



COLLÈGE
DE FRANCE
—1530—

Logiques de programmes, cinquième cours

Quelques extensions de la logique de séparation

Xavier Leroy

2021-04-01

Collège de France, chaire de sciences du logiciel

`xavier.leroy@college-de-france.fr`

**Quelques coups
de baguette magique**



Un problème d'adaptation

Dans beaucoup d'étapes de vérification, on veut appliquer

- une «petite» règle $\{ \ell \mapsto - \} \text{set}(\ell, v) \{ \lambda.. \ell \mapsto v \}$
- ou une «petite» spécification de fonction $\{ \text{list}(w, p) \} \text{reverse}(p) \{ \lambda r. \text{list}(\text{rev}(w), r) \}$

dans un contexte «plus grand», par exemple

$$\text{list}(p, w) * \text{list}(q, w') * \langle x > 0 \rangle * t \mapsto x * t + 1 \mapsto q$$

En général, il faut 1- dérouler des prédicats de représentation,
2- trouver un encadrement, 3- appliquer la règle de conséquence.

La règle de conséquence encadrée

(Dérivée de la règle d'encadrement + la règle de conséquence.)

Une manière générale d'adapter ce que l'on sait déjà $\{P'\} c \{Q'\}$ à ce que l'on cherche à montrer $\{P\} c \{Q\}$.

$$\frac{\{P'\} c \{Q'\} \quad P \Rightarrow P' \star R \quad \forall v, Q' v \star R \Rightarrow Q v}{\{P\} c \{Q\}}$$

La démonstration (semi-)automatique fonctionne assez bien pour montrer les implications $P \Rightarrow P' \star R$ et $Q' v \star R \Rightarrow Q v$.

La difficulté est de trouver l'assertion R .

Le problème de l'abduction

Étant donnés P et Q , trouver X (minimal) tel que $P \star X \Rightarrow Q$.

(En d'autres termes : que manque-t'il à P pour garantir Q ?)

À défaut de calculer une forme simple pour la solution X , on peut la caractériser comme suit :

$$X h = \forall h', h' \perp h \wedge P h' \Rightarrow Q(h' \uplus h)$$

On note cette opération $P \rightarrowstar Q$ et on l'appelle «baguette magique» (*magic wand*) :

$$P \rightarrowstar Q \stackrel{\text{def}}{=} \lambda h. \forall h' \perp h, P h' \Rightarrow Q(h' \uplus h)$$

Baguette magique = implication séparante

L'implication séparante \rightarrow^* est à la conjonction séparante $*$ ce que l'implication usuelle \Rightarrow est à la conjonction usuelle \wedge .

Adjonction :

$$H \Rightarrow (P \rightarrow^* Q) \iff H * P \Rightarrow Q$$

Autres propriétés :

$P * (P \rightarrow^* Q)$	\Rightarrow	Q	(élimination)
emp	\Rightarrow	$P \rightarrow^* P$	(idempotence)
$(P \rightarrow^* Q) * (Q \rightarrow^* R)$	\Rightarrow	$P \rightarrow^* R$	(transitivité)
$(P * Q) \rightarrow^* R$	$=$	$P \rightarrow^* Q \rightarrow^* R$	(curryfication)
$(P \rightarrow^* Q) * R$	\Rightarrow	$P \rightarrow^* (Q * R)$	(distribution)

La règle de conséquence ramifiée

$$\frac{\{P'\} c \{Q'\} \quad P \Rightarrow (P' \star (\forall v, Q' v \rightarrow \star Q v))}{\{P\} c \{Q\}}$$

Comme la règle de conséquence encadrée, mais avec un choix canonique pour le «cadre» : $R = \forall v, Q' v \rightarrow \star Q v$

Remplace la difficulté de trouver R par la difficulté de raisonner sur des formules utilisant $\rightarrow \star$ et \star .

Plus faibles préconditions

En logique de séparation tout comme en logique de Hoare, une commande c avec postcondition Q admet une **plus faible précondition** $wp\ c\ Q$, caractérisée comme suit :

- C'est une précondition : $\{ wp\ c\ Q \} c \{ Q \}$
- C'est la plus faible : si $\{ P \} c \{ Q \}$ alors $P \Rightarrow wp\ c\ Q$

On peut définir $wp\ c\ Q$ de plusieurs manières :

à partir de la sémantique : $wp\ c\ Q = \lambda h. \text{Term } c\ h\ Q$ (ou Safe)

à partir des triplets : $wp\ c\ Q = \exists P. P \star \langle \{ P \} c \{ Q \} \rangle$

Une équivalence avec les triplets :

$$\{P\} c \{Q\} \text{ si et seulement si } P \Rightarrow wp\ c\ Q$$

Une manière d'aborder la vérification sous un angle plus calculatoire (et dirigé par la syntaxe de la commande c) :

« Étant donné un code c et la spécification Q de son résultat, quelle précondition doit vérifier l'état initial pour que c s'exécute sans erreurs et satisfasse Q ? »

Les règles de la logique de séparation se ré-expriment directement en termes de wp :

$$Q \llbracket a \rrbracket \Rightarrow wp \ a \ Q$$

$$wp \ c \ (\lambda v. wp \ c' [x \leftarrow v] \ Q) \Rightarrow wp \ (\text{let } x = c \text{ in } c') \ Q$$

$$(\text{si } \llbracket b \rrbracket \text{ alors } wp \ c_1 \ Q \text{ sinon } wp \ c_2 \ Q) \Rightarrow wp \ (\text{if } b \text{ then } c_1 \text{ else } c_2) \ Q$$

Calculer les plus faibles préconditions

Pour les constructions impératives, les «petites» règles nous donnent des équations de wp qui ne sont pas utilisables, car elles imposent une forme particulière à la postcondition Q .

$$\text{emp} \Rightarrow wp(\text{alloc}(N)) (\lambda \ell. \ell \mapsto _ \star \dots \star \ell + N - 1 \mapsto _)$$

$$\llbracket a \rrbracket \mapsto x \Rightarrow wp(\text{get}(a)) (\lambda v. \langle v = x \rangle \star \llbracket a \rrbracket \mapsto x)$$

$$\llbracket a \rrbracket \mapsto _ \Rightarrow wp(\text{set}(a, a')) (\lambda v. \llbracket a \rrbracket \mapsto \llbracket a' \rrbracket)$$

$$\llbracket a \rrbracket \mapsto _ \Rightarrow wp(\text{free}(a)) (\lambda v. \text{emp})$$

C'est le moment de sortir la baguette magique...

