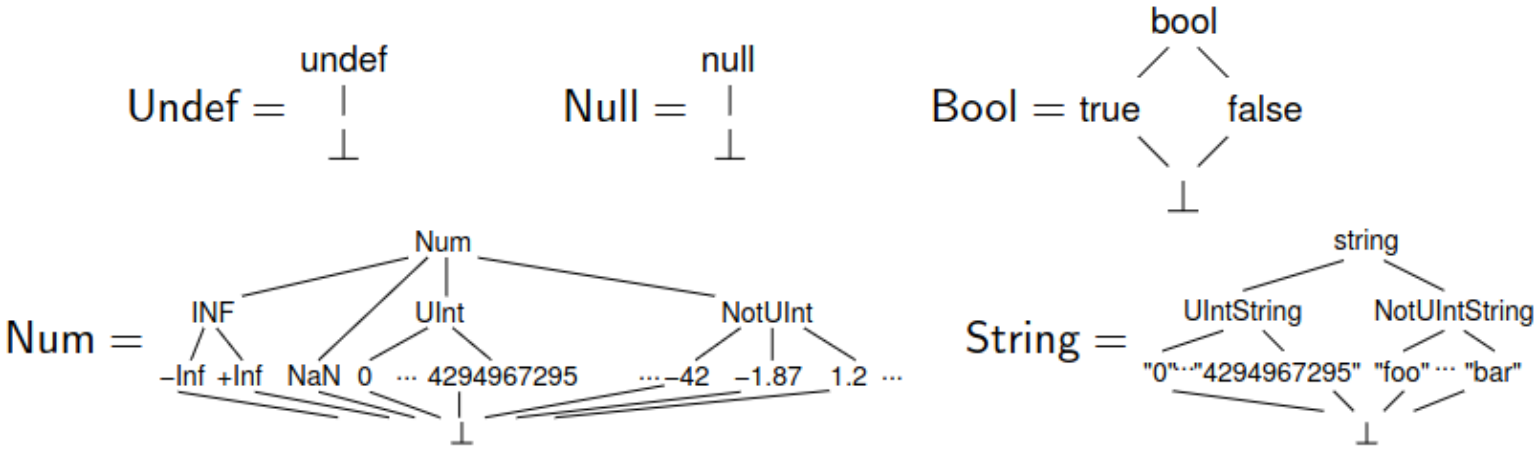
Version avancée (plus de précision)



Abstract values are described by the lattice Value:

$$\mathsf{Value} = \mathsf{Undef} \times \mathsf{Null} \times \mathsf{Bool} \times \mathsf{Num} \times \mathsf{String} \times \mathcal{P}(L)$$

The components of Value describe the different types of values.



Jensen, Møller & Thiemann. (2009). Type Analysis for JavaScript. SAS.

Fonction sur un treillis

Soit le treillis de signe. On souhaite définir une fonction $\hat{+}$ qui représente une addition dans le treillis :

$$
\begin{aligned}
+ \mathbin{\hat{+}} + &= + \\
+ \mathbin{\hat{+}} 0 &= + \\
+ \mathbin{\hat{+}} - &= \top \\
0 \mathbin{\hat{+}} x &= x \\
- \mathbin{\hat{+}} 0 &= - \\
- \mathbin{\hat{+}} + &= \top
\end{aligned}
$$

On verra plus tard comment s'assurer qu'une telle définition est correcte.

Monotonie

Fonction monotone

Soit deux *poset* $(L_1, \sqsubseteq_1)$ et $(L_2, \sqsubseteq_2)$ et une fonction $f : L_1 \to L_2$. f est monotone ssi :

$$\forall x, y \in L_1 \ : x \sqsubseteq_1 y \implies f(x) \sqsubseteq_2 f(y)$$

En particulier, si $L_1 = L_2$, on parle de **monotonie interne** avec $f : L \to L$ monotone ssi :

$$\forall x, y \in L \ : x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$$

Intuitivement, une fonction monotone *préserve la relation d'ordre*

Monotonie

Pour le treillis $(\mathcal{P}(\{0,1,2\}, \subseteq)$, lesquelles des fonctions suivantes sont monotones ?

- $\lambda x.x$
- $\lambda x.\{0\}$
- $\lambda x.x \cup \{0\}$
- $\lambda x.x \cap \{0,1\}$
- $\lambda x.\top - x$

Point fixe

Point fixe

Pour une fonction $f : A \to A$, $x \in A$ est un point fixe ssi :

$$f(x) = x$$

Note : il peut exister plusieurs points fixes.

Exercice : calculer le point fixe des fonctions monotones précédentes.

Théorème du point fixe de Kleene

Théorème du point fixe de Kleene

Pour un treillis $(L, \sqsubseteq)$ et une fonction monotone $f : L \to L$, le plus petit point fixe de f est le supremum de la chaîne :

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots \sqsubseteq f^n(\bot) \sqsubseteq \cdots$$

C'est-à-dire : pour trouver le plus petit point fixe d'une fonction, on peut itérer à partir de $\bot$

Exercice : calculer le point fixe des fonctions précédentes avec cette méthode.

Tarski. (1955). A Lattice-Theoretical Fixpoint Theorem and its Applications. Pacific Journal of Mathematic.

Trouver un point fixe : méthode naïve

Fixpoint.h

```
template<typename T>
struct NaiveFixpointSolver {
    virtual T fun(const T &x) = 0;
    T analyze(P p) {
        T cur = SemiLattice<T>::bottom();
        T prev = cur;
        do {
            prev = cur;
            cur = fun(cur);
        } while (prev != cur);
        return cur;
    }
};
```

En pratique, on va utiliser des algorithmes plus efficaces

Trouver un point fixe : méthode naïve

Fixpoint.cpp

```
struct Solver : NaiveFixpointSolver<std::set<int> > {
    std::set<int> fun(const std::set<int> &x) {
        std::set y(x);
        y.insert(0);
        return y;
    }
};
```

Framework Monotone

Résumé de la théorie

On a :

- des treillis
- des fonctions monotones sur des treillis
- la possibilité de calculer le point fixe d'une fonction monotone sur un treillis

On peut combiner le tout pour calculer différentes analyses.

