Sémantiques mécanisées, troisième cours

Compiler mieux: optimisations, analyses statiques, et leur vérification

Xavier Leroy

2019-12-19

Collège de France, chaire de sciences du logiciel

Les optimisations dans les compilateurs

Transformer automatiquement le code écrit par le programmeur en code équivalent qui

- S'exécute plus rapidement
  - Éliminer des calculs inutiles ou redondants
  - Utiliser des opérations moins coûteuses
  - Augmenter le parallélisme (d'instructions, de *threads*).
- Est plus compact
- Consomme moins d'énergie
- Résiste mieux aux attaques.

On connaît des dizaines d'optimisations, chacune s'attaquant à un type d'inefficacité spécifique.

Quelques optimisations classiques

Propagation des constantes :

```
a = 1;                    a = 1;
b = 2;          -->       b = 2;
c = a + b;                c = 3;
d = x - a;                d = x + (-1);
```

Élimination du code mort :

```
a = 1;                    skip;
b = 2;          -->       b = 2;
c = 3;                    c = 3;
```

(si `a` est inutilisée par la suite)

Quelques optimisations classiques

Factorisation des calculs répétés :

```
c = a;                          c = a;
d = a + b;          -->         d = a + b;
e = c + b;                      e = d;
```

Propagation des copies :

```
e = d;                          skip;
f = d + 1;          -->         f = d + 1;
g = e * 2;                      g = d * 2;
```

Quelques optimisations sur les boucles

Sortir des boucles les calculs invariants :

```
for i = 1 to N do                    c = a + b;
  c = a + b;              -->        for i = 1 to N do
  x = x + c * A[i];                    x = x + c * A[i];
done                                 done
```

Simplifier les variables d'induction :

```
                                     a = p;
 for i = 1 to N do                   for i = 1 to N do
   a = p + i * 4;        -->           skip;
   ...                                 ...
                                       a = a + 4;
 done                                done
```

Une optimisation de cache

Pour améliorer la localité spatiale des accès à la mémoire.

≪Tuilage≫ de boucles :

```
for i = 0 to N-1 do
  for j = 0 to N-1 do
    a[i][j] = b[j][i]
  done
done
```

-->

```
for i = 0 to N-1 step K do
  for j = 0 to N-1 step K do
    for i2 = i to i+K-1 do
      for j2 = j to j+K-1 do
        a[i2][j2] = b[j2][i2]
      done
    done
  done
done
```

Optimisations et analyse statique

Certaines optimisations sont valides sans conditions, p.ex.

$$\texttt{x} * 2 \;\to\; \texttt{x} + \texttt{x}$$

$$\texttt{x} * 4 \;\to\; \texttt{x} << 2$$

Beaucoup d'optimisations ne s'appliquent que si certaines conditions sont vraies :

$$\begin{array}{rcll}
\texttt{x} / 4 & \to & \texttt{x} >> 2 & \text{seulement si } \texttt{x} \geq 0 \\
\texttt{x} + 1 & \to & 1 & \text{seulement si } \texttt{x} = 0 \\
\texttt{if x} < \texttt{y then } c_1 \texttt{ else } c_2 & \to & c_1 & \text{seulement si } \texttt{x} < \texttt{y} \\
\texttt{x} := \texttt{y} + 1 & \to & \texttt{skip} & \text{seulement si } \texttt{x} \\
 & & & \text{est inutilisée ensuite}
\end{array}$$

Il faut analyser statiquement le code avant de le transformer.

L'analyse statique

Déterminer à l'avance (≪statiquement≫) des propriétés qui sont vraies de toutes les exécutions possibles du programme.

Le plus souvent ce sont des propriétés des valeurs des variables à un certain point de programme, p.ex.

$$\mathrm{x} = n \qquad \mathrm{x} \in [n, m] \qquad \mathrm{x} = expr \qquad a.\mathrm{x} + b.\mathrm{y} \leq n$$

(x, y : variables du programme ;
n, m, a, b constantes déterminées par l'analyse.)

Il ne s'agit pas d'exécuter le programme une ou plusieurs fois :

- Les entrées du programme sont inconnues.
- L'analyse doit terminer.
- L'analyse doit prendre peu de temps et d'espace.

Utilisations d'analyses statiques

1960–2000 : pour améliorer les performances du code

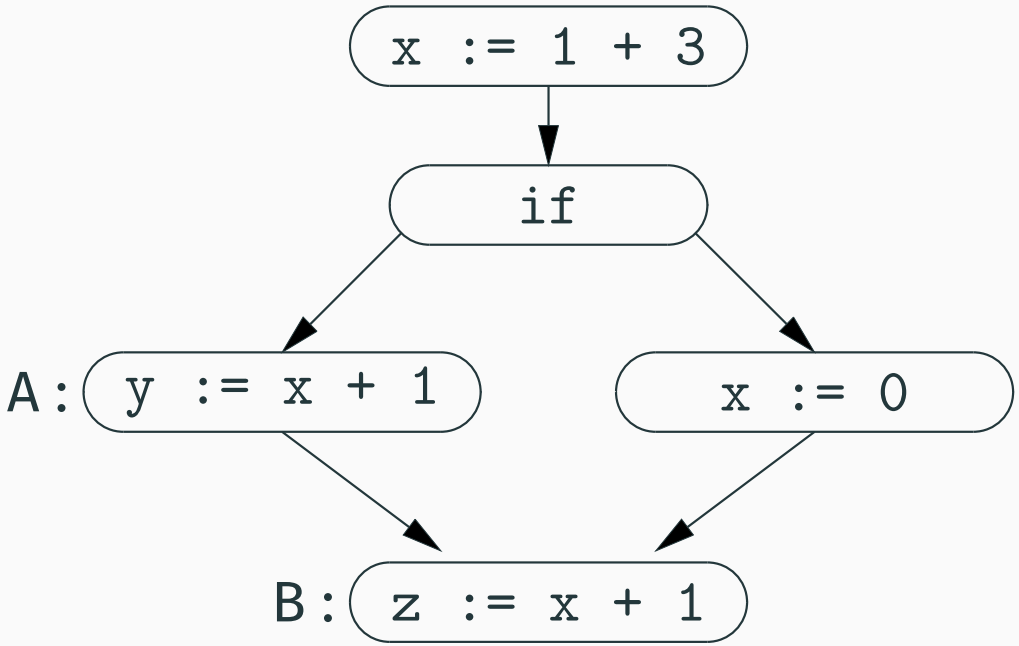
- Établir quand une optimisation peut s'effectuer.
- Guider des heuristiques de génération de code.