Règles structurelles pour les plus faibles préconditions

Encadrement :

$$(wp \ c \ Q) \ * \ R \Rightarrow wp \ c \ (\lambda v. Q \ v \ * \ R)$$

Conséquence :

$$\frac{\forall v, Q \ v \Rightarrow Q' \ v}{wp \ c \ Q \Rightarrow wp \ c \ Q'}$$

Ramification :

$$wp \ c \ Q \ * \ (\forall v, Q \ v \ \rightarrow \ * \ Q' \ v) \Rightarrow wp \ c \ Q'$$

Calculer les plus faibles préconditions

En appliquant la ramification au calcul de wp des constructions impératives, on obtient des équations utilisables pour toute postcondition Q .

$$\forall l, (l \mapsto _ \star \dots \star l + N - 1 \mapsto _) \dashv\star Q \Rightarrow wp(\text{alloc}(N)) Q$$

$$\exists x, \llbracket a \rrbracket \mapsto x \star (\llbracket a \rrbracket \mapsto x \dashv\star Q x) \Rightarrow wp(\text{get}(a)) Q$$

$$\llbracket a \rrbracket \mapsto _ \star (\forall v, \llbracket a \rrbracket \mapsto \llbracket a' \rrbracket \dashv\star Q v) \Rightarrow wp(\text{set}(a, a')) Q$$

$$(\llbracket a \rrbracket \mapsto _) \star (\forall v, Q v) \Rightarrow wp(\text{free}(a)) Q$$

Un avant-goût du «mode de preuve Iris» (*Iris proof mode*)

Dans les assistants à la démonstration comme Coq, on préfère manipuler des contextes de preuve

$$\frac{x_1 \dots x_n \quad H_1 \dots H_m}{P}$$

plutôt que des formules $\forall x_1, \dots, x_n, H_1 \wedge \dots \wedge H_m \Rightarrow P$.

Un avant-goût du «mode de preuve Iris» (*Iris proof mode*)

Dans les assistants à la démonstration comme Coq, on préfère manipuler des contextes de preuve

$$\frac{x_1 \dots x_n \quad H_1 \dots H_m}{P}$$

plutôt que des formules $\forall x_1, \dots, x_n, H_1 \wedge \dots \wedge H_m \Rightarrow P$.

L'équivalent en logique de séparation est :

$$\frac{x_1 \dots x_n \quad H_1 \dots H_m \quad (\text{hypothèses usuelles})}{\frac{P_1 \dots P_k \quad (\text{hypothèses spatiales})}{Q} \quad (\text{but})}$$

qui représente $\forall x_i, H_1 \wedge \dots \wedge H_m \Rightarrow P_1 \star \dots \star P_k \Rightarrow Q$.

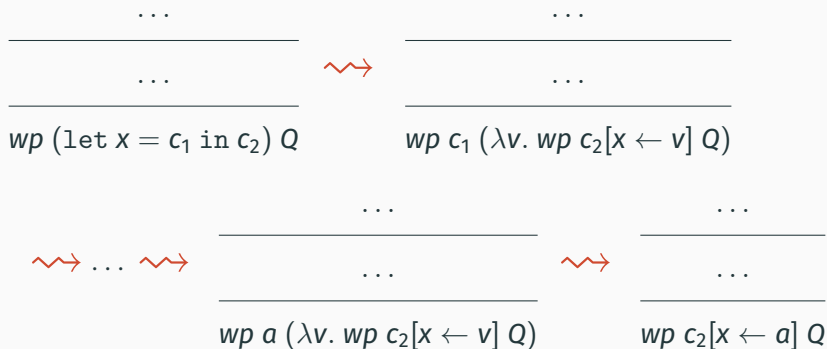
Quelques règles d'introduction

$$\frac{\frac{\dots}{\dots}}{P_1 * \dots * P_n \rightarrow Q} \rightsquigarrow \frac{\dots}{\dots \frac{P_1 \dots P_n}{Q}}$$

$$\frac{\frac{\dots}{\dots \langle H \rangle}}{\dots} \rightsquigarrow \frac{\dots H}{\dots}$$
$$\frac{\dots}{\dots \exists x, P x} \rightsquigarrow \frac{\dots x}{\dots P x}$$

Plus faibles préconditions \approx exécution symbolique

$wp\ c\ Q \approx$ «on fait c puis on aura Q ». La postcondition Q joue le rôle de continuation, mémorisant la suite de l'exécution.