Par exemple : une analyse de signes

Treillis : exemple de l'analyse de signe

- Treillis de base : notre treillis de signe (treillis plat)

$$L_0 = \{\bot, -, 0, +, \top\}$$

- Chaque variable peut avoir sa valeur (treillis associatif)

$$L_\sigma = \mathit{Var} \to L_0$$

- Chaque point de programme peut avoir sa valeur (treillis associatif)

$$L = N \to L_\sigma$$

où N représente les nœuds du flot de contrôle du programme (les *basic blocks* par exemple)

Treillis : exemple de l'analyse de signe

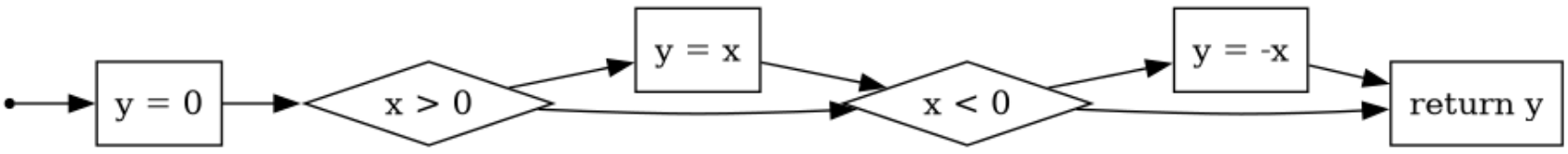
```
int foo(int x) { // [x -> inconnu, y -> inconnu]
  int y = 0;     // [x -> inconnu, y -> 0]
  if (x > 0) {   // [x -> +, y -> 0]
      y = x;     // [x -> +, y -> +]
  }
  if (x < 0) {   // [x -> -, y -> 0]
      y = -x;    // [x -> -, y -> +]
  }
                 // [x -> inconnu, y -> +]
  return y;      // résultat : +
}
```

Notre treillis est : $N \to \{\mathrm{x}, \mathrm{y}\} \to \{\top, \bot, +, 0, -\}\}$

Treillis : exemple de l'analyse de signe

Notre treillis est : $N \to \{\mathtt{x}, \mathtt{y}\} \to \{\top, \bot, +, 0, -\}\}$

Où N représente les nœuds du CFG :



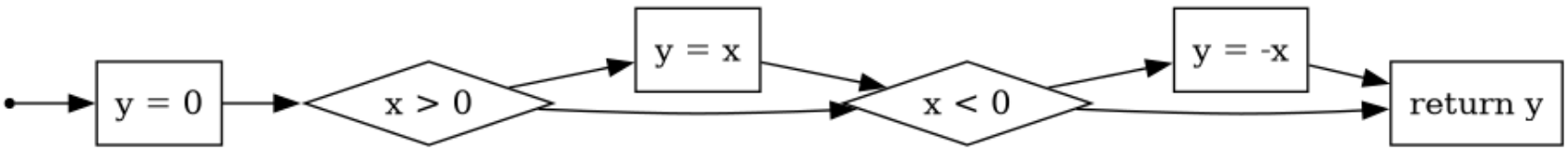
Fonction d'analyse

On veut une fonction $f : L \to L$ qui :

- pour une ancienne valeur sur le treillis
  - les valeurs connues de nos variables
- donne une nouvelle valeur (plus grande ou égale)
  - qui correspond à une "propagation" de l'information accumulée

Fonction d'analyse : exemple

Exemple : en commençant avec $\bot_\sigma$ sur chaque nœud, on applique f



Fonction monotone

La fonction monotone sur notre treillis va permettre de calculer l'analyse.

$$f : L \to L$$
$$f = \lambda s. \bigsqcup_{n \in N} [n \mapsto [\![n]\!](s)]$$

Où on définit $[\![n]\!]$ au cas par cas pour chaque nœud, pour chaque analyse.

$$[\![\cdot]\!] : N \to L \to L$$

Fonction de jonction

On définit une fonction auxiliaire :

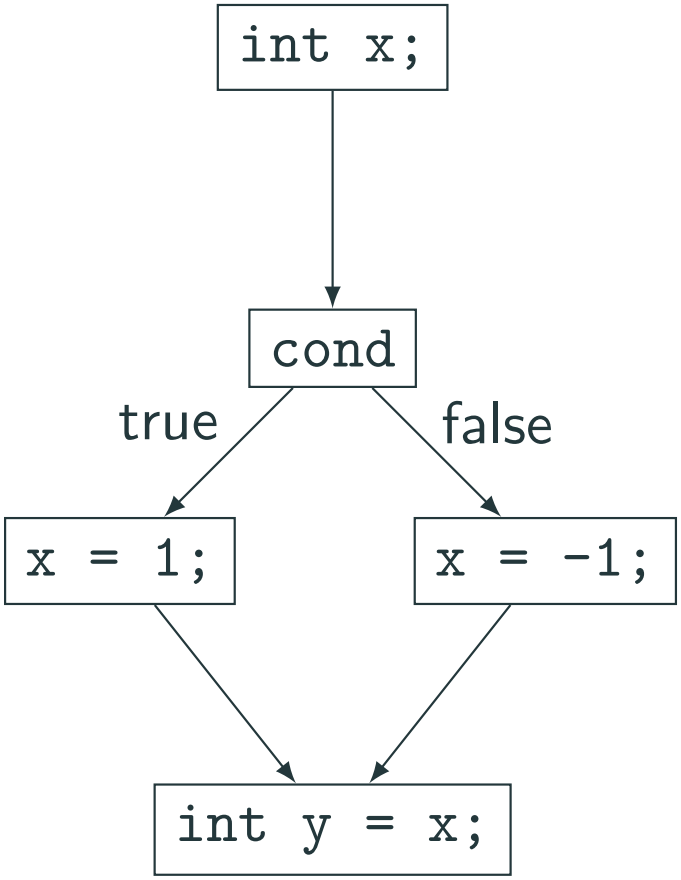
$$\textsc{join} : N \to L \to L_\sigma$$
$$\textsc{join} = \lambda n.\lambda s. \bigsqcup_{p \in \mathsf{pred}(n)} [\![p]\!](s)$$

Fonction de jonction

Utile pour quand on a une jonction dans le CFG, par exemple :

```
int x;
if (cond) {
  x = 1;
} else {
  x = -1;
}
int y = x;
```

Fonction de jonction



Fonction monotone pour l'analyse de signe

Définition au cas par cas, par exemple :

- Si n est l'entrée du CFG :

$$[\![n]\!] = \lambda s.\bot$$

- Si n est $\mathtt{x} = e$:

$$[\![n]\!] = \sigma[\mathtt{x} \mapsto \mathit{eval}(a, \sigma)]$$
$$\text{où } \sigma = \textsc{join}(n)(s)$$

- Sinon,

$$[\![n]\!] = \lambda s.\textsc{join}(n)(s)$$

Où

$$\textsc{join} = \lambda n.\lambda s. \bigsqcup_{p \in \mathsf{pred}(n)} [\![p]\!](s)$$

Fonction d'évaluation

$$\begin{aligned} \mathit{eval} : \mathit{Expr} &\to L_\sigma \to L_0 \\ \mathit{eval}(x + y, \sigma) &= \mathit{eval}(x) \mathbin{\hat{+}} \mathit{eval}(y) \\ \mathit{eval}(x - y, \sigma) &= \mathit{eval}(x) \mathbin{\hat{-}} \mathit{eval}(y) \\ &\ldots \end{aligned}$$

Où $\hat{+}$, $\hat{-}$, etc. sont définies sur le treillis de base.

