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

dans un contexte «plus grand», par exemple

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

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 $\{ P \}\ c\ \{ Q \}$.

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

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.

Abduction et baguette magique

Le problème de l'abduction

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

(En d'autre 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 $P \mathrel{-\!\!\ast} Q$ et on l'appelle «baguette magique» (*magic wand*) :

$$P \mathrel{-\!\!\ast} Q \stackrel{def}{=} \lambda h.\ \forall h' \perp h, P\ h' \Rightarrow Q(h' \uplus h)$$

Baguette magique = implication séparante

L'implication séparante $\mathrel{-\!\!*}$ est à la conjonction séparante $\star$ ce que l'implication usuelle $\Rightarrow$ est à la conjonction usuelle $\wedge$.

Adjonction :

$$H \Rightarrow (P \mathrel{-\!\!*} Q) \quad \Longleftrightarrow \quad H \star P \Rightarrow Q$$

Autres propriétés :

$$\begin{array}{rcll}
P \star (P \mathrel{-\!\!*} Q) & \Rightarrow & Q & \text{(élimination)} \\
emp & \Rightarrow & P \mathrel{-\!\!*} P & \text{(idempotence)} \\
(P \mathrel{-\!\!*} Q) \star (Q \mathrel{-\!\!*} R) & \Rightarrow & P \mathrel{-\!\!*} R & \text{(transitivité)} \\
(P \star Q) \mathrel{-\!\!*} R & = & P \mathrel{-\!\!*} Q \mathrel{-\!\!*} R & \text{(curryfication)} \\
(P \mathrel{-\!\!*} Q) \star R & \Rightarrow & P \mathrel{-\!\!*} (Q \star R) & \text{(distribution)}
\end{array}$$

La règle de conséquence ramifiée

$$\frac{\{\,P'\,\}\ c\ \{\,Q'\,\} \qquad P \Rightarrow (P' \star (\forall v, Q'\ v \mathrel{-\!\!\star} Q\ v))}{\{\,P\,\}\ c\ \{\,Q\,\}}$$

Comme la règle de conséquence encadrée, mais avec un choix canonique pour le «cadre» : $\quad R = \forall v, Q'\ v \mathrel{-\!\!\star} Q\ v$

Remplace la difficulté de trouver R par la difficulté de raisonner sur des formules utilisant $\mathrel{-\!\!\star}$ 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 : $wp\ c\ Q \;=\; \lambda h.\, \mathtt{Term}\ c\ h\ Q$ (ou $\mathtt{Safe}$)

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

Plus faibles préconditions

Une équivalence avec les triplets :

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

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

« Étant donnés 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 ? »

Calculer les plus faibles préconditions

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

$$
\begin{aligned}
Q\,[\![a]\!] &\Rightarrow wp\ a\ Q \\
wp\ c\ (\lambda v.\ wp\ c'[x \leftarrow v]\ Q) &\Rightarrow wp\ (\texttt{let}\ x = c\ \texttt{in}\ c')\ Q \\
(\text{si } [\![b]\!] \text{ alors } wp\ c_1\ Q \text{ sinon } wp\ c_2\ Q) &\Rightarrow wp\ (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ Q
\end{aligned}
$$

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.

$$\begin{aligned}
\texttt{emp} &\Rightarrow wp\,(\texttt{alloc}(N))\,(\lambda \ell.\, \ell \mapsto _ \star \cdots \star \ell + N - 1 \mapsto _) \\
[\![a]\!] \mapsto x &\Rightarrow wp\,(\texttt{get}(a))\,(\lambda v.\, \langle v = x \rangle \star [\![a]\!] \mapsto x) \\
[\![a]\!] \mapsto _ &\Rightarrow wp\,(\texttt{set}(a, a'))\,(\lambda v.\, [\![a]\!] \mapsto [\![a']\!]) \\
[\![a]\!] \mapsto _ &\Rightarrow wp\,(\texttt{free}(a))\,(\lambda v.\, \texttt{emp})
\end{aligned}$$

