Logiques de programmes, quatrième cours

Parallélisme à mémoire partagée : la logique de séparation concurrente

Xavier Leroy

2021-03-25

Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

Introduction :
Le calcul parallèle
à mémoire partagée

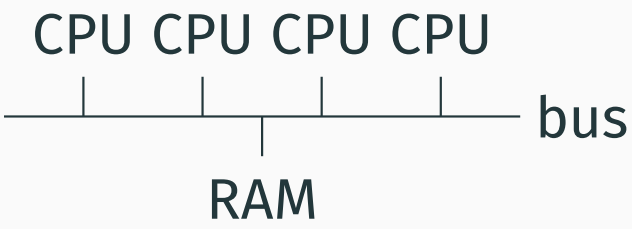
Bonus Bureau, Computing Divison, 11/24/1924

Le calcul parallèle

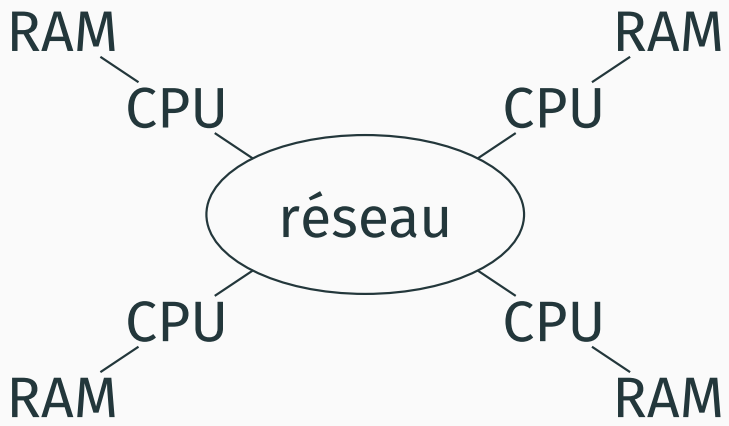
Faire travailler ensemble plusieurs unités de calcul (CPU) pour effectuer une tâche plus rapidement.

Deux principales modalités du parallélisme :

à mémoire partagée



à mémoire distribuée



De nombreuses réalisations combinant les deux modèles : multicœurs, multiprocesseurs, GPU, *clusters*, *grids*, *cloud*, …

Quelques dates dans l'histoire du calcul parallèle

1962 Premier multiprocesseur symétrique : le Burroughs D825 (1 à 4 CPUs partageant 1 à 16 modules mémoire).

1965 Début du projet Multics, le premier système d'exploitation moderne avec support pour multiprocesseurs.

1973 Xerox Parc : stations de travail Alto + réseau Ethernet. Premier gros calcul distribué (rendu d'images).

1999 Lancement de SETI@home et de Folding@home, deux énormes calculs distribués sur Internet .

2006 Premiers processeurs multicœurs «grand public» (Intel Core Duo et AMD Athlon 64 X2).

2012 (environ) Tous les processeurs pour PC, tablettes et smartphones sont multicœurs.

Le parallélisme à mémoire partagée

Intérets :

- Tous les processeurs ont accès direct à toutes les données.
- Pas de duplication des données.
- Communications très rapides (via des zones de mémoire partagée).

Difficultés :

- Risque d'interférence entre les actions des processeurs.
- En particulier : les courses critiques (*race conditions*).

Course critique (*race condition*)

Plusieurs accès simultanés à la même case mémoire, dont au moins un accès en écriture.

Cas 1 : deux écritures simultanées.

$$\texttt{set}(\ell, 1) \parallel \texttt{set}(\ell, 2)$$

Le programme ne contrôle pas quelle valeur finit dans la case ℓ.

Cas 2 : écriture et lecture simultanées

$$\texttt{set}(\ell, 1) \parallel \texttt{let } x = \texttt{get}(\ell)$$

Le programme ne contrôle pas quelle valeur est lue dans x.

Un exemple de course critique

$$x := x + 1 \;\Big\|\; x := x + 1$$

Compilé en 3 instructions (lecture, calcul, écriture) :

$$\begin{array}{l} \texttt{let } t = \texttt{get}(\&x) \texttt{ in} \\ \texttt{let } t = t + 1 \texttt{ in} \\ \texttt{set}(\&x, t) \end{array} \;\Big\|\; \begin{array}{l} \texttt{let } t = \texttt{get}(\&x) \texttt{ in} \\ \texttt{let } t = t + 1 \texttt{ in} \\ \texttt{set}(\&x, t) \end{array}$$

Un exemple de course critique

$$x := x + 1 \;\Big\|\; x := x + 1$$

Une exécution possible :

$$\begin{array}{l|l}
\texttt{let } t = \texttt{get}(\&x) \texttt{ in} & \\
\texttt{let } t = t + 1 \texttt{ in} & \\
\texttt{set}(\&x, t) & \\
 & \texttt{let } t = \texttt{get}(\&x) \texttt{ in} \\
 & \texttt{let } t = t + 1 \texttt{ in} \\
 & \texttt{set}(\&x, t)
\end{array}$$

Avec $x = 0$ initialement, on termine sur $x = 2$.