Fonction sur notre treillis

On peut définir $\hat{+}$ comme suit :

$\hat{+}$	$\bot$	0	$-$	$+$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	$-$	$+$	$\top$
$-$	$\bot$	$-$	$-$	$\top$	$\top$
$+$	$\bot$	$+$	$\top$	$+$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$

Monotonie de $\hat{+}$

$\hat{+}$ est monotone ssi :

$$x \sqsubseteq y \wedge x' \sqsubseteq y' \implies x \hat{+} y \sqsubseteq x' \hat{+} y'$$

On peut le vérifier pour tous les cas. Par exemple, pour $x = +$, $y = -$:

- x' peut valoir $+$ ou $\top$
- y' peut valoir $-$ ou $\top$

On a donc les cas suivants :

- $+ \hat{+} - \sqsubseteq + \hat{+} -$: trivial
- $+ \hat{+} - \sqsubseteq + \hat{+} \top$: on obtient $\top$ des deux côtés
- $+ \hat{+} - \sqsubseteq \top \hat{+} -$: on obtient $\top$ des deux côtés
- $+ \hat{+} - \sqsubseteq \top \hat{+} \top$: on obtient $\top$ des deux côtés

Définition sûre de $\hat{+}$

$\hat{+}$ est correct ssi :

$$\forall x, y \in \mathit{Val} : \alpha(x + y) \sqsubseteq \alpha(x) \hat{+} \alpha(y)$$

où $\alpha : \mathit{Val} \to L_0$

$$\alpha(1 + 1) = \alpha(2) = + \sqsubseteq \alpha(1) \hat{+} \alpha(1) = + \hat{+} + = +$$
$$\alpha(1 + 0) = \alpha(1) = + \sqsubseteq \alpha(1) \hat{+} \alpha(0) = + \hat{+} 0 = +$$
$$\alpha(1 + -1) = \alpha(0) = 0 \sqsubseteq \alpha(1) \hat{+} \alpha(-1) = + \hat{+} - = \top$$
$$\ldots$$

Plus de détails sur cette théorie dans le chapitre 6

Fonction de transfert

En pratique, on a souvent

$$[\![n]\!] = \lambda s.t_n(\textsc{join}(n)(s))$$

On appelle t une **fonction de transfert**. On peut alors réécrire notre analyse :

- Si n est l'entrée du CFG :

$$t_n(\sigma) = \bot$$

- Si n est $\mathtt{x} = e$:

$$t_n(\sigma) = \sigma[\mathtt{x} \mapsto \mathit{eval}(a, \sigma)]$$

- Sinon,

$$t_n(\sigma) = \sigma$$

Calcul de l'analyse

$$f : L \to L$$
$$f = \lambda s. \bigsqcup_{n \in N} [n \mapsto [\![n]\!](x)]$$

On peut donc calculer le point fixe de f et on aura le résultat de l'analyse.

Système d'équation

On peut écrire f sous forme d'un système d'équation :

$$
\begin{aligned}
f_1(s) &= [\![n_1]\!](s) \\
&\dots \\
f_i(s) &= [\![n_i]\!](s) \\
&\dots \\
f_n(s) &= [\![n_n]\!](s)
\end{aligned}
$$

On peut donc calculer l'analyse nœud par nœud.

Résolution : tourniquet (*round-robin*) `code/Fixpoint.h`

Classe `RoundRobinSolver` :

- `StateLattice fun(Lattice x, Node node)`
  - la fonction dont on veut calculer le point fixe
  - définie pour un nœud particulier
  - `Lattice` est notre L_σ
- `allNodes(P p)`
  - donne N, l'ensemble des nœuds du programme p à analyser
- `Lattice analyze(P p)`
  - le calcul de point fixe
  - définit pour un programme `p`
  - tant qu'il y a eu un changement, pour tout $n \in N$
    - on recalcule `fun(res, n)` en chaque nœud
    - on détecte s'il y a un changement

Résolution : liste de travail (*worklist*) `code/Fixpoint.h`

Classe `WorklistFixpointSolver` :

- `StateLattice fun(Lattice x, Node node)`
  - comme avant
- `Lattice analyze(P p)`
  - on commence avec `res` $= \bot$ et $W = N$ (tous les nœuds)
  - tant qu'il y a un élément $n \in W$:
    - on exécute `fun(res, n)`
    - si le résultat est différent du résultat précédent
      on ajoute tous les successeurs de n à W
    - on met à jour `res[node]` avec le nouveau résultat

Résolution : liste de travail et fonction de transfert

`code/Fixpoint.h`

Classe `TransferWorklistFixpointSolver` :

- basée sur `WorklistFixpointSolver`
- `join` est la fonction $\textsc{Join}$:
  - prend tous les prédécesseurs et les joint
- `fun` est la fonction F
  - applique la fonction de transfert à $\textsc{Join}(n, s)$

Gestion des boucles

Les branchements sont gérés par le framework monotone : c'est ce que fait JOIN.

Les boucles ne sont que des formes de branchements :

```
while (cond) {
   body;
}
```

est équivalent à :

```
if (cond) {
   body;
   if (cond) {
     body;
     ...
   }
```

Terminaison avec treillis fini

Est-ce que notre analyse va terminer ?

Cas facile :

- On a un treillis d'une taille finie
- On a une fonction monotone :
  - Soit on monte dans le treillis
  - Soit on a atteint un point fixe
  - On ne peut pas descendre

Avec un treillis fini, on a donc une garantie de terminaison

Terminaison avec treillis infini

Cas facile de treillis infini :

- Condition de chaîne ascendante : il n'existe pas de chaîne infinie $x_0 \subseteq x_1 \subseteq ...$
- Même raisonnement : on ne peut que monter ou stagner

Condition de chaîne ascendante

Un ensemble partiellement ordonné $(L, \sqsubseteq)$ satisfait à la condition de chaîne ascendante ssi L ne c ontient pas de suite infinie strictement croissante. C'est-à-dire, toute suite $l_1 \sqsubset l_2 \sqsubset ...$ se stabilise.

Terminaison avec treillis à hauteur infinie

Par exemple : treillis d'intervalles

$$L = \{[l, h] | l, h \in \mathbb{Z}\}$$
$$[l, h] \sqsubseteq [l', h'] \iff l' \leq l \wedge h \leq h'$$

Il existe des méthodes pour garantir la terminaison dans ces cas.