C'est le moment de sortir la baguette magique…

Règles structurelles pour les plus faibles préconditions

Encadrement :

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

Conséquence :

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

Ramification :

$$wp\ c\ Q \star (\forall v,\ Q\ v \mathrel{-\!\!\star} Q'\ v) \Rightarrow wp\ c\ Q'$$

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.

$$\begin{array}{rcl} \forall \ell,\ (\ell \mapsto _ \star \cdots \star \ell + N - 1 \mapsto _) \mathrel{-\!\!\star} Q\,\ell & \Rightarrow & wp\,(\texttt{alloc}(N))\,Q \\ \exists x,\ [\![a]\!] \mapsto x \star ([\![a]\!] \mapsto x \mathrel{-\!\!\star} Q\,x) & \Rightarrow & wp\,(\texttt{get}(a))\,Q \\ [\![a]\!] \mapsto _ \star (\forall v,\ [\![a]\!] \mapsto [\![a']\!] \mathrel{-\!\!\star} Q\,v) & \Rightarrow & wp\,(\texttt{set}(a, a'))\,Q \\ ([\![a]\!] \mapsto _) \star (\forall v, Q\,v) & \Rightarrow & wp\,(\texttt{free}(a))\,Q \end{array}$$

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

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

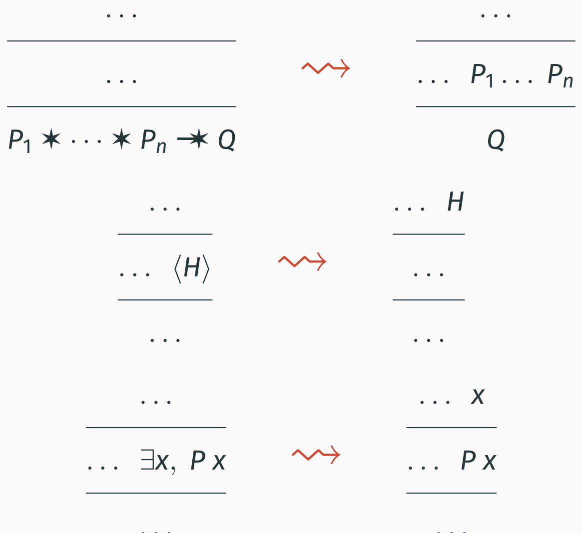
plutôt que des formules $\forall x_1, \dots x_n, H_1 \wedge \dots \wedge H_m \Rightarrow P$.

L'équivalent en logique de séparation est :

$$\begin{array}{cl} x_1 \dots x_n \quad H_1 \dots H_m & \text{(hypothèses usuelles)} \\ \hline P_1 \dots P_k & \text{(hypothèses spatiales)} \\ \hline Q & \text{(but)} \end{array}$$

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

Quelques règles d'introduction



Plus faibles préconditions ≈ exécution symbolique

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

$$\dfrac{\dfrac{\cdots}{\cdots}}{wp\ (\texttt{let}\ x = c_1\ \texttt{in}\ c_2)\ Q} \quad\rightsquigarrow\quad \dfrac{\dfrac{\cdots}{\cdots}}{wp\ c_1\ (\lambda v.\ wp\ c_2[x \leftarrow v]\ Q)}$$

$$\rightsquigarrow \ \cdots \ \rightsquigarrow \quad \dfrac{\dfrac{\cdots}{\cdots}}{wp\ a\ (\lambda v.\ wp\ c_2[x \leftarrow v]\ Q)} \quad\rightsquigarrow\quad \dfrac{\dfrac{\cdots}{\cdots}}{wp\ c_2[x \leftarrow a]\ Q}$$