Depuis 2000 : pour améliorer la fiabilité des programmes

- Garantir l'absence de certaines erreurs d'exécution (p.ex. accès hors bornes dans des tableaux).
- Plus modestement : signaler des erreurs plausibles de programmation.

L'approche par flux (de contrôle et de données)

Sur le graphe de flux de contrôle, relier les définitions et les utilisations des variables.



Point d'utilisation A, une seule définition possible de `x` : `x = 4`.
Point d'utilisation B, deux définitions incompatibles possibles : `x = 4` et `x = 0`.

Analyse par flux de données (*dataflow analysis)*

Associer aux noeuds n du graphe de flux de contrôle des ensembles de faits (p.ex «variable = constante»), reliés par des équations de flux :

$$in(n) = \bigcap \{out(p) \mid p \text{ prédecesseur de } n\}$$

$$out(n) = gen(n) \cup (in(n) \setminus kill(n))$$

$in(n)$: faits vrais «avant» l'exécution de n
$out(n)$: faits vrais «après» l'exécution de n
$kill(n)$: faits invalidés par l'exécution de n
(p.ex. $x := a$ invalide «$x = N$» pour tout N)
$gen(n)$: faits établis par l'exécution de n
(p.ex. $x := 1 + 2$ garantit «$x = 3$»)

Analyse par flux de données (*dataflow analysis*)

$$in(n) = \bigcap \{out(p) \mid p \text{ prédecesseur de } n\}$$
$$out(n) = gen(n) \cup (in(n) \setminus kill(n))$$

Résoudre ces équations par itération de point fixe.
($\approx$ Recalculer $out(n)$ dès que l'un des $out(p)$ a changé.)

Se généralise des ensembles à des treillis (de hauteur finie).

G. Kildall, *A unified approach to global program optimization*, POPL 1973.

J. B. Kam et J. D. Ullman, *Monotone data flow analysis frameworks*, Acta Informatica 1977.

Exemple : élimination de code mort par analyse de vivacité

Éliminer les affectations $x := a$, en les remplaçant par `skip`, si x n'est jamais utilisée plus tard dans l'exécution du programme.

Exemple

```
x := 1;  y := y + 1;  x := 2
```

L'affectation `x := 1` peut être supprimée car `x` n'est pas utilisée avant d'être redéfinie par `x := 2`.

Cette optimisation s'appuie sur une analyse statique appelée analyse de vivacité (*liveness analysis*).

C'est une analyse de type ≪flux de données en arrière≫.

L’analyse de vivacité

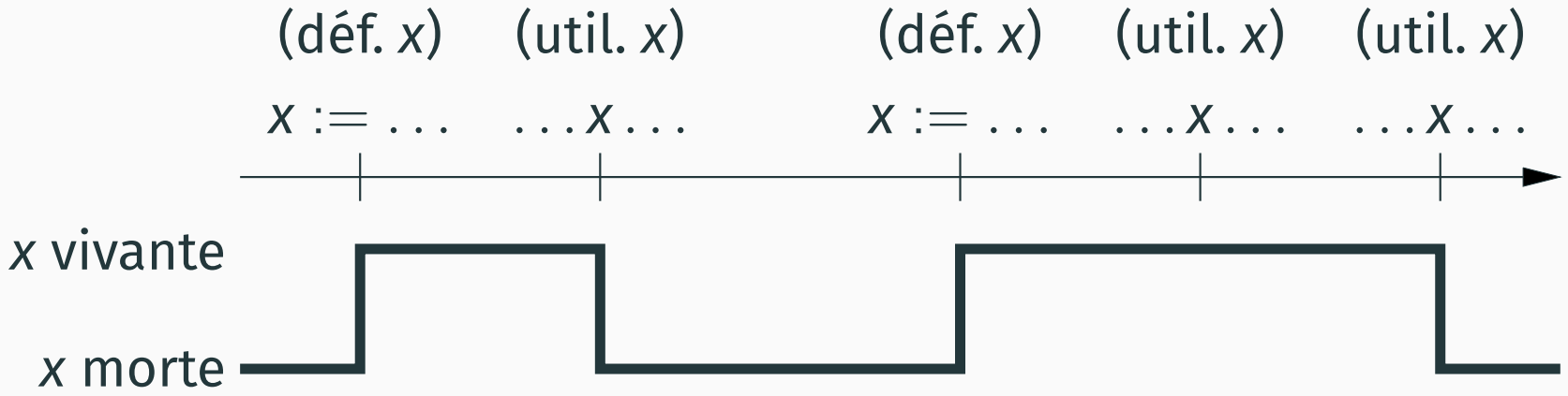
Vivacité d'une variable

Une variable est morte en un point de programme si sa valeur n'est pas utilisée ensuite dans l'exécution du programme :

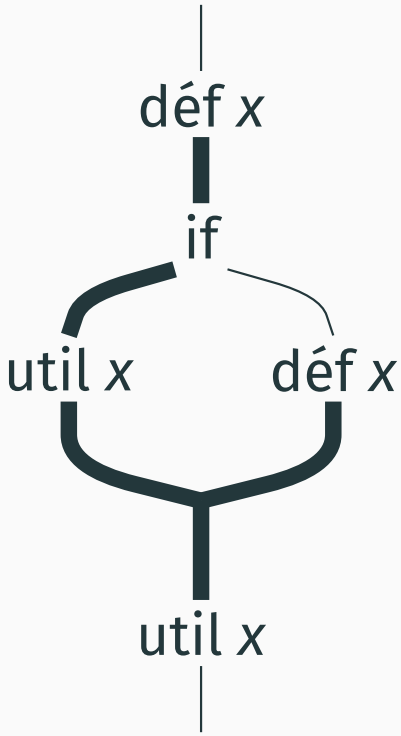
- la variable n'est plus mentionnée jusqu'à la fin de sa portée ;
- ou elle est redéfinie avant toute utilisation.

Une variable est vivante si elle n'est pas morte.