Exécution symbolique des opérations sur la mémoire

Les règles de wp pour les opérations sur la mémoire deviennent plus compréhensibles :

$$\exists x, \llbracket a \rrbracket \mapsto x \star (\llbracket a \rrbracket \mapsto x \rightarrow \star Q x) \Rightarrow wp(\text{get}(a)) Q$$

$$\frac{\dots}{\llbracket a \rrbracket \mapsto x \dots} \rightsquigarrow \frac{\dots}{\llbracket a \rrbracket \mapsto x \dots}$$
$$\frac{\dots}{\llbracket a \rrbracket \mapsto x \dots} \rightsquigarrow \frac{\dots}{Q x}$$

$$\llbracket a \rrbracket \mapsto _ \star (\forall v, \llbracket a \rrbracket \mapsto \llbracket a' \rrbracket \rightarrow \star Q v) \Rightarrow wp(\text{set}(a, a')) Q$$

$$\frac{\dots}{\llbracket a \rrbracket \mapsto _ \dots} \rightsquigarrow \frac{\dots \quad v}{\llbracket a \rrbracket \mapsto \llbracket a' \rrbracket \dots}$$
$$\frac{\dots}{\llbracket a \rrbracket \mapsto _ \dots} \rightsquigarrow \frac{\dots \quad v}{Q v}$$

Exécution symbolique des opérations sur la mémoire

$\forall l, (l \mapsto _ * \dots * l + N - 1 \mapsto _) \multimap Q \ell \Rightarrow wp(\text{alloc}(N)) Q$

$$\frac{\dots}{\dots} \rightsquigarrow \frac{\dots \quad l \quad \dots \quad l + N - 1}{\dots \quad l \mapsto _ \quad \dots \quad l + N - 1 \mapsto _}$$
$$\frac{}{wp(\text{alloc}(N)) Q} \qquad \frac{}{Q \ell}$$

$(\llbracket a \rrbracket \mapsto _) \multimap (\forall v, Q v) \Rightarrow wp(\text{free}(a)) Q$

$$\frac{\dots}{\llbracket a \rrbracket \mapsto _ \quad \dots} \rightsquigarrow \frac{\dots \quad v}{\dots}$$
$$\frac{}{wp(\text{free}(a)) Q} \qquad \frac{}{Q v}$$

Permissions partielles

Le partage en lecture seule

Plusieurs processus accèdent sans synchronisation à une structure de données partagée, mais ne la modifient pas.

$$x := \dots T[i] \dots \parallel y := \dots T[i] \dots \parallel z := \dots T[i] \dots$$

C'est sans risques :

- Pas de course critique (deux lectures simultanées à la même adresse produisent un résultat parfaitement déterminé).
- Cf. le principe Rust : «partage ou-exclusif mutation».

Le partage en lecture seule

Plusieurs processus accèdent sans synchronisation à une structure de données partagée, mais ne la modifient pas.

$$x := \dots T[i] \dots \parallel y := \dots T[i] \dots \parallel z := \dots T[i] \dots$$

C'est efficace :

- Pas besoin de dupliquer la structure dans chaque processus. (\neq parallélisme à mémoire distribuée).

Le partage en lecture seule

Plusieurs processus accèdent sans synchronisation à une structure de données partagée, mais ne la modifient pas.

$$x := \dots T[i] \dots \parallel y := \dots T[i] \dots \parallel z := \dots T[i] \dots$$

Ce n'est pas exprimable dans notre logique de séparation!

- Hors section critique ou atomique, une case mémoire est accessible (en écriture comme en lecture) par un seul processus.

$$\frac{\{P_1\} c_1 \{ \lambda_{-}. Q_1 \} \quad \{P_2\} c_2 \{ \lambda_{-}. Q_2 \}}{\{P_1 \star P_2\} c_1 \parallel c_2 \{ \lambda_{-}. Q_1 \star Q_2 \}}$$

- Cf. le théorème d'absence de courses critiques du cours 4.

Un modèle de permissions

L'assertion $\ell \mapsto v$, «la case mémoire ℓ contient la valeur v », peut aussi être lue comme une permission d'accéder à la case ℓ (lecture, écriture, ou libération).

Idée naïve : distinguer deux permissions

Permission complète : $\ell \xrightarrow{1} v$ (lire, écrire, libérer)

Permission en lecture seule : $\ell \xrightarrow{R} v$ (uniquement lire)

Un modèle de permissions

La «petite règle» pour get accepte les deux permissions. Les autres produisent ou exigent des permissions complètes.

$$\begin{array}{lll} \{ \text{emp} \} & \text{alloc}(N) & \{ \lambda l. l \stackrel{1}{\mapsto} _ \star \dots \star l + N - 1 \stackrel{1}{\mapsto} _ \} \\ \{ \llbracket a \rrbracket \stackrel{\pi}{\mapsto} x \} & \text{get}(a) & \{ \lambda v. \langle v = x \rangle \star \llbracket a \rrbracket \stackrel{\pi}{\mapsto} x \} \quad (\pi \in \{1, R\}) \\ \{ \llbracket a \rrbracket \stackrel{1}{\mapsto} _ \} & \text{set}(a, a') & \{ \lambda v. \llbracket a \rrbracket \stackrel{1}{\mapsto} \llbracket a' \rrbracket \} \\ \{ \llbracket a \rrbracket \stackrel{1}{\mapsto} _ \} & \text{free}(a) & \{ \lambda v. \text{emp} \} \end{array}$$

Une permission complète peut être **affaiblie** :

$$l \stackrel{1}{\mapsto} v \Rightarrow l \stackrel{R}{\mapsto} v$$

Une permission en lecture seule peut être **dupliquée** :

$$l \stackrel{R}{\mapsto} v = l \stackrel{R}{\mapsto} v \star l \stackrel{R}{\mapsto} v$$

Exemple de partage en lecture seule

```
let t = alloc(1) in
```

```
  set(t, f(x));
```

$$\{t \xrightarrow{1} f(x)\} \Rightarrow \{t \xrightarrow{R} f(x)\} \Rightarrow \{t \xrightarrow{R} f(x) * t \xrightarrow{R} f(x)\}$$