Exécution symbolique des opérations sur la mémoire

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

$$\exists x,\ [\![a]\!] \mapsto x \star ([\![a]\!] \mapsto x \mathrel{-\!\!\star} Q\,x) \Rightarrow wp\ (\mathtt{get}(a))\ Q$$

$$\dfrac{\dfrac{\dots}{[\![a]\!] \mapsto x \quad \dots}}{wp\ (\mathtt{get}(a))\ Q} \quad \rightsquigarrow \quad \dfrac{\dfrac{\dots}{[\![a]\!] \mapsto x \quad \dots}}{Q\,x}$$

$$[\![a]\!] \mapsto _ \star (\forall v,\ [\![a]\!] \mapsto [\![a']\!] \mathrel{-\!\!\star} Q\,v) \Rightarrow wp\ (\mathtt{set}(a, a'))\ Q$$

$$\dfrac{\dfrac{\dots}{[\![a]\!] \mapsto _ \quad \dots}}{wp\ (\mathtt{set}(a, a'))\ Q} \quad \rightsquigarrow \quad \dfrac{\dfrac{\dots \quad v}{[\![a]\!] \mapsto [\![a']\!] \quad \dots}}{Q\,v}$$

Exécution symbolique des opérations sur la mémoire

$$\forall \ell,\ (\ell \mapsto _ \ast \cdots \ast \ell + N - 1 \mapsto _) \mathrel{-\!\!\ast} Q\,\ell \Rightarrow wp\ (\texttt{alloc}(N))\ Q$$

$$\dfrac{\dfrac{\ldots}{\ldots}}{wp\ (\texttt{alloc}(N))\ Q} \rightsquigarrow \dfrac{\dfrac{\ldots \quad \ell}{\ldots \quad \ell \mapsto _ \quad \ldots \quad \ell + N - 1 \mapsto _}}{Q\,\ell}$$

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

$$\dfrac{\dfrac{\ldots}{\llbracket a \rrbracket \mapsto _ \quad \ldots}}{wp\ (\texttt{free}(a))\ Q} \rightsquigarrow \dfrac{\dfrac{\ldots \quad v}{\ldots}}{Q\ v}$$

Permissions partielles

Le partage en lecture seule

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

$$x := \ldots T[i] \ldots \;\Big\|\; y := \ldots T[i] \ldots \;\Big\|\; z := \ldots T[i] \ldots$$

C'est sans risques :

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

Le partage en lecture seule

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

$$x := \ldots T[i] \ldots \;\Big\|\; y := \ldots T[i] \ldots \;\Big\|\; z := \ldots T[i] \ldots$$

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.

$$x := \dots T[i] \dots \;\Big\|\; y := \dots T[i] \dots \;\Big\|\; z := \dots T[i] \dots$$

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.

$$\frac{\{\, P_1 \,\}\; c_1 \;\{\, \lambda_.\; Q_1 \,\} \qquad \{\, P_2 \,\}\; c_2 \;\{\, \lambda_.\; Q_2 \,\}}{\{\, P_1 \star P_2 \,\}\; c_1 \parallel c_2 \;\{\, \lambda_.\; Q_1 \star Q_2 \,\}}$$

- 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 :	$\ell \overset{1}{\mapsto} v$	(lire, écrire, libérer)
Permission en lecture seule :	$\ell \overset{R}{\mapsto} v$	(uniquement lire)

Un modèle de permissions

La ≪petite règle≫ pour `get` accepte les deux permissions. Les autres produisent ou exigent des permissions complètes.

$$
\begin{array}{ccl}
\{\, \mathtt{emp} \,\} & \mathtt{alloc}(N) & \{\, \lambda\ell.\ \ell \overset{1}{\mapsto} _ \star \cdots \star \ell + N - 1 \overset{1}{\mapsto} _ \,\} \\
\{\, [\![a]\!] \overset{\pi}{\mapsto} x \,\} & \mathtt{get}(a) & \{\, \lambda v.\ \langle v = x \rangle \star [\![a]\!] \overset{\pi}{\mapsto} x \,\} \quad (\pi \in \{1, R\}) \\
\{\, [\![a]\!] \overset{1}{\mapsto} _ \,\} & \mathtt{set}(a, a') & \{\, \lambda v.\ [\![a]\!] \overset{1}{\mapsto} [\![a']\!] \,\} \\
\{\, [\![a]\!] \overset{1}{\mapsto} _ \,\} & \mathtt{free}(a) & \{\, \lambda v.\ \mathtt{emp} \,\}
\end{array}
$$