Facile à déterminer pour du code sans branchements :



Vivacité d'une variable



En présence de branchements, on fait une sur-approximation de la vivacité, en supposant que les conditions des `if` peuvent être vraies ou fausses, et que les boucles `while` sont exécutées 0 ou plusieurs fois.

Équations ≪flux de données≫ pour la vivacité

Si L est l'ensemble des variables vivantes ≪après≫ la commande c, on définit $\texttt{live}\ c\ L$, l'ensemble des variables vivantes ≪avant≫ c.

$$\texttt{live}\ \texttt{SKIP}\ L = L$$

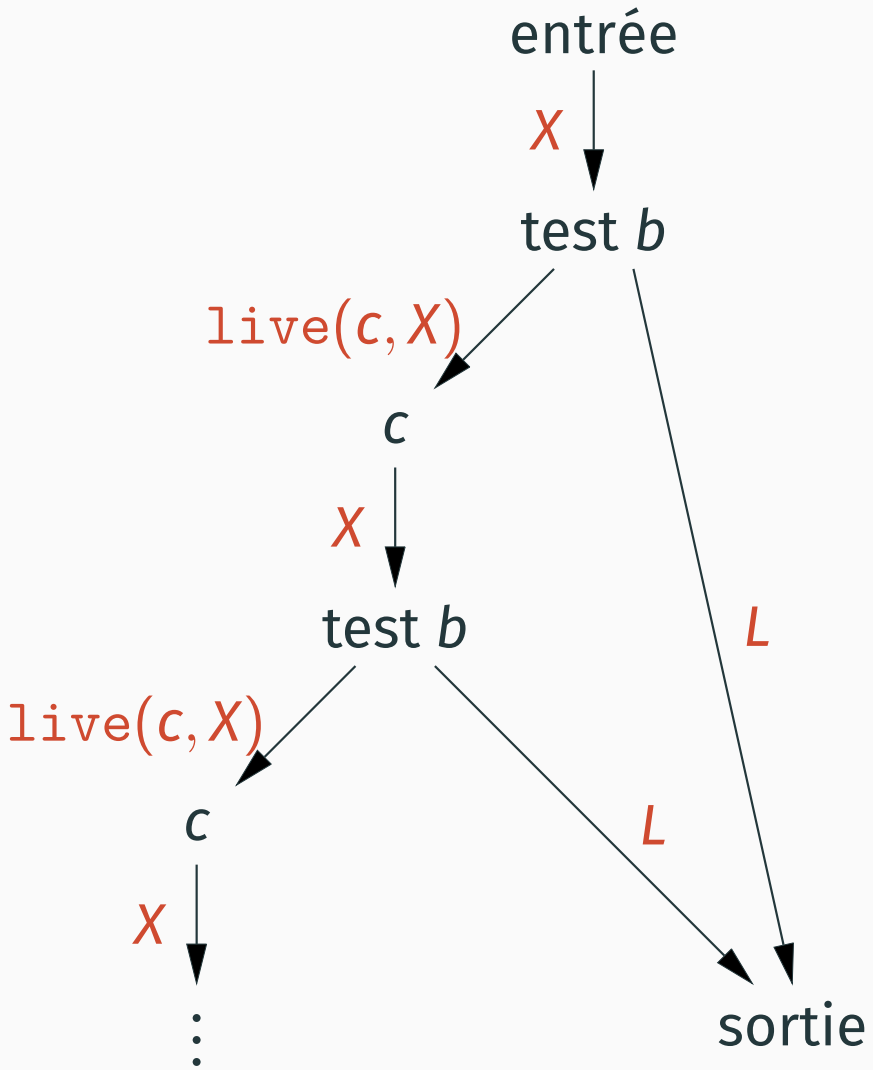
$$\texttt{live}\ (x := a)\ L = \begin{cases} (L \setminus \{x\}) \cup FV(a) & \text{si } x \in L; \\ L & \text{si } x \notin L. \end{cases}$$

$$\texttt{live}\ (c_1; c_2)\ L = \texttt{live}\ c_1\ (\texttt{live}\ c_2\ L)$$

$$\texttt{live}\ (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ L = FV(b) \cup \texttt{live}\ c_1\ L \cup \texttt{live}\ c_2\ L$$

$$\texttt{live}\ (\texttt{while}\ b\ \texttt{do}\ c)\ L = X \text{ tel que } X = L \cup FV(b) \cup \texttt{live}\ c\ X$$

Analyse de vivacité pour une boucle



X doit vérifier :

- $FV(b) \subseteq X$
  (pour évaluer b)
- $L \subseteq X$
  (si b est fausse)
- $\texttt{live}(c, X) \subseteq X$
  (si b est vraie et c est exécutée)