Analyses classiques

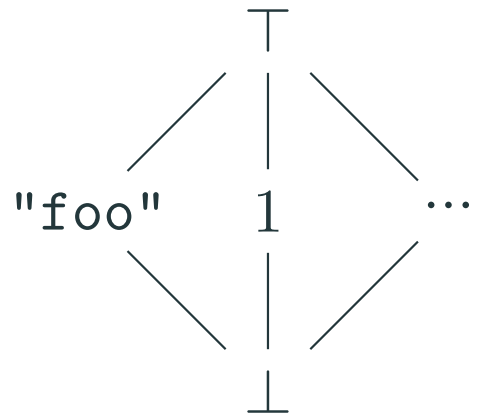
Analyse de propagation de constante

On souhaite savoir quels variables restent constantes

- Utile pour l'optimisation
- En pratique, une forme avancée fait propagation de constante et suppression de code mort (*sparse conditional constant propagation*, SCCP)

Analyse de propagation de constante : treillis

L_0 sera un treillis plat basé sur les valeurs du langage (*Val*)



$$L_\sigma = \mathit{Var} \to L_0$$

Analyse de propagation de constante : exemple

```
int x = 14;                    // [x -> 14, y -> bot, z -> bot
int y = 7 - x / 2;             // [x -> 14, y -> 0, z -> bot]
int z;                         // [x -> 14, y -> 0, z -> bot]
if (y > 0) {
  z = 1;                       // [x -> 14, y -> 0, z -> 1]
} else {
  z = -1;                      // [x -> 14, y -> 0, z -> -1]
}
                               // [x -> 14, y -> 0, z -> top]
```

Analyse de propagation de constante : équations

- Si n est l'entrée :

$$[\![n]\!] = \lambda s.\lambda v \bot$$

- Si n est une assignation $x = a$:

$$[\![n]\!] = \lambda s.\textsc{join}(n)(s)[\mathtt{x} \mapsto a]$$

- Sinon :

$$[\![n]\!] = \lambda s.\textsc{join}(n)(s)$$

Analyse de définition atteignante (*reaching definition*)

On souhaite savoir quels assignations ont donné la valeur courante d'une variable.

- Utile pour des analyses de dépendances (slicing, taint), cf. Chapitre 5
- Utilisé dans plusieurs passes d'optimisation (propagation de constante, élimination de sous-expression communes)

Définition atteignante

Les **définition atteignantes** un point de programme p sont les assignations $v = a$ qui ont *possiblement* été exécutée dans un chemin entre le point d'entrée et le point p et qui peuvent donner les valeurs des variables courantes.

Analyse de définition atteignante

code/reachingdef.c

```c
int main(int argc, char *argv[]) {
    int x, y, z;
    x = atoi(argv[1]);
    while (x > 1) {
        y = x / 2;
        if (y > 3) x = x - y;
        z = x - 4;
        if (z > 0) x = x / 2;
        z = z - 1;
    }
    printf("%d\n", x);
}
```

Analyse de définition atteignante

code/reachingdef.c

```
int main(int argc, char *argv) {
    int x, y, z; // {}
    x = atoi(argv[1]); // {x = atoi(argv[1])}
    while (x > 1) {
        // {x = atoi(argv[1]), y = x / 2, x = x - y, x = x / 2, z = z - 1}
        y = x / 2;
        // {x = atoi(argv[1]), y = x / 2, x = x - y, x = x / 2, z = z - 1}
        if (y > 3) x = x - y; // {y = x / 2, x = x - y, z = z - 1}
        z = x - 4;
        // {x = atoi(argv[1]), y = x / 2, x = x - y, x = x / 2, z = x - 4}
        if (z > 0) x = x / 2; // {y = x / 2, x = x / 2, z = x - 4}
        z = z - 1;
        // {x = atoi(argv[1]), y = x / 2, x = x - y, x = x / 2, z = z - 1}
    }
    printf("%d\n", x);
    // {x = atoi(argv[1]), y = x / 2, x = x - y, x = x / 2, z = z - 1}
}
```

Analyse de définition atteignante : treillis

Pour chaque point de programme, on veut un ensemble de définitions atteignantes, donc :

$$(L_\sigma, \sqsubseteq) = (\mathcal{P}(Defs), \subseteq)$$

où *Defs* est l'ensemble des définitions qui apparaissent dans le programme.

Analyse de définition atteignante : équations

- Si n est l'entrée :

$$[\![n]\!] = \lambda s.\{\}$$

- Si n est une assignation $x = a$:

$$[\![n]\!] = \lambda s.\{x' = a' \in \textsc{join}(n)(s) \text{ si } x \neq x'\} \cup \{x = a\}$$

- Sinon :

$$[\![n]\!] = \lambda s.\textsc{join}(n)(s)$$

Analyse de variable vivante (*live variables*)

On souhaite savoir quelles variables vont *peut-être* être utilisées dans le futur de l'exécution du programme.

On parle ici du **futur** de l'exécution, on va donc commencer l'analyse à la *fin* du programme.

Utile dans les compilateurs :

- on peut éliminer les variables non vivantes
- plusieurs variables vivantes à des moments différents peuvent partager le même emplacement

Variable vivante

Une variable v est *vivante* pour un point de programme p si v peut être utilisée sur un chemin d'exécution à partir de p, avant que v soit écrasée.

Analyse de variable vivante : exemple

code/livevars.c

```
int main(int argc, char *argv) {
    int x, y, z;
    x = atoi(argv[1]);
    while (x > 1) {
        y = x / 2;
        if (y > 3) x = x - y;
        z = x - 4;
        if (z > 0) x = x / 2;
        z = z - 1;
    }
    printf("%d\n", x);
}
```

Analyse de variable vivante : exemple

code/livevars.c

```
int main(int argc, char *argv) {
    /* {}        */ int x, y, z;
    /* {}        */ x = atoi(argv[1]);
    /* {x}       */ while (x > 1) {
    /* {x}       */     y = x / 2;
    /* {x, y} */     if (y > 3) x = x - y;
    /* {x}       */     z = x - 4;
    /* {x, z}  */     if (z > 0) x = x / 2;
    /* {x}       */     z = z - 1;
                    }
    /* {x}       */ printf("%d\n", x);
}
```

Analyse de variable vivante : treillis

On veut savoir pour chaque point de programme, quelles variables sont vivantes.

$$(L, \sqsubseteq) = (\mathcal{P}(\mathit{Vars}), \subseteq)$$

Analyse de variable vivante : équations

- Si n est la sortie :

$$[\![n]\!] = \lambda s.\{\}$$

- Si n est une assignation `x` $=a$:

$$[\![n]\!] = \lambda s.(\textsc{join}(n)(s) \backslash \{x\}) \cup \mathit{vars}(a)$$

- Sinon, si n contient une expression a :

$$[\![n]\!] = \lambda s.\textsc{join}(n)(s) \cup \mathit{vars}(a)$$

Où :

- $\mathit{vars}(a)$ est l'ensemble des variables utilisées par une expression
- $\textsc{join}(n) = \lambda n.\lambda s. \bigsqcup_{n' \in \mathsf{succ}(n)} [\![n']\!](s)$ car on procède en sens inverse

Analyse d'expressions disponibles (*available expressions*)

On souhaite savoir quelles expressions ont déjà été évaluées et sont toujours valides.