Une permission complète peut être affaiblie :

$$\ell \overset{1}{\mapsto} v \ \Rightarrow\ \ell \overset{R}{\mapsto} v$$

Une permission en lecture seule peut être dupliquée :

$$\ell \overset{R}{\mapsto} v \ =\ \ell \overset{R}{\mapsto} v \star \ell \overset{R}{\mapsto} v$$

Exemple de partage en lecture seule

$$\texttt{let } t = \texttt{alloc}(1) \texttt{ in}$$
$$\texttt{set}(t, f(x));$$
$$\{\, t \overset{1}{\mapsto} f(x) \,\} \Rightarrow \{\, t \overset{R}{\mapsto} f(x) \,\} \Rightarrow \{\, t \overset{R}{\mapsto} f(x) \ast t \overset{R}{\mapsto} f(x) \,\}$$

$$\begin{array}{l||l} \{\, t \overset{R}{\mapsto} f(x) \,\} & \{\, t \overset{R}{\mapsto} f(x) \,\} \\ \quad \ldots \texttt{get}(t) \ldots & \quad \ldots \texttt{get}(t) \ldots \\ \quad \ldots & \quad \ldots \\ \{\, t \overset{R}{\mapsto} f(x) \ast Q_1 \,\} & \{\, t \overset{R}{\mapsto} f(x) \ast Q_2 \,\} \end{array}$$

$$\{\, t \overset{R}{\mapsto} f(x) \ast t \overset{R}{\mapsto} f(x) \ast Q_1 \ast Q_2 \,\}$$

Exemple de partage en lecture seule

$$\texttt{let}\ t = \texttt{alloc}(1)\ \texttt{in}$$
$$\texttt{set}(t, f(x));$$
$$\{\, t \overset{1}{\mapsto} f(x) \,\} \Rightarrow \{\, t \overset{R}{\mapsto} f(x) \,\} \Rightarrow \{\, t \overset{R}{\mapsto} f(x) \ast t \overset{R}{\mapsto} f(x) \,\}$$

$$\begin{array}{l|l}
\{\, t \overset{R}{\mapsto} f(x) \,\} & \{\, t \overset{R}{\mapsto} f(x) \,\} \\
\quad \ldots \texttt{get}(t) \ldots & \quad \ldots \texttt{get}(t) \ldots \\
\quad \ldots & \quad \ldots \\
\{\, t \overset{R}{\mapsto} f(x) \ast Q_1 \,\} & \{\, t \overset{R}{\mapsto} f(x) \ast Q_2 \,\}
\end{array}$$

$$\{\, t \overset{R}{\mapsto} f(x) \ast t \overset{R}{\mapsto} f(x) \ast Q_1 \ast Q_2 \,\}$$

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 :

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

Partage en lecture seule avec permissions fractionnaires

$$\texttt{let}\ t = \texttt{alloc}(1)\ \texttt{in}$$
$$\texttt{set}(t, f(x));$$
$$\{\, t \overset{1}{\mapsto} f(x) \,\} \Rightarrow \{\, t \overset{1/2}{\mapsto} f(x) \star t \overset{1/2}{\mapsto} f(x) \,\}$$

$$\begin{array}{l|l} \{\, t \overset{1/2}{\mapsto} f(x) \,\} & \{\, t \overset{1/2}{\mapsto} f(x) \,\} \\ \quad \ldots \texttt{get}(t) \ldots & \quad \ldots \texttt{get}(t) \ldots \\ \quad \ldots & \quad \ldots \\ \{\, t \overset{1/2}{\mapsto} f(x) \star Q_1 \,\} & \{\, t \overset{1/2}{\mapsto} f(x) \star Q_2 \,\} \end{array}$$

$$\{\, t \overset{1/2}{\mapsto} f(x) \star t \overset{1/2}{\mapsto} f(x) \star Q_1 \star Q_2 \,\} \Rightarrow \{\, t \overset{1}{\mapsto} f(x) \star Q_1 \star Q_2 \,\}$$
$$\texttt{free}(t)$$
$$\{\, Q_1 \star Q_2 \,\}$$

Algèbre de permissions

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».