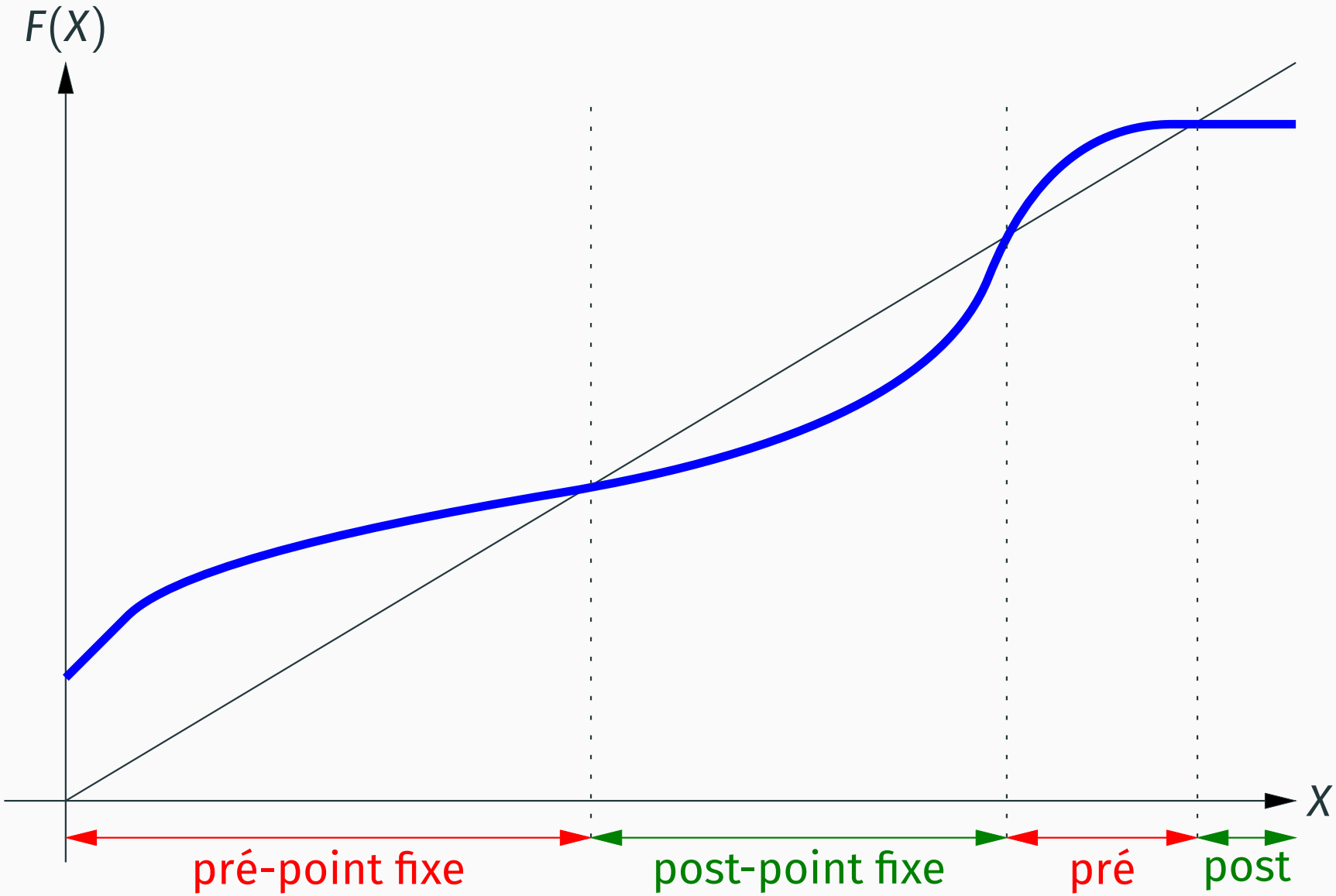
Les points fixes

Soit $F = \lambda X.\ L \cup FV(b) \cup \texttt{live}(c, X)$.

Pour analyser la boucle `while` b `do` c, on voudrait calculer un plus petit point fixe de F, c'est à dire un X minimal tel que $F(X) = X$.

C'est ce qui maximise la précision de l'analyse. Pour la correction sémantique, il suffit d'avoir un post-point fixe de F, c'est à dire un X tel que $F(X) \subseteq X$.

Points fixes d'une fonction croissante



L'approche du mathématicien

Théorème (Knaster-Tarski)

Soit $A, \leq$ un type partiellement ordonné, et $F : A \to A$.

La suite $\bot,\ F(\bot),\ F(F(\bot)),\ \ldots,\ F^n(\bot), \ldots$

converge sur le plus petit point fixe de F, à condition que

- *F est croissante : $x \leq y \Rightarrow F(x) \leq F(y)$.*
- *$\bot$ est le plus petit élément de A.*
- *Il n'existe pas de suite infinie strictement croissante $x_0 < x_1 < \cdots < x_n < \cdots$*

Difficultés avec le théorème de Knaster-Tarski

La condition sur les suites strictement croissantes est

1. pas facile à formaliser et à utiliser
   (ordres bien fondés + récursion Noetherienne);
2. parfois fausse ! Dans le cas de l'analyse de vivacité, l'ordre $\subset$ admet des suites infinies croissantes :
   $\emptyset \subset \{x_1\} \subset \{x_1, x_2\} \subset \cdots$

Essayons une autre approche…

L'approche de l'ingénieur

$$F = \lambda X.\ L \cup FV(b) \cup \texttt{live}(c, X)$$

On procède par itération bornée :

- Calculer $F(\emptyset), F(F(\emptyset)), \ldots, F^N(\emptyset)$ jusqu'à un N donné.
- S'arrêter dès qu'on trouve un post-point fixe ($F^{i+1}(\emptyset) \subseteq F^i(\emptyset)$).
- Sinon, renvoyer une sur-approximation que l'on sait être un post-point fixe (dans notre cas, $L \cup FV(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done})$).

Un compromis entre temps d'analyse et précision de l'analyse.

Élimination de code mort

Élimination de code mort

La transformation de programme supprime les affectations à des variables mortes :

$x := a$ devient `SKIP` si x n'est pas vivante «après» l'affectation

Implémenté comme une fonction

$$\texttt{dce} : \texttt{com} \to \texttt{IdentSet.t} \to \texttt{com}$$

qui prend en second argument l'ensemble des variables vivantes «après» la commande, et tient à jour cet ensemble pendant la récursion.

(Implémentation & exemples dans le module `Optim`.)

Interprétation sémantique de la vivacité

Comment caractériser sémantiquement qu'une variable x est vivante en un point du programme ?

Hmmm…

Interprétation sémantique de la vivacité

Comment caractériser sémantiquement qu'une variable x est vivante en un point du programme ?

Hmmm…

Comment caractériser sémantiquement qu'une variable x est morte en un point du programme ?

Par le fait que la valeur exacte de x en ce point n'a pas d'impact sur l'exécution du programme !

La vivacité : une ≪hyper-propriété≫ de deux exécutions

Considérons deux exécutions de la même commande c dans des états initiaux différents :

$$c/s_1 \Downarrow s_1' \qquad\qquad c/s_2 \Downarrow s_2'$$

Supposons que les états initiaux coïncident sur les variables $\texttt{live}\ c\ L$ qui sont vivantes ≪avant≫ c :

$$\forall x \in \texttt{live}\ c\ L,\ \ s_1(x) = s_2(x)$$

Alors, les deux exécutions terminent sur des états finaux qui coïncident sur les variable L vivantes ≪après≫ c :

$$\forall x \in L,\ \ s_1'(x) = s_2'(x)$$

La relation `agree` et ses propriétés

```
Definition agree (L: IdentSet.t) (s1 s2: store) : Prop :=
  forall x, IdentSet.In x L -> s1 x = s2 x.
```