- Utile pour supprimer des calculs redondants

Expression disponible

Une expression e est **disponible** à un point de programme p si e a été calculé *sur tous les chemins* menant à p, et sa valeur n'a pas changé depuis qu'elle a été calculée.

Analyse d'expression disponible

code/availableexp.c

```
int foo(int a, int b) {
    int x, y, z;
    z = a + b;
    y = a * b;
    while (y > a + b) {
        a = a + 1;
        x = a + b;
    }
    return a;
}
```

Analyse d'expression disponible

code/availableexp.c

```
int foo(int a, int b) {
    int x, y, z;            // {}
    z = a + b;              // {a+b}
    y = a * b;              // {a+b, a*b}
    while (y > a + b) {     // {a+b, y > a+b}
        a = a + 1;          // {a+1}
        x = a + b;          // {a+1, a+b}
    }
    return a;               // {a+b, y > a+b}
}
```

Analyse d'expression disponible : treillis

On veut un ensemble d'expression, donc :

$$(L, \sqsubseteq) = (\mathcal{P}(\mathit{Exp}), \subseteq)$$

où *Exp* est l'ensemble des expressions qui apparaissent dans le programme.

Mais, on veut des **garanties** (*must*) : on ne va donc pas utiliser $\cup$ mais $\cap$ pour $\sqcup$

Analyse d'expression disponible : équations

- Si n est l'entrée :

$$[\![n]\!] = \lambda s.\{\}$$

- Si n est une assignation $x = a$:

$$[\![n]\!] = \lambda s.(\textsc{join}(n)(s) \cup \mathit{exprs}(a)) \setminus \mathit{exprsWith}(x)$$

- Si n contient une expression a :

$$[\![n]\!] = \lambda s.\textsc{join}(n)(s) \cup \mathit{exprs}(a)$$

- Sinon :

$$[\![n]\!] = \lambda s.\textsc{join}(n)(s)$$

Où :

- $\textsc{join} = \lambda n.\lambda s. \bigsqcup_{p \in \mathsf{pred}(n)} [\![p]\!](s)$
- $\mathit{exprs}(a)$ est l'ensemble des expressions contenues dans a
- $\mathit{exprsWith}(x)$ est l'ensemble des expressions qui contiennent la variable x

Catégorisation d'analyses dataflow

	Forward	*Backward*
May	Définition atteignante	Variable vivante
Must	Expression disponible	Expression fort occupée

Analyse d'expression *fort occupées* (*very busy expression*)

Expression fort occupée

Une expression est *fort occupée* ssi il est garantit que cette expression va être évaluée dans le futur avant que sa valeur ne change.

- On parle de *garantie* : c'est une analyse *must*
- On parle du futur : c'est une analyse *backwards*

Éléments avancés

Cadre à vecteur de bits (*bitvector Framework*)

Plusieurs analyses classiques peuvent être exprimées par :

- Un treillis $(L, \subseteq)$ où L est un ensemble fini
- Une fonction

$$[\![n]\!] = \lambda s.(\textsc{join}(n)(s) \setminus \mathit{kill}(n)) \cup \mathit{gen}(n)$$

où $\mathit{gen}(n)$ et $\mathit{kill}(n)$ ne dépendent **que** de n

On peut les implémenter **efficacement** :

- $l \in L$ est représenté par un vecteur de bits de taille prédéfinie
- $\mathit{gen}(n)$ et $\mathit{kill}(n)$ peuvent être précalculée en une passe linéaire sur le programme
- Le calcul du point fixe se fait avec des opérations bit-à-bit

Analyse interprocédurale

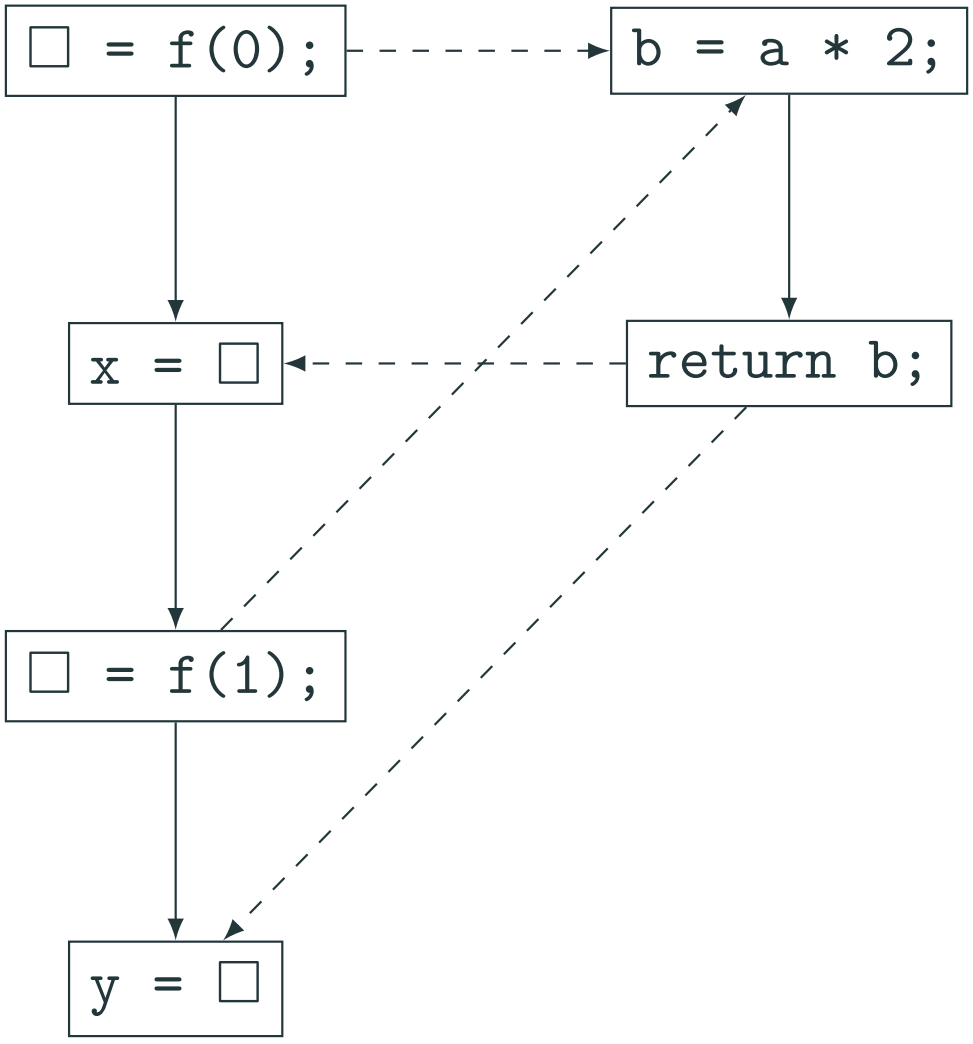
Soit le programme :

```
int f(a) {
  int b = a * 2;
  return b;
}

int main() {
  int x = f(0);
  int y = f(1);
  return x + y;
}
```