Un exemple de course critique

$$x := x + 1 \;\Big\|\; x := x + 1$$

Une autre exécution possible :

$$\begin{array}{l|l}
\texttt{let } t = \texttt{get}(\&x) \texttt{ in} & \\
\texttt{let } t = t + 1 \texttt{ in} & \\
 & \texttt{let } t = \texttt{get}(\&x) \texttt{ in} \\
\texttt{set}(\&x, t) & \\
 & \texttt{let } t = t + 1 \texttt{ in} \\
 & \texttt{set}(\&x, t)
\end{array}$$

Avec $x = 0$ initialement, on termine sur $x = 1$.

Un exemple plus réaliste

La partie ≪producteur≫ d'un schéma producteur/consommateur : chaque processus produit des données x et les stocke dans un tampon partagé T (un tableau de taille N indexé par i).

$$\texttt{while}\ i \geq N\ \texttt{do}\ \texttt{pause}();$$
$$T[i] := x;$$
$$i := i + 1;$$

Un exemple plus réaliste

Avec deux producteurs en parallèle :

`while` $i \geq N$ `do pause();`	
	`while` $i \geq N$ `do pause();`
	$T[i] := x_1;$
	$i := i + 1;$
$T[i] := x_2;$ ✘	
$i := i + 1;$	

Risque d'accès hors borne dans le tableau
(si $i = N - 1$ initialement).

Un exemple plus réaliste

Avec deux producteurs en parallèle :

`while` $i \geq N$ `do pause();`	
	`while` $i \geq N$ `do pause();`
$T[i] := x_1;$	
	$T[i] := x_2;$
	$i := i + 1;$
$i := i + 1;$	

Une des deux données x_1, x_2 est perdue.

L'entrée $T[i-1]$ du tableau n'est pas initialisée.

Synchronisation par sections critiques

En Java :

```
synchronized (obj) {
  ...
}
```

En C :

```
pthread_mutex_lock(mut);
...
pthread_mutex_unlock(mut);
```

Garantissent l'exclusion mutuelle : à tout moment au plus un processus exécute la section critique.

Exemple : un producteur bien synchronisé.

```
synchronized (buff) {
    while (buff.i >= N) buff.wait();
    buff.T [ buff.i ] = x;
    buff.i ++ ;
}
```

Synchronisation et logiques de programmes

De nombreux mécanismes de synchronisation :

- exclusion mutuelle : sémaphores, verrous, …
- barrières et vagues de calcul ;
- passage de messages ;
- instructions atomiques des processeurs ;

($\rightarrow$ algorithmes *lock-free*)

Quelles logiques de programmes pour raisonner sur l'interférence et garantir la bonne synchronisation, notamment l'absence de courses critiques ?

Parallélisme sans partage de ressources

Exécuter deux commandes en parallèle

Commandes :

$$c := \ldots \mid c_1 \parallel c_2 \quad \text{exécute } c_1 \text{ et } c_2 \text{ en parallèle}$$

Sémantique : un entrelacement des réductions de c_1 et c_2.

$$(a_1 \parallel a_2)/h \to 0/h \qquad \text{(ou toute combinaison de } a_1 \text{ et } a_2\text{)}$$

$$(c_1 \parallel c_2)/h \to (c_1' \parallel c_2)/h' \qquad \text{si } c_1/h \to c_1'/h'$$

$$(c_1 \parallel c_2)/h \to (c_1 \parallel c_2')/h' \qquad \text{si } c_2/h \to c_2'/h'$$

$$(c_1 \parallel c_2)/h \to \texttt{err} \qquad \text{si } c_1/h \to \texttt{err} \text{ ou } c_2/h \to \texttt{err}$$

La règle de séparation pour l'exécution parallèle

$$\frac{\{\, P_1 \,\}\ c_1\ \{\, \lambda\_.\ Q_1 \,\} \qquad \{\, P_2 \,\}\ c_2\ \{\, \lambda\_.\ Q_2 \,\}}{\{\, P_1 \mathbin{*} P_2 \,\}\ c_1 \parallel c_2\ \{\, \lambda\_.\ Q_1 \mathbin{*} Q_2 \,\}}$$

Intuition :

- l'état mémoire initial h se décompose en $h_1 \uplus h_2$ avec h_1 satisfaisant P_1 et h_2 satisfaisant P_2 ;
- c_1 s'exécute dans h_1 sans modifier h_2 ;
- c_2 s'exécute dans h_2 sans modifier h_1 ;
- les états finaux h_1', h_2' satisfont Q_1, Q_2 et sont disjoints.