La relation est décroissante par-rapport à l'ensemble `L` :

```
Lemma agree_mon:
  forall L L' s1 s2,
  agree L' s1 s2 -> IdentSet.Subset L L' -> agree L s1 s2.
```

Une expression s'évalue à la même valeur dans deux états qui coïncident sur ses variables libres :

```
Lemma aeval_agree:
  forall a s1 s2, agree (fv_aexp a) s1 s2 -> aeval a s1 = aeval a s2.
Lemma beval_agree:
  forall b s1 s2, agree (fv_bexp b) s1 s2 -> beval b s1 = beval b s2.
```

La relation `agree` et ses propriétés

La relation est préservée par affectation en parallèle à une variable :

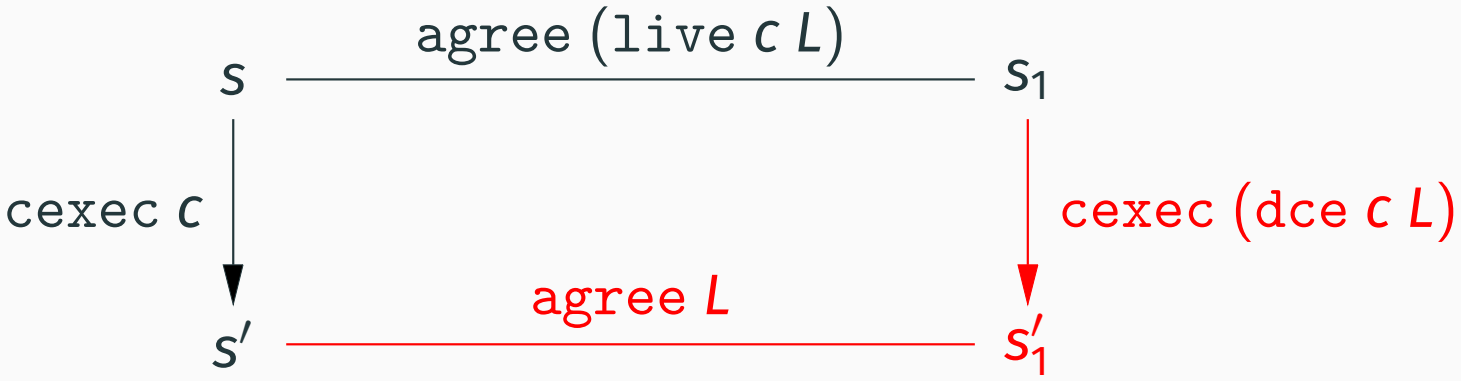
```
Lemma agree_update_live:
  forall s1 s2 L x v,
  agree (IdentSet.remove x L) s1 s2 ->
  agree L (update x v s1) (update x v s2).
```

La relation est aussi préservée par affectation unilatérale à une variable qui est morte :

```
Lemma agree_update_dead:
  forall s1 s2 L x v,
  agree L s1 s2 -> ~IdentSet.In x L ->
  agree L (update x v s1) s2.
```

Préservation de la sémantique

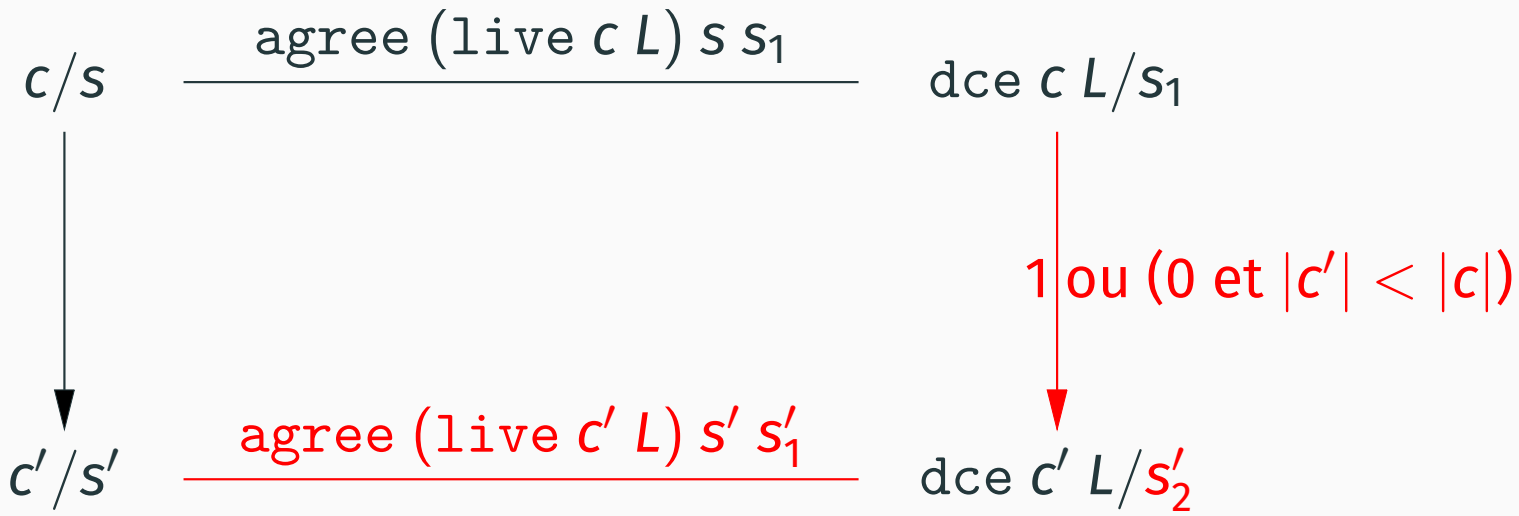
On montre que l'exécution de `dce c L` simule l'exécution de c en préservant la relation `agree` entre les états mémoire.




```
Theorem dce_correct_terminating:
  forall s c s', cexec s c s' ->
  forall L s1,
  agree (live c L) s s1 ->
  exists s1', cexec s1 (dce c L) s1' /\ agree L s' s1'.
```

Préservation de la sémantique

On peut étendre ce résultat aux programmes qui divergent en montrant un diagramme de simulation utilisant la sémantique à réductions d'IMP. (Exercice.)



Pour aller plus loin : l'allocation de registres

L'allocation de registres

Placer les variables utilisées par le programme (en quantité arbitraire) dans :

- ou bien des registres du processeur
  (accès très rapide ; petit nombre de registres disponibles)
- ou bien des cases mémoire (généralement dans la pile)
  (disponibles en quantité arbitraire ; accès plus lent).