$\{t \xrightarrow{R} f(x)\}$		$\{t \xrightarrow{R} f(x)\}$
...get(t)...		...get(t)...
...		...
$\{t \xrightarrow{R} f(x) * Q_1\}$		$\{t \xrightarrow{R} f(x) * Q_2\}$

$$\{t \xrightarrow{R} f(x) * t \xrightarrow{R} f(x) * Q_1 * Q_2\}$$

Exemple de partage en lecture seule

```
let t = alloc(1) in
```

```
  set(t, f(x));
```

$$\{t \xrightarrow{1} f(x)\} \Rightarrow \{t \xrightarrow{R} f(x)\} \Rightarrow \{t \xrightarrow{R} f(x) * t \xrightarrow{R} f(x)\}$$

$$\begin{array}{c} \{t \xrightarrow{R} f(x)\} \\ \dots \text{get}(t) \dots \\ \dots \\ \{t \xrightarrow{R} f(x) * Q_1\} \end{array} \quad \parallel \quad \begin{array}{c} \{t \xrightarrow{R} f(x)\} \\ \dots \text{get}(t) \dots \\ \dots \\ \{t \xrightarrow{R} f(x) * Q_2\} \end{array}$$

$$\{t \xrightarrow{R} f(x) * t \xrightarrow{R} f(x) * Q_1 * Q_2\}$$

Problème : on ne peut plus libérer t !

Les permissions fractionnaires

$$\frac{1}{2} + \frac{1}{2} = 1$$

Boyland (2003) : les permissions π sont des rationnels dans $]0, 1]$.

- $\pi = 1$: permission complète.
- $0 < \pi < 1$: permission en lecture seule.

Une loi qui régit le partage et la recombinaison de permissions :

$$\ell \xrightarrow{\pi+\pi'} v = \ell \xrightarrow{\pi} v \star \ell \xrightarrow{\pi'} v \quad \text{si } \pi + \pi' \in]0, 1]$$

Partage en lecture seule avec permissions fractionnaires

```
let t = alloc(1) in  
set(t, f(x));
```

$$\{t \xrightarrow{1} f(x)\} \Rightarrow \{t \xrightarrow{1/2} f(x) * t \xrightarrow{1/2} f(x)\}$$

$$\begin{array}{c} \{t \xrightarrow{1/2} f(x)\} \\ \dots \text{get}(t) \dots \\ \dots \\ \{t \xrightarrow{1/2} f(x) * Q_1\} \end{array} \parallel \begin{array}{c} \{t \xrightarrow{1/2} f(x)\} \\ \dots \text{get}(t) \dots \\ \dots \\ \{t \xrightarrow{1/2} f(x) * Q_2\} \end{array}$$

$$\{t \xrightarrow{1/2} f(x) * t \xrightarrow{1/2} f(x) * Q_1 * Q_2\} \Rightarrow \{t \xrightarrow{1} f(x) * Q_1 * Q_2\}$$

`free(t)`

$$\{Q_1 * Q_2\}$$

Un ensemble Π muni d'une opération **partielle** \oplus de combinaison de deux permissions, qui est

commutative

$$\pi_1 \oplus \pi_2 = \pi_2 \oplus \pi_1$$

et **associative**

$$(\pi_1 \oplus \pi_2) \oplus \pi_3 = \pi_1 \oplus (\pi_2 \oplus \pi_3).$$

N.B. : il faut lire $\pi_1 \oplus \pi_2 = \pi$ comme «la combinaison $\pi_1 \oplus \pi_2$ est définie et égale à π ».

Un état mémoire h est une fonction finie des adresses dans les paires (π, v) d'une permission et d'une valeur.

On définit la combinaison de deux telles paires comme :

$$(\pi_1, v_1) \oplus (\pi_2, v_2) = (\pi_1 \oplus \pi_2, v_1) \text{ si } v_1 = v_2 \text{ et } \pi_1 \oplus \pi_2 \text{ est définie}$$

États mémoires avec permissions

La combinaison $h_1 \oplus h_2$ de deux états mémoire est définie si

$h_1 \perp h_2 \stackrel{\text{def}}{=} \forall \ell \in \text{Dom}(h_1) \cap \text{Dom}(h_2), h_1(\ell) \oplus h_2(\ell)$ est définie

La combinaison est caractérisée par

$$\text{Dom}(h_1 \oplus h_2) = \text{Dom}(h_1) \cup \text{Dom}(h_2)$$

$$(h_1 \oplus h_2)(\ell) = \begin{cases} h_1(\ell) \oplus h_2(\ell) & \text{si } \ell \in \text{Dom}(h_1) \cap \text{Dom}(h_2) \\ h_1(\ell) & \text{si } \ell \in \text{Dom}(h_1) \setminus \text{Dom}(h_2) \\ h_2(\ell) & \text{si } \ell \in \text{Dom}(h_2) \setminus \text{Dom}(h_1) \end{cases}$$

Généralise la notion d'union disjointe $h_1 \uplus h_2$.

Conjonction et implication séparantes

Les définitions usuelles où la combinaison d'états mémoire \oplus remplace l'union disjointe \uplus .

$$P \star Q = \lambda h. \exists h_1, h_2, h = h_1 \oplus h_2 \wedge P h_1 \wedge Q h_2$$

$$P \rightarrow \star Q = \lambda h. \exists h_1, h_2, h_2 = h \oplus h_1 \wedge P h_1 \wedge Q h_2$$

En particulier, on a $\ell \xrightarrow{\pi_1} v_1 \star \ell \xrightarrow{\pi_2} v_2$ si et seulement si $\pi_1 \oplus \pi_2$ est défini, $v_1 = v_2$, et $\ell \xrightarrow{\pi_1 \oplus \pi_2} v_1$.

Une propriété intéressante : toute commande c prouvable avec la précondition «permission en lecture seule sur l'adresse l » ne peut pas modifier l .