États mémoires avec permissions

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

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

La combinaison est caractérisée par

$$Dom(h_1 \oplus h_2) = Dom(h_1) \cup Dom(h_2)$$

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

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 $\uplus$.

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

$$P \mathrel{-\!\!\star} Q = \lambda h.\ \exists h_1, h_2,\ h_2 = h \oplus h_1 \wedge P\ h_1 \wedge Q\ h_2$$

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

Passivité (Bornat, Calcagno, O'Hearn, Parkinson 2005)

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 $\{\, \ell \overset{1/2}{\mapsto} v \,\}\ c\ \{\, \lambda_.\ \ell \overset{1/2}{\mapsto} v' \,\}$ avec $v' \neq v$.

Par encadrement avec $\ell \overset{1/2}{\mapsto} v$, on obtient

$$\{\, \ell \overset{1}{\mapsto} v \,\}\ c\ \{\, \lambda_.\ \ell \overset{1/2}{\mapsto} v' \star \ell \overset{1/2}{\mapsto} v \,\}$$

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

Passivité et sections atomiques

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 $\ell \overset{1/2}{\mapsto} _$ manquante, on peut dériver

$$\ell \overset{1/2}{\mapsto} _ \vdash \{\, \ell \overset{1/2}{\mapsto} v \,\}\ \texttt{atomic}(\texttt{set}(\ell, v'))\ \{\, \lambda_.\ \ell \overset{1/2}{\mapsto} v' \,\}$$

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 :

$$\ell \overset{n}{\mapsto} v \;=\; \ell \overset{n+1}{\longmapsto} v \star \ell \overset{-1}{\longmapsto} v \qquad \text{si } n \geq 0$$

Un écrivain, plusieurs lecteurs (Courtois, Heymans, Parnas, 1972)

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

Invariant de *write* : $b \overset{0}{\mapsto} _$

Invariant de *read* : $\exists n,\ count \overset{0}{\mapsto} n \star (\langle n = 0\rangle \vee \langle n > 0\rangle \star b \overset{n}{\mapsto} _)$

Code fantôme

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

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

Variables auxiliaires : des variables mathématiques $\alpha, \beta, \ldots$ quantifiées universellement en tête du triplet.

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

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

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

Le code fantôme

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`

$$\{ r \geq 0 \land \exists q,\ a = b \cdot q + r \}$$

$r := r - b$

`done`

$$\{ r = a \bmod b \}$$

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

Exemple : le reste de la division euclidienne

$$\{\, a \geq 0 \,\}$$

$r := a;$

👻 $q := 0;$

`while` $r \geq b$ `do`

$$\{\, r \geq 0 \land a = b \cdot q + r \,\}$$

  👻 $q := q + 1;$

  $r := r - b$

`done`

$$\{\, r = a \bmod b \,\}$$

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 3e cours : marquer tous les nœuds atteignables depuis la racine *r*.

$\texttt{def}\ DFS\ r =$
$\quad \texttt{if MARK}[r] = 0\ \texttt{then begin}$
$\quad\quad \texttt{MARK}[r] := 1;$
$\quad\quad \texttt{for}\ i = 0\ \texttt{to ARITY}[r] - 1\ \texttt{do}\ DFS(\texttt{CHILD}[r][i])\ \texttt{done}$
$\quad \texttt{end}$

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 \ {p};
  if MARK[p] = 0 then begin
    MARK[p] := 1;
    👻 W := W ∪ {CHILD[p][i] | 0 ≤ i < 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

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

et conclure

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

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.