Essayer de maximiser l'utilisation des registres du processeur.

Une étape cruciale pour produire du code machine efficace.

Deux approches de l'allocation de registres

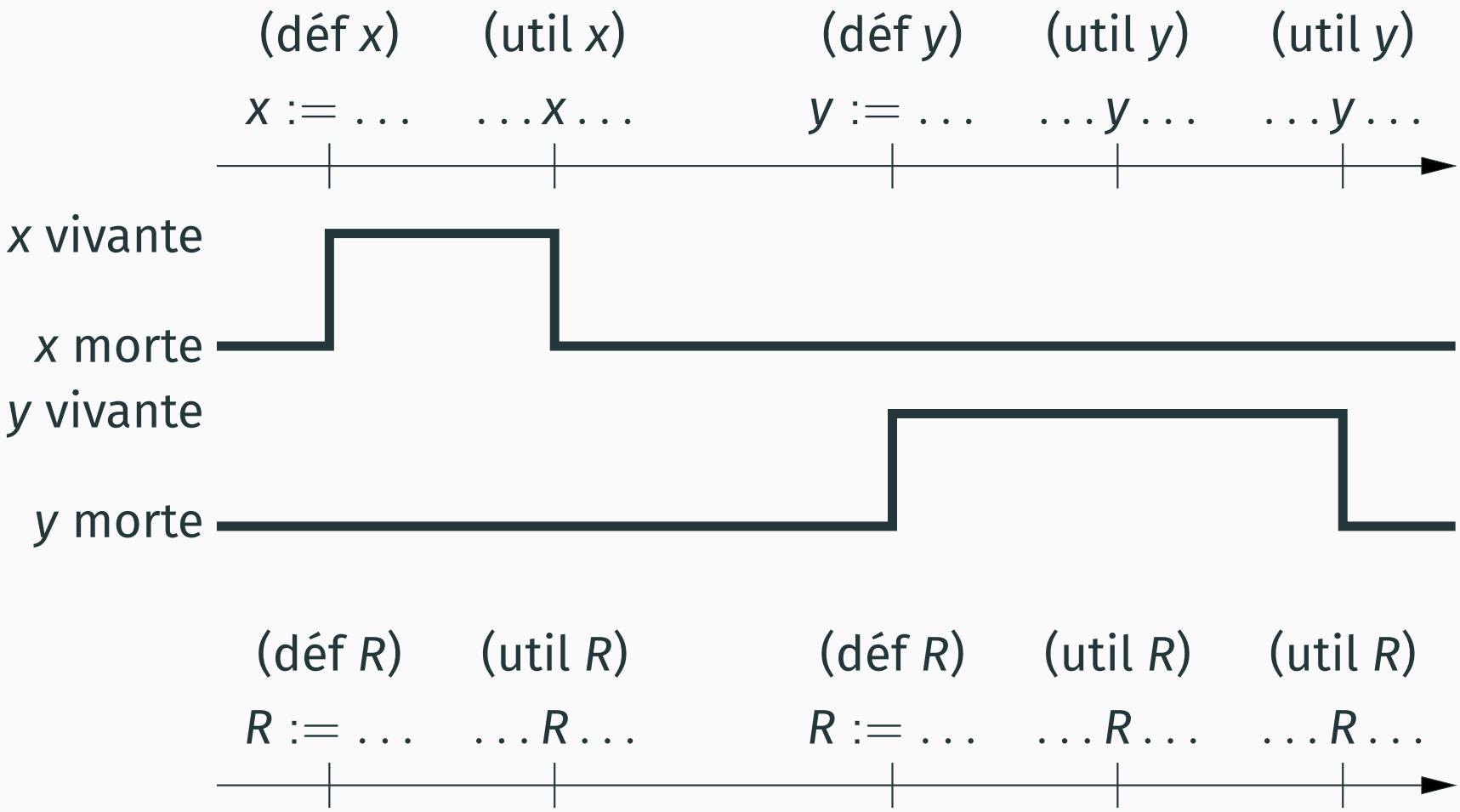
Approche naïve : (placement injectif)

- Placer les N variables les plus utilisées dans les N registres disponibles.
- Placer les autres variables en mémoire.

Approche optimisée (placement non injectif)

- Placer plusieurs variables dans un même registre, pourvu que ces variables ne sont jamais vivantes en même temps.

Exemple de partage de registre



Une allocation de registres pour IMP

Présentation simplifiée : une transformation IMP $\to$ IMP qui cherche à minimiser le nombre de variables différentes utilisées.

Le renommage (non injectif) des variables peut être vu comme le placement des variables en registres ou emplacements mémoire.

La transformation de programme

On suppose donné un placement des variables
$f : \texttt{ident} \to \texttt{ident}$.

La transformation de programme consiste à :

- Renommer les variables : x devient $f\ x$.
- Éliminer le code mort :

  $$x := a \quad \longrightarrow \quad \texttt{SKIP} \quad \text{si } x \text{ est morte } \ll\text{après}\gg$$

- Éliminer les affectations redondantes :

  $$x := y \quad \longrightarrow \quad \texttt{SKIP} \quad \text{si } f\ x = f\ y$$

Conditions de correction sur le placement des variables

Tous les placements f ne préservent pas la sémantique !

Exemple

Supposons $f\ \mathtt{x} = f\ \mathtt{y} = f\ \mathtt{z} = \mathtt{R}$

```
x := 1;                    R := 1;
y := 2;          ---->     R := 2;
z := x + y;                R := R + R;
```

Le code transformé met 4 dans `R` au lieu de 3…

Quelles sont les conditions suffisantes sur f pour préserver la sémantique ? On peut les découvrir en retravaillant notre démonstration de l'élimination de code mort.