Esquisse de démonstration dans le cas séquentiel :

Par l'absurde, supposons $\{l \xrightarrow{1/2} v\} c \{ \lambda.. l \xrightarrow{1/2} v' \}$ avec $v' \neq v$.

Par encadrement avec $l \xrightarrow{1/2} v$, on obtient

$$\{l \xrightarrow{1} v\} c \{ \lambda.. l \xrightarrow{1/2} v' \star l \xrightarrow{1/2} v \}$$

La précondition est satisfiable mais la postcondition est toujours fausse, car l'adresse l ne peut pas contenir à la fois v et v' .

En présence de sections atomiques ou de sections critiques, la propriété de passivité est moins claire.

En effet, si l'invariant sur la mémoire partagée nous donne la permission $\ell \stackrel{1/2}{\mapsto} _$ manquante, on peut dériver

$$\ell \stackrel{1/2}{\mapsto} _ \vdash \{ \ell \stackrel{1/2}{\mapsto} v \} \text{atomic}(\text{set}(\ell, v')) \{ \lambda _. \ell \stackrel{1/2}{\mapsto} v' \}$$

même si $v' \neq v$.

Partage en lecture seule avec permissions comptées

Un autre schéma de permissions, adapté aux programmes utilisant des verrous à lecteurs multiples (*readers-writer locks*) :

Les permissions π sont des entiers ≥ -1 :

- 0 : permission complète (get, set, alloc, free)
- -1 : permission en lecture seule (get)
- $n > 0$: nombre de permissions en lecture seule accordées.

On a :

$$l \xrightarrow{n} v = l \xrightarrow{n+1} v \star l \xrightarrow{-1} v \quad \text{si } n \geq 0$$

	Lecteurs	Écrivain
$\{ \text{emp} \}$	$P(\text{read});$ $\text{count} := \text{count} + 1;$ $\text{if } \text{count} = 1 \text{ then } P(\text{write});$ $V(\text{read});$	
$\{ b \stackrel{-1}{\mapsto} - \}$	lire b	$\{ \text{emp} \}$ $P(\text{write});$
$\{ b \stackrel{-1}{\mapsto} - \}$	$P(\text{read});$ $\text{count} := \text{count} - 1;$ $\text{if } \text{count} = 0 \text{ then } V(\text{write});$ $V(\text{read});$	$\{ b \stackrel{0}{\mapsto} - \}$ écrire b
$\{ \text{emp} \}$		$V(\text{write});$ $\{ \text{emp} \}$

Invariant de $\text{write} : b \stackrel{0}{\mapsto} -$

Invariant de $\text{read} : \exists n, \text{count} \stackrel{0}{\mapsto} n \star (\langle n = 0 \rangle \vee \langle n > 0 \rangle \star b \stackrel{n}{\mapsto} -)$

Code fantôme

Variables auxiliaires, variables fantômes (rappel du 2^e cours)

Deux manières de faciliter l'écriture de spécifications sous forme de triplets de Hoare.

Variables auxiliaires : des variables mathématiques α, β, \dots quantifiées universellement en tête du triplet.

$$\forall \alpha, \beta, \{ x = \alpha \wedge y = \beta \} \text{ if } x < y \text{ then } x := y \{ x = \max(\alpha, \beta) \}$$

Variables fantômes : des variables du langage de programmation qui n'apparaissent pas dans le programme.

$$\{ z = x \} \text{ if } x < y \text{ then } x := y \{ x = \max(z, y) \}$$



Pour faciliter la vérification, on peut ajouter du **code fantôme** : des commandes qui modifient les variables fantômes mais n'ont pas d'effet sur les variables normales.

Ce code fantôme peut être éliminé avant exécution, puisque le code normal ne dépend pas des variables fantômes.

Exemple : le reste de la division euclidienne

```

                                {  $a \geq 0$  }
r := a;

while r  $\geq$  b do
                                {  $r \geq 0 \wedge \exists q, a = b \cdot q + r$  }

    r := r - b
done
                                {  $r = a \bmod b$  }
```

Les démonstrateurs automatiques ont souvent des problèmes avec les quantificateurs existentiels....

Exemple : le reste de la division euclidienne

```

                                { a ≥ 0 }
r := a;
👻 q := 0;
while r ≥ b do
                                { r ≥ 0 ∧ a = b · q + r }
    👻 q := q + 1;
    r := r - b
done
                                { r = a mod b }
```

Le code fantôme calcule «la bonne valeur» de q .
Le démonstrateur automatique n'a plus qu'à la vérifier.

Exemple : un parcours de graphe récursif

Comme au 3^e cours : marquer tous les nœuds atteignables depuis la racine r .

```
def DFS r =  
  if MARK[r] = 0 then begin  
    MARK[r] := 1;  
    for i = 0 to ARITY[r] - 1 do DFS(CHILD[r][i]) done  
  end
```

Code très difficile à montrer correct!

Exemple : un parcours de graphe récursif

On réintroduit la *worklist* W sous forme de variable fantôme.
($W \approx$ les nœuds qui restent à parcourir)

```
def DFSREC  $p$  =  
  👻  $W := W \setminus \{p\}$ ;  
  if MARK[ $p$ ] = 0 then begin  
    MARK[ $p$ ] := 1;  
    👻  $W := W \cup \{\text{CHILD}[p][i] \mid 0 \leq i < \text{ARITY}[p]\}$ ;  
    for  $i = 0$  to ARITY[ $p$ ] - 1 do DFS (CHILD[ $p$ ][ $i$ ]) done  
  end  
def DFS  $r$  =  
  👻  $W := \{r\}$ ;  
  DFSREC  $r$ 
```

Exemple : un parcours de graphe récursif

On peut alors montrer comme invariant

$$\forall x, \text{path}(r, x) \iff \text{MARK}[x] = 1 \vee \exists p \in W, \text{path}(p, x)$$

et conclure

$$\{ \forall x, \text{MARK}[x] = 0 \} \text{ DFS } r \{ \forall x, \text{path}(r, x) \iff \text{MARK}[x] = 1 \}$$

À retenir : le code fantôme n'est pas forcément exécutable, et les variables fantômes ne sont pas forcément d'un des types exprimables dans le langage de programmation!