Pour l'analyser, il faut une analyse **interprocédurale** qui prend en compte les appels de fonctions

Graphe de flot de contrôle interprocédural



Analyse interprocédurale : manque de sensitivité

On analyse donc deux fois `f`, avec une analyse de signe :

- première itération : $\mathtt{a} = 0$
- seconde itération : $\mathtt{a} = 0 \sqcup +\top$

Donc, on trouve que la valeur finale est $\top$!

Il y a des chemins *superflus* dans l'analyse.

Analyse interprocédurale avec clonage

On peut analyser chaque appel à `f` indépendamment :

- améliore la précision
- problème de montée en échelle : le temps d'analyse explose
- impossibilité de supporter les appels récursifs

Sensitivité au contexte (*context sensitivity*)

On peut remplacer le treillis $L = N \to L_\sigma$ par :

$$L = \mathit{Context} \to N \to L_\sigma$$

- Il faut adapter la définition de JOIN pour prendre le contexte en compte
- *Context* représente la sensitivité de l'analyse
- Chaque nœud est analysé en fonction du contexte dans lequel on l'atteint
- Le calcul de l'analyse est rendu plus complexe
- La précision peut être meilleure

Sensitivité au site d'appel (*call-site sensitivity*) : chaîne d'appel (*call string approach*)

On préserve les k appels de fonctions (syntaxiques) de la pile d'appel

$$\mathit{Context} = \mathit{Call}^{\leq k}$$

```
int x = f(0); // contexte = f(0)
int y = f(1); // contexte = f(1)
```

- même précision que si on *inline* les appels avec une profondeur k
- $k = 1$ est souvent pas assez précis
- $k \geq 1$ est souvent trop coûteux
- utilisation d'heuristique pour trouver la valeur de k en fonction du site d'appel

Sensitivité au site d'appel (*call-site sensitivity*) : approche fonctionnelle (*functional approach*)

Soit la variation suivante :

```
int x = f(5); // contexte = f(5)
int y = f(1); // contexte = f(1)
```

On analyse deux fois `f` avec exactement le même état !

$$Context = L_\sigma$$

- Précision optimale
- Complexité élevée
- Souvent appliquée de façon sélective
  - seulement pour des fonctions cruciales
  - en restreignant l'état à seulement certaines valeurs (e.g., l'objet receveur)

Analyse d'intervalles

On souhaite savoir pour chaque variable entière, sa valeur sous forme d'intervalle :

$$L = \{[l, h] | l, h \in \mathbb{Z}\}$$
$$[l, h] \sqsubseteq [l', h'] \iff l' \leq l \wedge h \leq h'$$

Utile pour détecter des erreurs de bornes :

- `x` a une valeur dans $[1, 65]$ d'après l'analyse
- `a` a une taille dans $[0, 64]$ d'après l'analyse
- $\Rightarrow$ `a[x]` peut mener à un accès hors borne

Analyse d'intervalles : équations

- Si n est une assignation $x = a$:

$$[\![n]\!] = \lambda s.\sigma[x \mapsto \mathit{eval}(\sigma, a)$$
$$\text{où } \sigma = \textsc{join}(n)(s)$$

- Sinon :

$$[\![n]\!] = \lambda s.\textsc{join}(n)(s)$$

Avec *eval* qui définit comment évaluer une expression sur les intervalles

Analyse d'intervalles : eval

$$
\begin{aligned}
\mathit{eval}(\sigma, \mathtt{x}) &= \sigma(\mathtt{x}) && \text{variable} \\
\mathit{eval}(\sigma, i) &= [i, i] && \text{constante} \\
\mathit{eval}(\sigma, \mathtt{input}) &=]-\infty, +\infty[&& \text{entrée utilisateur} \\
\mathit{eval}(\sigma, e_1 + e_2) &= \mathit{eval}(\sigma, e_1) \hat{+} \mathit{eval}(\sigma, e_2) && \text{opération arithmétique} \\
\text{où } [l, h] \hat{+} [l', h'] &= [\min\{l + l', l + h', \\
&\qquad l' + h, l' + h'\}, \\
&\quad \max\{l + l', l + h', \\
&\qquad l' + h, l' + h'\}] \\
&\ldots
\end{aligned}
$$

Analyse d'intervalles : problème

```
int n = 0;
while (random()) {
  n += 1
}
```

Le calcul du point fixe trouve les valeurs suivantes pour `n` :

- $\texttt{n} = [0, 0]$
- $\texttt{n} = [0, 1]$
- $\texttt{n} = [0, 2]$
- ...

Mais le calcul ne se termine jamais !

Analyse d'intervalles : source du problème

Notre treillis contient des *chaîne ascendante infinies* !

Le calcul du point fixe nous garantit la terminaison seulement pour les treillis qui garantissent la condition de chaîne ascendante (*ascending chain condition*, ACC)

Élargissement *widening*

Fonction d'élargissement

Pour un treillis $(L, \sqsubseteq)$, une fonction d'élargissement $\omega : L \to L$ est une fonction qui est :

- Monotone : $\forall x, y \in L : x \sqsubseteq y \implies \omega(x) \sqsubseteq \omega(y)$
- Extensive : $\forall x \in L \ : x \sqsubseteq \omega(x)$
- Ayant une image $\{y \in L | \exists x \in L : y = \omega(x)\}$ de taille finie

Une fonction d'élargissement garantit que la séquence $(\omega \circ f)^i(\bot)$ converge vers un point fixe plus large que $f^i(\bot)$.