👻 $PR := \varepsilon; CO := \varepsilon;$

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

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

Le puzzle : $1 + 1 = 2$?

$$\texttt{set}(n, 0);$$
$$\texttt{atomic}(\mathit{incr}(n)) \parallel \texttt{atomic}(\mathit{incr}(n))$$

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

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 $J = \exists x,\ n \mapsto x \ast \langle x \geq 0 \rangle$.

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

Tracer les processus avec du code fantôme

$$\texttt{set}(n, 0);$$

$$👻\ \texttt{set}(a, 0); \texttt{set}(b, 0);$$

$$\texttt{atomic}(incr(n); 👻\ incr(a)) \parallel \texttt{atomic}(incr(n); 👻\ incr(b))$$

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

Tracer les processus avec du code fantôme

$$\texttt{set}(n, 0);$$

$$👻\ \texttt{set}(a, 0); \texttt{set}(b, 0);$$

$$\texttt{atomic}(\mathit{incr}(n); 👻\ \mathit{incr}(a)) \mathrel{\|} \texttt{atomic}(\mathit{incr}(n); 👻\ \mathit{incr}(b))$$

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

$$J \quad = \quad \exists x, y,\ a \overset{1/2}{\mapsto} x \ast b \overset{1/2}{\mapsto} y \ast n \overset{1}{\mapsto} x + y$$

On a :

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

$incr(n);$

$incr(a);$

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

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

Les permissions fractionnaires à la rescousse

On peut alors dériver :

$$\texttt{set}(n,0); \texttt{set}(a,0); \texttt{set}(b,0);$$

$$\{\, n \overset{1}{\mapsto} 0 \ast a \overset{1}{\mapsto} 0 \ast b \overset{1}{\mapsto} 0 \,\} \Rightarrow \{\, J \ast a \overset{1/2}{\mapsto} 0 \ast b \overset{1/2}{\mapsto} 0 \,\}$$

$$\left.\begin{array}{l} \{\, a \overset{1/2}{\mapsto} 0 \,\} \\ \quad \texttt{atomic}(incr(n); incr(a)) \\ \{\, a \overset{1/2}{\mapsto} 1 \,\} \end{array}\right\| \begin{array}{l} \{\, b \overset{1/2}{\mapsto} 0 \,\} \\ \quad \texttt{atomic}(incr(n); incr(b)) \\ \{\, b \overset{1/2}{\mapsto} 1 \,\} \end{array}$$

$$\{\, J \ast a \overset{1/2}{\mapsto} 1 \ast b \overset{1/2}{\mapsto} 1 \,\} \Rightarrow \{\, n \overset{1}{\mapsto} 2 \ast a \overset{1}{\mapsto} 1 \ast b \overset{1}{\mapsto} 1 \,\}$$

Et donc $n = 2$ à la fin !

Les permissions fractionnaires à la rescousse

On peut alors dériver :

$$\texttt{set}(n,0); \texttt{set}(a,0); \texttt{set}(b,0);$$

$$\{\, n \overset{1}{\mapsto} 0 \ast a \overset{1}{\mapsto} 0 \ast b \overset{1}{\mapsto} 0 \,\} \Rightarrow \{\, J \ast a \overset{1/2}{\mapsto} 0 \ast b \overset{1/2}{\mapsto} 0 \,\}$$

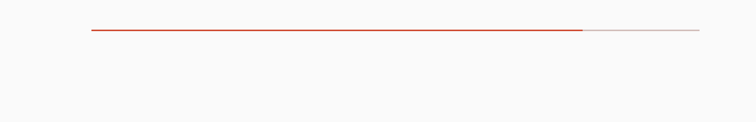
$$\begin{array}{l||l} \{\, a \overset{1/2}{\mapsto} 0 \,\} & \{\, b \overset{1/2}{\mapsto} 0 \,\} \\ \quad \texttt{atomic}(\mathit{incr}(n); \mathit{incr}(a)) & \quad \texttt{atomic}(\mathit{incr}(n); \mathit{incr}(b)) \\ \{\, a \overset{1/2}{\mapsto} 1 \,\} & \{\, b \overset{1/2}{\mapsto} 1 \,\} \end{array}$$

$$\{\, J \ast a \overset{1/2}{\mapsto} 1 \ast b \overset{1/2}{\mapsto} 1 \,\} \Rightarrow \{\, n \overset{1}{\mapsto} 2 \ast a \overset{1}{\mapsto} 1 \ast b \overset{1}{\mapsto} 1 \,\}$$