Retour sur la relation `agree`

```
Definition agree' (L: IdentSet.t) (s1 s2: store) : Prop :=
  forall x, IdentSet.In x L -> s1 x = s2 (f x).
```

Une expression et son renommage s'évaluent en la même valeur dans des états mémoire reliés :

```
Lemma aeval_agree':
  forall a s1 s2,
  agree' (fv_aexp a) s1 s2 -> aeval a s1 = aeval (rename_aexp a) s2.
Lemma beval_agree':
  forall b s1 s2,
  agree' (fv_bexp b) s1 s2 -> beval b s1 = beval (rename_bexp b) s2.
```

Retour sur la relation `agree`

Comme précédemment, la relation est préservée par affectation unilatérale à une variable morte :

```
Lemma agree'_update_dead:
  forall s1 s2 L x v,
  agree' L s1 s2 -> ~IdentSet.In x L ->
  agree' L (update x v s1) s2.
```

Retour sur la relation `agree`

La relation est préservée par affectation parallèle à une variable x et son renommage $f\ x$, mais seulement si f satisfait une condition de non-interférence (en rouge ci-dessous).

```
Lemma agree'_update_live:
  forall s1 s2 L x v,
  agree' (IdentSet.remove x L) s1 s2 ->
  (forall z, IdentSet.In z L -> z <> x -> f z <> f x) ->
  agree' L (update x v s1) (update (f x) v s2).
```

Un cas particulier pour les copies de variables

Dans le cas d'une copie de variable $x := y$, la valeur affectée à x n'est pas arbitraire : on sait que c'est la valeur de y. Cela permet d'affaiblir la condition de non-interférence.

```
Lemma agree'_update_move:
  forall s1 s2 L x y,
  agree' (IdentSet.union (IdentSet.remove x L) (IdentSet.singleton y))
         s1 s2 ->
  (forall z, IdentSet.In z L -> z <> x -> z <> y -> f z <> f x) ->
  agree' L (update x (s1 y) s1) (update (f x) (s2 (f y)) s2).
```

Cela permet de placer x et y dans le même registre, même si x et y sont vivantes en même temps.

Le graphe d'interférences

Les contraintes de non-interférence $f\ x \neq f\ y$ peuvent être regroupées et représentées par un graphe d'interférence :

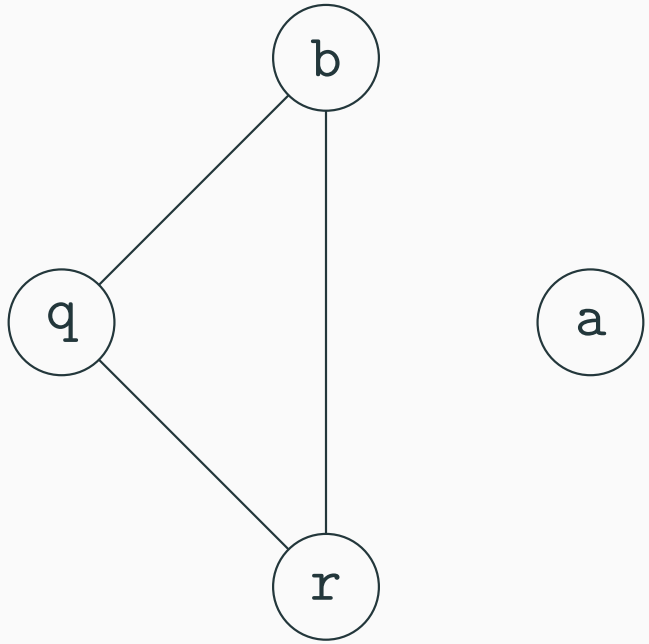
- Noeuds = variables du programme.
- Arc non orienté entre x et y =
  x et y ne doivent pas être placées au même endroit.

Construction du graphe d'interférence (algorithme de Chaitin) :

- Pour chaque copie $x := y$, ajouter un arc entre x et toute variable z vivante ≪après≫ autre que x et y.
- Pour chaque affectation $x := a$, ajouter un arc entre x et toute variable z vivante ≪après≫ autre que x.

Exemple de graphe d'interférences

```
r := a;
q := 0;
while b <= r do
  r := r - b;
  q := q + 1
done
```



Allocation de registres par coloriage de graphe

(G. Chaitin et al, *Register allocation via coloring*, 1981.)
(P. Briggs, *Register allocation via graph coloring*, 1992.)
(L. George et A. W. Appel, *Iterated register coalescing*, 1996.)

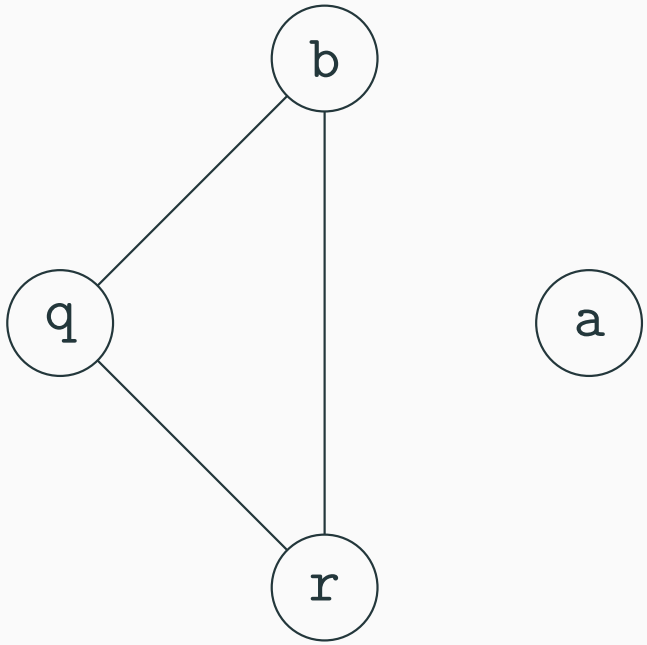
Colorier chaque noeud du graphe d'interférence, avec une couleur qui est un registre ou une case mémoire,

sous la contrainte que les deux extrémités d'un arc ont des couleurs différentes,

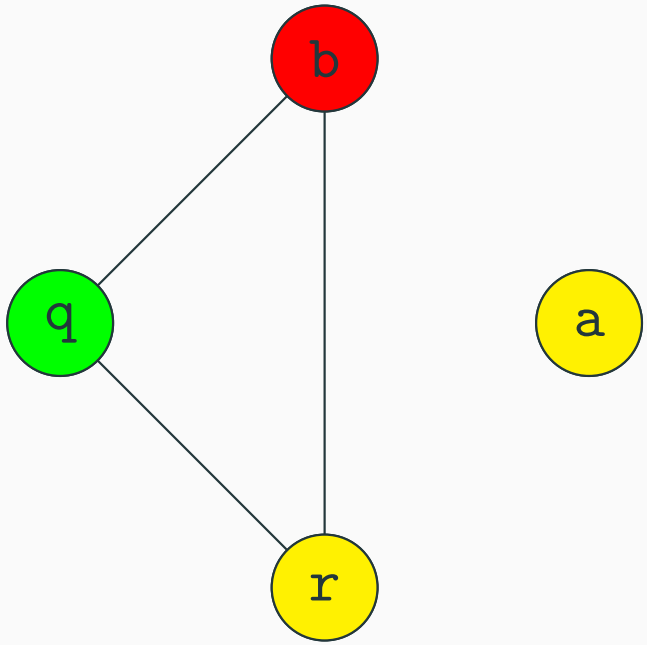
en minimisant le nombre des noeuds coloriés avec une case mémoire.