Ici, on a utilisé des ensembles mathématiques pour W .

Code fantôme et parallélisme

Dans un programme parallèle, on peut utiliser des variables fantômes et du code fantôme pour garder une trace des actions de chaque processus.

Exemple : le schéma producteur/consommateur.

```

                                👻  $PR := \epsilon; CO := \epsilon;$ 
while true do                   || while true do
  calculer x;                   ||   let  $y = consume()$  in
                                ||   👻  $CO := CO \cdot y$ 
                                ||   utiliser  $y$ 
                                || done
  👻  $PR := PR \cdot x$            ||
  produce(x);                   ||
done                             || done
```

Les listes fantômes PR et CO gardent trace des données produites ou consommées.

Le puzzle : $1 + 1 = 2$?

```
set(n, 0);  
atomic(incr(n)) || atomic(incr(n))
```

Avec $incr(p) \stackrel{def}{=} \text{let } x = \text{get}(p) \text{ in set}(p, x + 1)$.

Au précédent cours, on a vu comment montrer la sûreté de ce code et le fait que $n \geq 0$ à la fin, grâce à l'invariant de ressource $J = \exists x, n \mapsto x * \langle x \geq 0 \rangle$.

Mais comment montrer la correction? c.à.d. que $n = 2$ à la fin?

Tracer les processus avec du code fantôme

```
set(n, 0);
```

```
 set(a, 0); set(b, 0);
```

```
atomic(incr(n);  incr(a)) || atomic(incr(n);  incr(b))
```

a représente la contribution du processus gauche à la somme n ,
 b celle du processus droit.

Tracer les processus avec du code fantôme

```
set(n, 0);  
👻 set(a, 0); set(b, 0);  
atomic(incr(n); 👻 incr(a)) || atomic(incr(n); 👻 incr(b))
```

a représente la contribution du processus gauche à la somme n ,
 b celle du processus droit.

On voudrait refléter ça dans l'invariant :

$$\exists x, y, a \mapsto x \star b \mapsto y \star n \mapsto x + y.$$

Mais il faudrait que a et b appartiennent à l'état partagé.

On voudrait montrer $a \mapsto 1$ et $b \mapsto 1$ à la fin.

Mais il faudrait que a (resp. b) appartienne au processus de gauche (resp. droite).

Les permissions fractionnaires à la rescousse

Prenons comme invariant de ressource

$$J = \exists x, y, a \stackrel{1/2}{\mapsto} x \star b \stackrel{1/2}{\mapsto} y \star n \stackrel{1}{\mapsto} x + y$$

On a :

$$\{J \star a \stackrel{1/2}{\mapsto} x\} \Rightarrow \\ \{a \stackrel{1}{\mapsto} x \star \exists y, b \stackrel{1/2}{\mapsto} y \star n \stackrel{1}{\mapsto} x + y\}$$

incr(n);

incr(a);

$$\{a \stackrel{1}{\mapsto} x + 1 \star \exists y, b \stackrel{1/2}{\mapsto} y \star n \stackrel{1}{\mapsto} x + 1 + y\} \\ \Rightarrow \{J \star a \stackrel{1/2}{\mapsto} x + 1\}$$

Et donc : $J \vdash \{a \stackrel{1/2}{\mapsto} x\} \text{atomic}(\text{incr}(n); \text{incr}(a)) \{a \stackrel{1/2}{\mapsto} x + 1\}$

Les permissions fractionnaires à la rescousse

On peut alors dériver :

$$\begin{array}{c} \text{set}(n, 0); \text{set}(a, 0); \text{set}(b, 0); \\ \{n \xrightarrow{1} 0 * a \xrightarrow{1} 0 * b \xrightarrow{1} 0\} \Rightarrow \{J * a \xrightarrow{1/2} 0 * b \xrightarrow{1/2} 0\} \\ \left\{ \begin{array}{l} \{a \xrightarrow{1/2} 0\} \\ \text{atomic}(\text{incr}(n); \text{incr}(a)) \\ \{a \xrightarrow{1/2} 1\} \end{array} \right\} \parallel \left\{ \begin{array}{l} \{b \xrightarrow{1/2} 0\} \\ \text{atomic}(\text{incr}(n); \text{incr}(b)) \\ \{b \xrightarrow{1/2} 1\} \end{array} \right\} \\ \{J * a \xrightarrow{1/2} 1 * b \xrightarrow{1/2} 1\} \Rightarrow \{n \xrightarrow{1} 2 * a \xrightarrow{1} 1 * b \xrightarrow{1} 1\} \end{array}$$

Et donc $n = 2$ à la fin!

Les permissions fractionnaires à la rescousse

On peut alors dériver :

$$\begin{array}{c} \text{set}(n, 0); \text{set}(a, 0); \text{set}(b, 0); \\ \{n \xrightarrow{1} 0 * a \xrightarrow{1} 0 * b \xrightarrow{1} 0\} \Rightarrow \{J * a \xrightarrow{1/2} 0 * b \xrightarrow{1/2} 0\} \\ \{a \xrightarrow{1/2} 0\} \quad \left\| \quad \{b \xrightarrow{1/2} 0\} \right. \\ \text{atomic}(\text{incr}(n); \text{incr}(a)) \quad \left\| \quad \text{atomic}(\text{incr}(n); \text{incr}(b)) \right. \\ \{a \xrightarrow{1/2} 1\} \quad \left\| \quad \{b \xrightarrow{1/2} 1\} \right. \\ \{J * a \xrightarrow{1/2} 1 * b \xrightarrow{1/2} 1\} \Rightarrow \{n \xrightarrow{1} 2 * a \xrightarrow{1} 1 * b \xrightarrow{1} 1\} \end{array}$$

Et donc $n = 2$ à la fin!