La règle de séparation pour l'exécution parallèle

$$\frac{\{\, P_1 \,\}\ c_1\ \{\, \lambda\_.\ Q_1 \,\} \qquad \{\, P_2 \,\}\ c_2\ \{\, \lambda\_.\ Q_2 \,\}}{\{\, P_1 \mathbin{\ast} P_2 \,\}\ c_1 \parallel c_2\ \{\, \lambda\_.\ Q_1 \mathbin{\ast} Q_2 \,\}}$$

Autre intuition : la précondition $P_1 \mathbin{\ast} P_2$ garantit que les commandes c_1 et c_2 s'exécutent sans interférences.

L'exécution est donc sémantiquement équivalente à une exécution séquentielle $c_1; c_2$ ou $c_2; c_1$.

$$\frac{\dfrac{\{\, P_1 \,\}\ c_1\ \{\, \lambda\_.\ Q_1 \,\}}{\{\, P_1 \mathbin{\ast} P_2 \,\}\ c_1\ \{\, \lambda\_.\ Q_1 \mathbin{\ast} P_2 \,\}} \qquad \dfrac{\{\, P_2 \,\}\ c_2\ \{\, \lambda\_.\ Q_2 \,\}}{\{\, Q_1 \mathbin{\ast} P_2 \,\}\ c_2\ \{\, \lambda\_.\ Q_1 \mathbin{\ast} Q_2 \,\}}}{\{\, P_1 \mathbin{\ast} P_2 \,\}\ c_1; c_2\ \{\, \lambda\_.\ Q_1 \mathbin{\ast} Q_2 \,\}}$$

Parallélisme séparé entre sous-tableaux

Exemple : le tri Quicksort.

$$
\begin{array}{l}
\mathit{quicksort}\ T\ l\ h = \\
\quad \texttt{if}\ h - l \leq 20\ \texttt{then} \\
\quad\quad \mathit{insertionsort}\ T\ l\ h \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ m = \mathit{partition}\ T\ l\ h\ \texttt{in} \\
\quad\quad \mathit{quicksort}\ T\ l\ m \parallel \mathit{quicksort}\ T\ (m+1)\ h
\end{array}
$$

quicksort $T\ l\ h$ modifie le sous-tableau $T[l \ldots h]$ de T.

Les deux appels récursifs se font sur des sous-tableaux disjoints : $T[l \ldots m]$ et $T[m+1 \ldots h]$.

On peut donc les faire soit en séquence soit en parallèle.

Parallélisme séparé entre sous-arbres

$$\mathit{tree}(\texttt{Leaf}, p) = \langle p = \texttt{NULL} \rangle$$

$$\mathit{tree}(\texttt{Node}(t_1, x, t_2), p) = \exists p_1, p_2,\ p \mapsto p_1 \mathbin{\ast} p + 1 \mapsto x \mathbin{\ast} p + 2 \mapsto p_2 \mathbin{\ast} \mathit{tree}(t_1, p_1) \mathbin{\ast} \mathit{tree}(t_2, p_2)$$

Le prédicat de représentation garantit que les deux sous-arbres sont disjoints, et peuvent donc être parcourus et modifiés en parallèle.

$\mathit{incrtree}\ t\ \delta =$
  $\texttt{if}\ t \neq \texttt{NULL}\ \texttt{then}$
    $\texttt{let}\ l = \texttt{get}(t)\ \texttt{and}\ n = \texttt{get}(t+1)\ \texttt{and}\ r = \texttt{get}(t+2)\ \texttt{in}$
    $\texttt{set}(t+1, n+\delta);$
    $\mathit{incrtree}\ l\ \delta \parallel \mathit{incrtree}\ r\ \delta$

Absence de courses critiques

On ajoute une règle à la sémantique par réductions qui signale une erreur pour toute situation de course :

$$(c_1 \parallel c_2)/h \to \texttt{err} \quad \text{si} \quad Acc(c_1) \cap Acc(c_2) \neq \emptyset$$

$Acc(c)$ est l'ensemble des adresses mémoires que la commande c peut lire ou écrire à la prochaine étape de réduction :

$$Acc(\texttt{get}(a)) = Acc(\texttt{set}(a, a')) = Acc(\texttt{free}(a)) = \{a\}$$
$$Acc(\texttt{let}\ x = c_1\ \texttt{in}\ c_2) = Acc(c_1)$$
$$Acc(c_1 \parallel c_2) = Acc(c_1) \cup Acc(c_2)$$

Absence de courses critiques

On montre facilement que

$$c/h \not\to \texttt{err} \quad \Rightarrow \quad Acc(c) \subseteq Dom(h)$$

Par conséquent, si $c_1/h_1 \not\to \texttt{err}$ et $c_2/h_2 \not\to \texttt{err}$ et $h_1 \perp h_2$,

$$Acc(c_1) \cap Acc(c_2) \subseteq Dom(h_1) \cap Dom(h_2) = \emptyset$$

et donc $(c_1 \parallel c_2)/(h_1 \uplus h_2)$ ne peut pas se réduire en $\texttt{err}$ à cause d'une course critique.