(Problème NP-complet en général, mais on a de bonnes heuristiques en temps linéaire.)

Exemple de coloriage



Exemple de coloriage




```
yellow := yellow;
green := 0;
while red <= yellow do
  yellow := yellow - red;
  green := green + 1
done
```

Que faut-il vérifier en Coq ?

Vérification directe du compilateur :
formaliser et vérifier une bonne heuristique de coloriage de graphe.

L'algorithme IRC de George et Appel $\approx 6\,000$ lignes de Coq.

Validation a posteriori :
faire calculer un placement par un code non vérifié ;
faire valider que le placement satisfait les contraintes de non-interférence par un validateur formellement vérifié ;
arrêter la compilation si le validateur échoue.

Validation d'un placement de variables

On écrit facilement une fonction Coq à valeurs booléenne

```
correct_allocation:
            (ident -> ident) -> com -> IdentSet.t -> bool
```

qui renvoie `true` si et seulement si les propriétés de non-interférence attendues sont satisfaites.

(Ou, en d'autres termes : si et seulement si la fonction `ident -> ident` est un coloriage valide du graphe d'interférences.)

Préservation de la sémantique

Les démonstrations de préservation sémantique que nous avons faites pour l'élimination de code mort s'étendent facilement, avec l'hypothèse supplémentaire que `correct_allocation` renvoie `true` :

```
Theorem regalloc_correct_terminating:
  forall s c s', cexec s c s' ->
  forall L s1,
  agree' (live c L) s s1 ->
  correct_allocation c L = true ->
  exists s1', cexec s1 (regalloc c L) s1' ∧ agree' L s' s1'.
```

Retour sur les points fixes

Retour sur l'approche du mathématicien

Théorème (Knaster-Tarski)

Soit $A, \leq$ un type partiellement ordonné, et $F : A \to A$.

La suite $\bot,\ F(\bot),\ F(F(\bot)),\ \ldots,\ F^n(\bot), \ldots$

converge sur le plus petit point fixe de F, à condition que

- *F est croissante : $x \leq y \Rightarrow F(x) \leq F(y)$.*
- *$\bot$ est le plus petit élément de A.*
- *Il n'existe pas de suite infinie strictement croissante*
  $x_0 < x_1 < \cdots < x_n < \cdots$

Ce théorème fournit un algorithme de calcul de points fixes !
(Voir le fichier Coq `Fixpoints`).

Ordres bien fondés

Une reformulation plus constructive de la condition ≪il n'existe pas de suite infinie strictement croissante≫ :

Toutes les suites strictement croissantes sont finies.

Autrement dit : l'ordre $>$ est bien fondé.

Autrement dit : $\forall x : A, \texttt{Acc}\ x$
où `Acc` est le prédicat d'accessibilité

```
Inductive Acc: A -> Prop :=
  | Acc_intro : (forall y:A, y > x -> Acc y) -> Acc x.
```

Récurrence et récursion noetheriennes

Démonstration par récurrence noetherienne :
pour montrer $P(x)$, on peut supposer $P(y)$ vrai pour tout $y > x$.

($\approx$ récurrence structurelle sur la dérivation de `Acc` x.)

Programmation par récursion noetherienne :
$F(x)$ peut rappeler $F(y)$ pour un ou plusieurs $y > x$.

Application : un algorithme de calcul du plus petit point fixe !
(Voir le module `Fixpoints`.)

Application à l'analyse de vivacité

(Voir le module `Optim`, section 3.4.)

1. Pour que l'ordre $\subset$ soit bien fondé, il faut se restreindre aux sous-ensembles d'un univers fini `U`.

   ```
   Definition finset := { x: IdentSet.t | IdentSet.Subset x U }
   ```

2. On obtient alors un opérateur de point fixe :

   ```
   finset_fixpoint:
     forall (F: finset -> finset), finset_monotone F -> finset
   ```

3. Pour pouvoir l'utiliser dans l'analyse de vivacité, il faut en même temps définir la fonction d'analyse et montrer qu'elle est croissante.

   ```
   Program Fixpoint live'
        (c: com) (CONT: IdentSet.Subset (fv_com c) U)
     : { f: finset -> finset | monotone f } := ...
   ```

Point d’étape

Point d'étape

Les analyses statiques :

- Aujourd'hui : approche ≪flux de données≫.
- Variante : analyses parcimonieuses sur la forme SSA.
- Le 16 janvier : approche ≪interprétation abstraite≫.

Un besoin essentiel : le calcul efficace de post-points fixes précis.

- Aujourd'hui : itérations bornées locales (une pour chaque boucle).
- Classiquement dans les compilateurs : itération globale sur un graphe de flot de contrôle.
- Le 16 janvier : itérations locales accélérées par élargissement (*widening*).

Bibliographie

Bibliographie

Optimisations et analyses statiques «flux de données» :

- A. W. Appel, *Modern Compiler Implementation in Java / ML / C*, Cambridge University Press, 1998. Chapitres 10, 11, 17, 18.
- Pour aller plus loin : S. S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufman, 1997.

Comment les vérifier formellement :

- X. Leroy, *A formally verified compiler back-end*, J. Autom. Reasoning, 43(4), 2009.

Une variante de l'approche : la forme SSA (*Single Static Assignment*)

- A. W. Appel, *op. cit.*, chapitre 19.
- G. Barthe, D. Demange, D. Pichardie. *Formal verification of an SSA-based middle-end for CompCert*, ACM TOPLAS 36(1), 2014.