Ceci est un exemple élémentaire d'une technique très générale :
les protocoles d'évolution de l'état fantôme

→ séminaire de J. H. Jourdan le 2021-04-08.

Verrous stockés en mémoire

Deux types d'exclusion mutuelle :

- **À gros grain** : un verrou (global) qui protège une structure de données toute entière.

Bien décrit par le modèle de ressources de la logique de séparation concurrente de O'Hearn.

- **À grain fin** : un verrou par bloc mémoire composant la structure de données.

Besoin de raisonner sur des verrous qui sont stockés en mémoire dans le bloc qu'ils protègent.

Exemple : liste simplement chaînée

```
struct cell { lock lck; int val; struct cell * next; };
```

En verrouillant les nœuds les uns après les autres, on va pouvoir effectuer des opérations sur la liste en parallèle.

Exemple : un processus enlève «2», l'autre enlève «5».



Exemple : liste simplement chaînée

```
struct cell { lock lck; int val; struct cell * next; };
```

En verrouillant les nœuds les uns après les autres, on va pouvoir effectuer des opérations sur la liste en parallèle.

Exemple : un processus enlève «2», l'autre enlève «5».

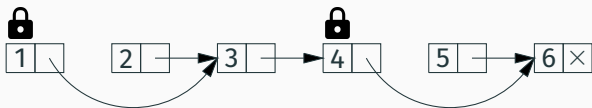


Exemple : liste simplement chaînée

```
struct cell { lock lck; int val; struct cell * next; };
```

En verrouillant les nœuds les uns après les autres, on va pouvoir effectuer des opérations sur la liste en parallèle.

Exemple : un processus enlève «2», l'autre enlève «5».

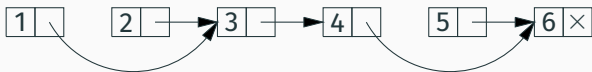


Exemple : liste simplement chaînée

```
struct cell { lock lck; int val; struct cell * next; };
```

En verrouillant les nœuds les uns après les autres, on va pouvoir effectuer des opérations sur la liste en parallèle.

Exemple : un processus enlève «2», l'autre enlève «5».



Exemple : liste simplement chaînée

```
struct cell { lock lck; int val; struct cell * next; };
```

En verrouillant les nœuds les uns après les autres, on va pouvoir effectuer des opérations sur la liste en parallèle.

Exemple : un processus enlève «2», l'autre enlève «5».



Verrouiller un seul nœud à la fois ne suffit pas!

Exemple : un processus enlève «3», l'autre enlève «4».

Exemple : liste simplement chaînée

```
struct cell { lock lck; int val; struct cell * next; };
```

En verrouillant les nœuds les uns après les autres, on va pouvoir effectuer des opérations sur la liste en parallèle.

Exemple : un processus enlève «2», l'autre enlève «5».



Verrouiller un seul nœud à la fois ne suffit pas!

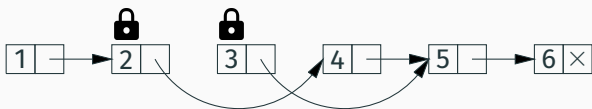
Exemple : un processus enlève «3», l'autre enlève «4».

Exemple : liste simplement chaînée

```
struct cell { lock lck; int val; struct cell * next; };
```

En verrouillant les nœuds les uns après les autres, on va pouvoir effectuer des opérations sur la liste en parallèle.

Exemple : un processus enlève «2», l'autre enlève «5».



Verrouiller un seul nœud à la fois ne suffit pas!

Exemple : un processus enlève «3», l'autre enlève «4».

Le résultat peut être [1; 2; 4; 5; 6] au lieu de [1; 2; 5; 6].

Verrouillage couplé (*hand-over-hand locking*)

Pour modifier un nœud, il faut l'avoir verrouillé **ainsi que son prédécesseur**.

Exemple : destruction de «4».



Verrouillage couplé (*hand-over-hand locking*)

Pour modifier un nœud, il faut l'avoir verrouillé **ainsi que son prédécesseur**.

Exemple : destruction de «4».



Verrouillage couplé (*hand-over-hand locking*)

Pour modifier un nœud, il faut l'avoir verrouillé **ainsi que son prédécesseur**.

Exemple : destruction de «4».



Verrouillage couplé (*hand-over-hand locking*)

Pour modifier un nœud, il faut l'avoir verrouillé **ainsi que son prédécesseur**.

Exemple : destruction de «4».



Verrouillage couplé (*hand-over-hand locking*)

Pour modifier un nœud, il faut l'avoir verrouillé **ainsi que son prédécesseur**.

Exemple : destruction de «4».



Verrouillage couplé (*hand-over-hand locking*)

Pour modifier un nœud, il faut l'avoir verrouillé **ainsi que son prédécesseur**.

Exemple : destruction de «4».



Verrouillage couplé (*hand-over-hand locking*)

Pour modifier un nœud, il faut l'avoir verrouillé **ainsi que son prédécesseur**.

Exemple : destruction de «4».