Élargissement d'intervalles

On peut définir une fonction d'élargissement sur notre treillis $L = N \to L_\sigma$:

$$\omega(s) = \bigsqcup_{n \mapsto \sigma \in s} \left(\bigsqcup_{x \mapsto v \in \sigma} [x \mapsto \omega'(v)] \right)$$

Où :

$$\omega'(\bot) = \bot$$
$$\omega'([l,h]) = [max\{i \in B | i \leq l\}, \min\{i \in B | h \leq i\}]$$

Avec un ensemble fini $B \subseteq \mathcal{Z}$ qui contient $-\infty$ et $+\infty$.

On utilise généralement l'ensemble des constantes apparaissant dans le programme pour construire B

Calcul de point fixe avec élargissement

On remplace le calcul de $\bigsqcup_i f^i(\bot)$ par $\bigsqcup_i (\omega \circ f)^i(\bot)$ car :

$$\bigsqcup_i f^i(\bot) \sqsubseteq \bigsqcup (\omega \circ f)^i(\bot)$$

L'idée est de *forcer* le point fixe à converger vers certaines bornes.

On obtient :

- une sur-approximation de la sur-approximation
- dans un temps fini

Rétrecissement (*narrowing*)

Une propriété de ω est que :

$$\forall n : \bigsqcup_i f^i(\bot) \sqsubseteq f^n(\bigsqcup_i (\omega \circ f)^i(\bot)) \sqsubseteq \bigsqcup_i (\omega \circ f)^i(\bot)$$

Après avoir effectuée une étape d'élargissement, on peut donc appliquer f un nombre fini de fois pour redescendre dans le treillis de façon sûre.

Élargissement traditionnel

Pour aller plus loin, on peut introduire un opérateur $\nabla : L \to L \to L$ qui prend en compte l'état courant du calcul.

Pour les intervalles, ∇ vient "stabiliser" les bornes instables.

On n'applique pas ∇ à chaque itération, mais seulement à certain points stratégiques. Typiquement, cela se fait en tête de boucle.

Sensitivité aux branches (*branch sensitivity*)

```
int foo() {
  int x = 10;
  while (x > 0) {
    x = x-1;
  }
  return x;
}
```

Sensitivité aux branches (*branch sensitivity*)

Une analyse d'intervalles nous donne :

- itération 1, entrée de la boucle : $\mathtt{x} \mapsto [10, 10]$
- itération 1, sortie de la boucle : $\mathtt{x} \mapsto [9, 9]$
- itération 2, entrée de la boucle : $\mathtt{x} \mapsto [10, 10] \sqcup [9, 9] = [9, 10]$
- itération 2, entrée de la boucle : $\mathtt{x} \mapsto [9, 9] \sqcup [8, 9] = [8, 9]$
- ...
- itération 10, entrée de la boucle : $\mathtt{x} \mapsto [0, 10]$
- itération 10, sortie de la boucle : $\mathtt{x} \mapsto [-1, 9]$
- itération 11, entrée de la boucle : $\mathtt{x} \mapsto [-1, 10]$
- itération 11, sortie de la boucle : $\mathtt{x} \mapsto [-2, 9]$
- après *widening*, à l'entrée de la boucle : $\mathtt{x} \mapsto]-\infty, 9]$

Sensitivité aux branches (*branch sensitivity*)

On peut encoder de l'information relative aux branches prises :

```
int foo() {
  int x = 10;
  while (x > 0) {
    assert(x > 0);
    x = x-1;
  }
  assert(!(x > 0));
  return x;
}
```

Sensitivité aux branches (*branch sensitivity*)

On adapte la fonction de transfert pour restreindre la valeur de $\mathtt{x}$ en fonction des assertions :

- à l'entrée de la boucle : $\mathtt{x} \mapsto]-\infty, 10] \sqcap [0, +\infty[= [0, 10]$
- à la sortie de la boucle : $\mathtt{x} \mapsto]-\infty, 9] \sqcap [0, +\infty[=]-\infty, 0]$

De façon générale, on parle de *sensitivité au chemin* (*path sensitivity*)

Analyse d'état de fichier

Considérons une analyse de fichier ouvert :

- on considère un programme qui peut ouvrir ou fermer un fichier unique
- treillis $(\mathcal{P}(\{\mathit{open}, \mathit{closed}\}), \subseteq)$

Garanties pertinentes :

- lors de l'ouverture de fichier, le fichier était fermé
- lors d'une fermeture de fichier, le fichier était ouvert
- le fichier est fermé à la fin de l'exécution

Analyse d'état de fichier

```
int flag;
if (cond) {
  open();
  flag = 1;
} else {
  flag = 0;
}

if (flag) {
  close();
}
```

Analyse d'état de fichier

- Si n est l'entrée :

$$[\![n]\!] = \lambda s.\{\mathit{closed}\}$$

- Si n est un appel à `open()` :

$$[\![n]\!] = \lambda s.\{\mathit{open}\}$$

- Si n est un appel à `close()` :

$$[\![n]\!] = \lambda s.\{\mathit{closed}\}$$

- Sinon :

$$[\![n]\!] = \lambda s.\textsc{join}(n)(s)$$

Analyse d'état de fichier

```
int flag;                  // {closed}
if (cond) {
  open();                  // {open}
  flag = 1;
} else {
  flag = 0;                // {closed}
}
                           // {open, closed}
if (flag) {
  close();                 // {closed}
}
                           // {open, closed}
```

$\Rightarrow$ il faut capturer la *relation* entre `flag` et l'état du fichier

Analyse relationnelle : treillis

On change le treillis en :

$$\mathit{Paths} \to \mathcal{P}(\{\mathit{open}, \mathit{closed}\})$$

Avec : $\mathit{Paths} = \{\texttt{flag=0}, \texttt{flag!=0}\}$

Analyse relationnelle : équations

- Si n est l'entrée :

$$[\![n]\!] = \lambda s.\lambda p.\{\mathit{closed}\}$$

- Si n est un appel à `open()` :

$$[\![n]\!] = \lambda s.\lambda p.\{\mathit{open}\}$$

- Si n est un appel à `close()` :

$$[\![n]\!] = \lambda s.\lambda p.\{\mathit{closed}\}$$

Analyse relationnelle : équations

- Si n est une assignation `flag = 0` :

$$[\![n]\!] = \lambda s.[\texttt{flag=0} \mapsto \bigsqcup_{p \in Paths} \textsc{join}(s)(n)(p),$$
$$\texttt{flag!=0} \mapsto \{\}]$$