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

$\rightarrow$ séminaire de J. H. Jourdan le 2021-04-08.


Verrous stockés en mémoire

Parallélisme à grain fin

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.

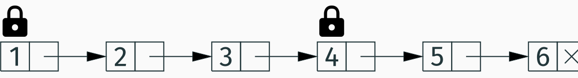
Exemple : un processus enlève ≪2≫, l’autre enlève ≪5≫.

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≫.

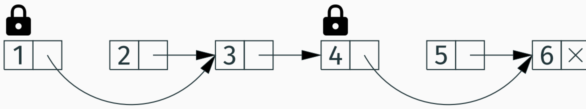


Exemple : liste simplement chaînée

```
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≫.

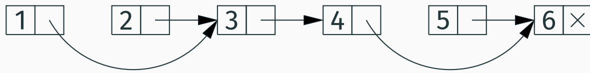


Exemple : liste simplement chaînée

```
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≫.

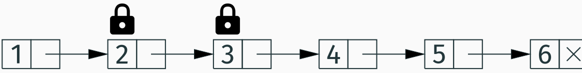
Verrouiller un seul nœud à la fois ne suffit pas !
Exemple : un processus enlève ≪3≫, l'autre enlève ≪4≫.

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 !

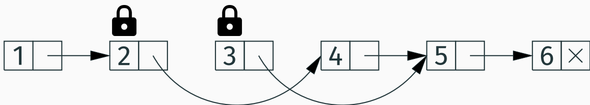
Exemple : un processus enlève ≪3≫, l'autre enlève ≪4≫.

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*)

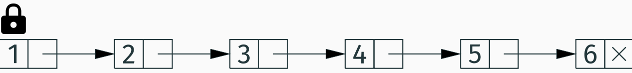
Pour modifier un nœud, il faut l'avoir verrouillé ainsi que son prédécesseur.

Exemple : destruction de ≪4≫.

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

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

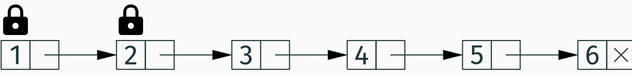
Exemple : destruction de ≪4≫.



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

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

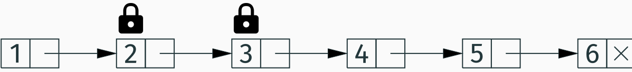
Exemple : destruction de ≪4≫.



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

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

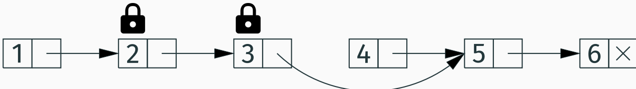
Exemple : destruction de ≪4≫.



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*)

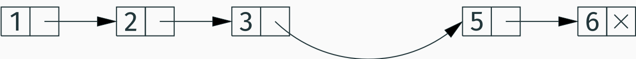
Pour modifier un nœud, il faut l'avoir verrouillé ainsi que son prédécesseur.