Spécification des verrous stockés en mémoire

Deux nouvelles assertions :

$l \bullet \xrightarrow{\pi} RI$ «à l'adresse l , avec permission π , il y a un verrou qui protège la ressource décrite par l'invariant RI »

$\text{lock}(l)$ «le verrou à l'adresse l est verrouillé par le processus courant»

Les «petites règles» pour les verrous :

$\{ l \bullet \xrightarrow{\pi} RI \} \quad \text{lock}(l) \quad \{ l \bullet \xrightarrow{\pi} RI * \text{lock}(l) * RI \}$

$\{ l \bullet \xrightarrow{\pi} RI * \text{lock}(l) * RI \} \quad \text{unlock}(l) \quad \{ l \bullet \xrightarrow{\pi} RI \}$

$\{ l \xrightarrow{1} _ * RI \} \quad \text{initlock}(l) \quad \{ l \bullet \xrightarrow{1} RI \}$

$\{ l \bullet \xrightarrow{1} RI \} \quad \text{destroylock}(l) \quad \{ l \xrightarrow{1} _ * RI \}$

Prédicat de représentation pour les listes triées

(Selon Gotsman, Berdine, Cook, Rinetzky et Sagiv, 2007.

Voir Jacobs et Piessens, 2011, pour une spécification plus fine.)

On ajoute une sentinelle $-\infty$ au début et une $+\infty$ à la fin.

$$\begin{aligned} list(p, n) = & (\langle n = +\infty \rangle \star p.val \xrightarrow{1} n \star p.next \xrightarrow{1} \text{NULL}) \\ & \vee (\exists q, n', \langle n < n' \rangle \star p.val \xrightarrow{1} n \star p.next \xrightarrow{1} q \\ & \star p.lock \bullet \xrightarrow{1} list(q, n')) \end{aligned}$$

$$\begin{aligned} listhead(p, \pi) = & \exists q, n, p.val \xrightarrow{\pi} -\infty \star p.next \xrightarrow{\pi} q \\ & \star p.lock \bullet \xrightarrow{\pi} list(q, n) \end{aligned}$$

La tête de liste (la sentinelle $-\infty$) est partageable ($\pi < 1$). Les autres nœuds de la liste sont en accès exclusif, protégés par le verrou du nœud prédécesseur.

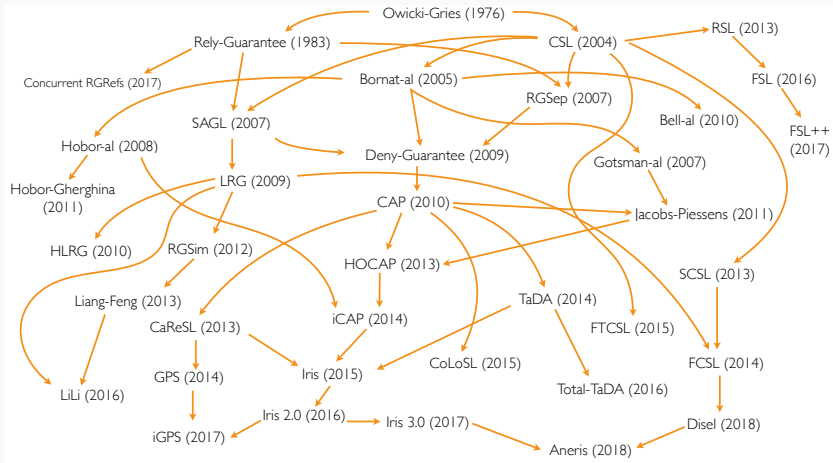
Point d'étape

Nous avons vu quelques extensions de la logique de séparation, séquentielle ou concurrente.

Bien d'autres extensions ont été étudiées dans les 20 dernières années :

- « X de première classe» pour diverses valeurs de X : fonctions, triplets de Hoare, identifiants de processus, ...
- Modularisation du raisonnement, p.ex. interactions entre un nombre arbitraire de processus au lieu de seulement 2.
- Vérification d'algorithmes concurrents avancés : verrouillage optimiste, sans verrouillage (*lock-free*), etc.

Un foisonnement de logiques de programmes



(Diagramme : Ilya Sergey)

Des assertions qui parlent de beaucoup de choses

- Des faits purement logiques $\langle P \rangle$
- Des faits à propos des variables $x = \alpha$ (en logique de Hoare)
- Des faits à propos du tas mémoire $\text{emp}, \ell \mapsto v, \ell \mapsto _$
- Même chose plus des permissions $\ell \overset{\pi}{\mapsto} v$
- Des faits à propos des verrous $\ell \overset{\pi}{\bullet\mapsto} RI, \text{🔒} \ell$
- Des faits à propos de l'état fantôme.
- Le temps disponible $(\rightarrow$ séminaire de F. Pottier)
- Des systèmes de transition $(\rightarrow$ séminaire de J. H. Jourdan)
- Et puis quoi encore ?

Iris : une consolidation autour de quatre notions

1– Les **algèbres de ressources**, auparavant appelées *partial commutative monoids*.

(Une opération \oplus commutative, associative, **partielle**, représentant la combinaison de deux «choses» compatibles.)

2– Les **transitions fantômes**, généralisant le code fantôme.

3– Les **invariants**, généralisant les différents types d'invariants de ressources que nous avons vu.

4– Une utilisation systématique du **comptage de pas** et de la **modalité «plus tard»** (\triangleright) pour contourner les circularités dans les notions d'ordre supérieur.

Bibliographie

Tout savoir sur Iris :

- Le séminaire de J.-H. Jourdan le 2021-04-08.
- Articles, tutoriels, développement Coq :
<https://iris-project.org/>

Permissions partielles :

- R. Bornat, C. Calcagno, P. O'Hearn, M. Parkinson, *Permission accounting in separation logic*, POPL 2005.

Verrous stockés en mémoire :

- A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, M. Sagiv, *Local reasoning for storable locks and threads*, APLAS 2007.