- Si n est une assignation `flag = 1` :

$$[\![n]\!] = \lambda s.[\texttt{flag=0} \mapsto \{\},$$
$$\texttt{flag!=0} \mapsto \bigsqcup_{p \in Paths} \textsc{join}(s)(n)(p)]$$

- Si n est une assignation `flag = a` (a inconnu) :

$$[\![n]\!] = \lambda s.\lambda q. \bigsqcup \textsc{join}(s)(n)(p)$$

Analyse relationnelle : équations

Il faut à cela ajouter une sensitivité de branches :

- Si n est `assert(flag)` :

$$[\![n]\!] = \lambda s.[\texttt{flag=0} \mapsto \{\},$$
$$\texttt{flag!=0} \mapsto \textsc{join}(n)(s)(\texttt{flag!=0})]$$

- Si n est `assert(!flag)` :

$$[\![n]\!] = \lambda s.[\texttt{flag=0} \mapsto \textsc{join}(n)(s)(\texttt{flag=0}),$$
$$\texttt{flag!=0} \mapsto \{\}]$$

Analyse relationnelle : résultats

```
int flag;                  // [flag=0 -> {closed}, flag!=0 -> {closed}]
if (cond) {
  open();                  // [flag=0 -> {open}, flag!=0 -> {open}]
  flag = 1;                // [flag=0 -> {}, flag!=0 -> {open}]
} else {
  flag = 0;                // [flag=0 -> {closed}, flag!=0 -> {}]
}
                           // [flag=0 -> {closed}, flag!=0 -> {open}]
if (flag) {
  assert(flag!=0);         // [flag=0 -> {}, flag!=0 -> {open}]
  close();                 // [flag=0 -> {}, flag!=0 -> {closed}]
} else {
  assert(flag=0);          // [flag=0 -> {closed}, flag!=0 -> {}]
}
                           // [flag=0 -> {closed}, flag!=0 -> {closed}]
```

Analyse relationnelle : résultats

Garanties :

- lors de l'ouverture de fichier, le fichier était fermé

```
int flag;                  // [flag=0 -> {closed}, flag!=0 -> {cl
if (cond) {
  open();                  // [flag=0 -> {open}, flag!=0 -> {open
```

- lors d'une fermeture de fichier, le fichier était ouvert

```
  assert(flag!=0);         // [flag=0 -> {}, flag!=0 -> {open}]
  close();                 // [flag=0 -> {}, flag!=0 -> {closed}]
```

- le fichier est fermé à la fin de l'exécution

```
                           // [flag=0 -> {closed}, flag!=0 -> {cl
```

Autres domaines relationnels : octagones

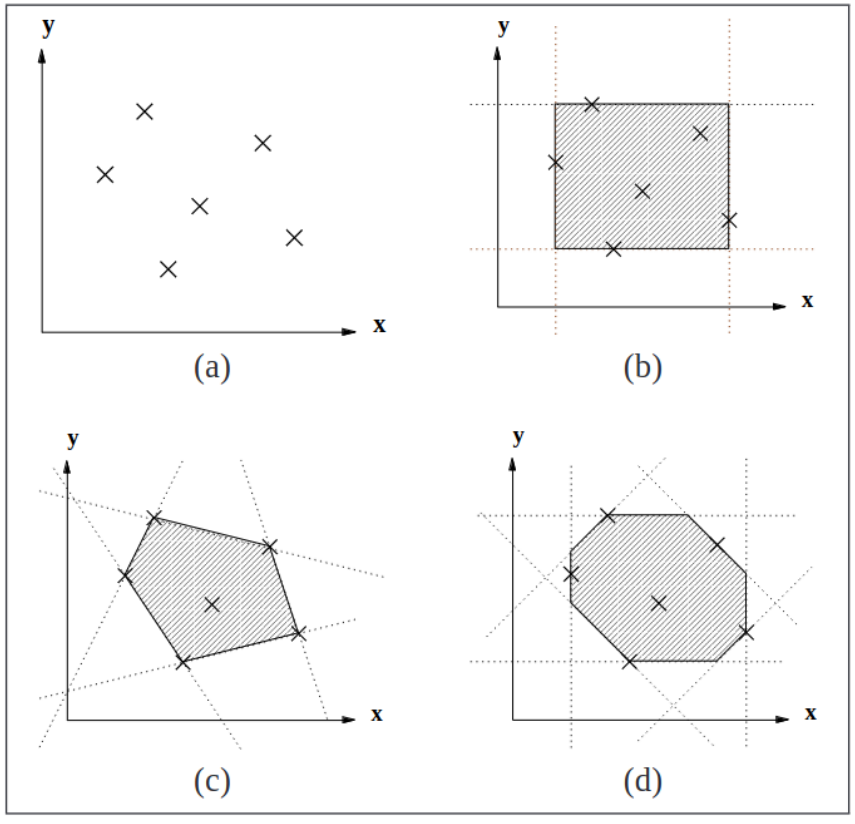


Fig. 2. A set of points (a), and its best approximation in the interval (b), polyhedron (c), and octagon (d) abstract domains.

Miné. (2006). The Octagon Abstract Domain. Higher-order and symbolic computation.

Outils

- LLVM contient de nombreuses analyses dataflow.
- Soot : framework d'analyse de bytecode Java avec possibilité de définir des analyses dataflow. Un peu daté, en cours de réécriture (SootUp) depuis 2022.
- Wala : framework d'analyse pour Java et JavaScript, avec analyses dataflow.
- Phasar : framework analyses dataflow pour LLVM 14.

Articles

- Wei et al. (2018). Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. ACM TOPS.
- Li et al. (2021). MirChecker: Detecting Bugs in Rust Programs via Static Analysis. CCS.
- Park, J. et al. (2016). Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild. ICSE.
- Späth, Ali, & Bodden. (2017). IDE^{al}: Efficient and precise alias-aware dataflow analysis.
- Späth, Ali, & Bodden. (2019). Context-, Flow-, and Field-Sensitive Data-Flow Analysis using Synchronized Pushdown Systems. POPL.
- Xia et al. (2023) Static Semantics Reconstruction for Enhancing JavaScript-WebAssembly Multilingual Malware Detection. ESORICS.

Pratique

Pratique

Pour chacune des analyses définies dans :

- `code/LiveVars/`
- `code/AvailableExps/`

1. Compiler et exécuter la passe LLVM correspondante (voir `README`)
2. Lire le code de l'analyse (`LiveVars.cpp` et `AvailableExps.cpp`)
3. Lire le code de `DataflowAnalysis.h` dans une des deux analyses

Pratique

Ensuite, implémenter soit une analyse de définition atteignante, soit une analyse d'expression fort occupée, selon le même schéma :

1. Implémenter le treillis
2. Implémenter la fonction de transfert
3. Exécuter sur un exemple