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 :

$\ell \overset{\pi}{\bullet\!\!\rightarrow} RI$ ≪à l'adresse ℓ, avec permission π, il y a un verrou qui protège la ressource décrite par l'invariant *RI*≫

🔒 ℓ ≪le verrou à l'adresse ℓ est verrouillé par le processus courant≫

Les ≪petites règles≫ pour les verrous :

$$\{\, \ell \overset{\pi}{\bullet\!\!\rightarrow} RI \,\} \quad \texttt{lock}(\ell) \quad \{\, \ell \overset{\pi}{\bullet\!\!\rightarrow} RI \star 🔒\, \ell \star RI \,\}$$

$$\{\, \ell \overset{\pi}{\bullet\!\!\rightarrow} RI \star 🔒\, \ell \star RI \,\} \quad \texttt{unlock}(\ell) \quad \{\, \ell \overset{\pi}{\bullet\!\!\rightarrow} RI \,\}$$

$$\{\, \ell \overset{1}{\mapsto} _ \star RI \,\} \quad \texttt{initlock}(\ell) \quad \{\, \ell \overset{1}{\bullet\!\!\rightarrow} RI \,\}$$

$$\{\, \ell \overset{1}{\bullet\!\!\rightarrow} RI \,\} \quad \texttt{destroylock}(\ell) \quad \{\, \ell \overset{1}{\mapsto} _ \star RI \,\}$$

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.

$$\begin{aligned}
\mathit{list}(p, n) = (&\langle n = +\infty\rangle \star p.\mathtt{val} \overset{1}{\mapsto} n \star p.\mathtt{next} \overset{1}{\mapsto} \mathtt{NULL}) \\
&\vee (\exists q, n',\ \ \langle n < n'\rangle \star p.\mathtt{val} \overset{1}{\mapsto} n \star p.\mathtt{next} \overset{1}{\mapsto} q \\
&\qquad \star\, p.\mathtt{lock} \overset{1}{\bullet\!\!\to} \mathit{list}(q, n'))
\end{aligned}$$

$$\begin{aligned}
\mathit{listhead}(p, \pi) = \exists q, n,\ \ &p.\mathtt{val} \overset{\pi}{\mapsto} -\infty \star p.\mathtt{next} \overset{\pi}{\mapsto} q \\
&\star\, p.\mathtt{lock} \overset{\pi}{\bullet\!\!\to} \mathit{list}(q, n)
\end{aligned}$$

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

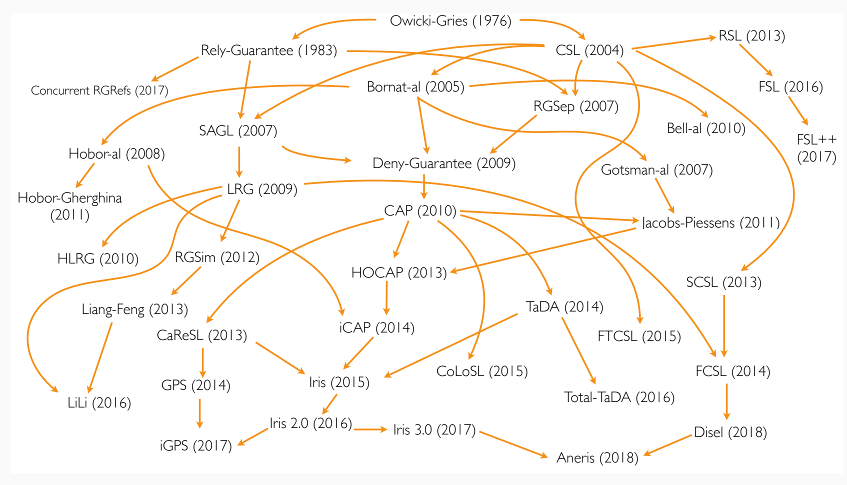
Point d'étape

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 $\mathtt{x} = \alpha$ (en logique de Hoare)
- Des faits à propos du tas mémoire $\mathtt{emp}$, $\ell \mapsto v$, $\ell \mapsto _$
- Même chose plus des permissions $\ell \overset{\pi}{\mapsto} v$
- Des faits à propos des verrous $\ell \overset{\pi}{\bullet\!\!\rightarrow} RI$, 🔒 ℓ
- Des faits à propos de l'état fantôme.
- Le temps disponible (→ séminaire de F. Pottier)
- Des systèmes de transition (→ 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

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.