Le résultat de correction sémantique (à la fin de ce cours) formalise cette analyse et montre que si $\{ P \}\ c\ \{ Q \}$, la commande c s'exécute sans courses critiques.

Parallélisme et partage de ressources

L’émergence de la logique de séparation concurrente

O’Hearn, Reynolds, Yang (2001), *Local Reasoning about Programs that Alter Data Structures*. La présentation moderne de la logique de séparation séquentielle.

O’Hearn (2001–2002), *Notes on separation logic for shared-variable concurrency*, non publié.

Reynolds (2002), *Separation Logic : A Logic for Shared Mutable Data Structures*. Donne la règle de séparation pour le parallélisme et mentionne les travaux en cours de O’Hearn.

O’Hearn (2004), *Resources, Concurrency and Local Reasoning*. Les idées clés + les principaux exemples.

Brookes (2004), *A Semantics for Concurrent Separation Logic*. Une sémantique et une preuve de correction pour la logique de O’Hearn.

Les ressources partagées

Une ressource se compose de

- une ou plusieurs cases mémoire :
  variables globales, objets alloués dynamiquement ;
- un verrou (*lock*) ou autre dispositif d'exclusion mutuelle qui protège les accès aux cases mémoire.

Exemple (un compteur partagé)

```
class Counter { int val; }
```

Exemple (une liste doublement chaînée partagée)

```
class DList { DListCell first, last; }
class DListCell { Object data; DListCell prev, next; }
```

Ressources partagées en logique de séparation

Idée géniale de O'Hearn : une ressource partagée peut être décrite par une assertion A de logique de séparation.

- L'empreinte mémoire de A définit l'ensemble des cases mémoires appartenant à la ressource.
- L'assertion A spécifie aussi la structure de ces cases (≪liste doublement chaînée≫) et d'autres invariants.

Exemple (un compteur partagé p)

$$p \mapsto n \mathbin{\ast} \langle n \geq 0 \rangle$$

Exemple (une liste doublement chaînée p, q)

$$\exists x, y, w,\ p \mapsto x \mathbin{\ast} q \mapsto y \mathbin{\ast} \mathit{dlist}(w, x, y)$$

Sections critiques en logique de séparation

L'accès à une ressource partagée r se fait uniquement dans une section critique

$$\texttt{with}\ r\ \texttt{do}\ c$$

en exclusion mutuelle avec les autres processus.

Notant RI_r l'assertion (l'invariant) associée à la ressource r :

$$\frac{\{\, RI_r \mathbin{\ast} P \,\}\ c\ \{\, RI_r \mathbin{\ast} Q \,\}}{\{\, P \,\}\ \texttt{with}\ r\ \texttt{do}\ c\ \{\, Q \,\}}$$

À l'entrée de la section critique, le processus acquiert la permission d'utiliser les cases mémoire de la ressource, décrites par RI_r.

Avant de sortir de la section critique, le processus doit rétablir l'invariant RI_r car d'autres processus vont entrer en section critique.

Sections critiques conditionnelles en logique de séparation

L'article original de O'Hearn considère des sections critiques conditionnelles

$$\texttt{with}\ r\ \texttt{when}\ b\ \texttt{do}\ c$$

où c est exécuté seulement lorsque la condition b est vraie.

La règle pour les s.c.c. est :

$$\frac{\{\, \langle b \rangle \ast RI_r \ast P \,\}\ c\ \{\, RI_r \ast Q \,\}}{\{\, P \,\}\ \texttt{with}\ r\ \texttt{when}\ b\ \texttt{do}\ c\ \{\, Q \,\}}$$

Exemple : décrémenter un compteur partagé

L'invariant est $RI_r = \exists n,\ p \mapsto n \ast \langle n \geq 0 \rangle$.

$\{\ \mathtt{emp}\ \}$

`with` r `do`

$\{\ \exists n,\ p \mapsto n \ast \langle n \geq 0 \rangle\ \}$

  `let` $n = \mathtt{get}(p)$ `in`

$\{\ p \mapsto n \ast \langle n \geq 0 \rangle\ \}$

  `if` $n > 0$ `then` $\mathtt{set}(p, n-1)$

$\{\ \exists n',\ p \mapsto n' \ast \langle n' \geq 0 \rangle\ \}$

`done`

$\{\ \mathtt{emp}\ \}$

Exemple : insertion dans une liste partagée

L'invariant est $RI_r = \exists q, w,\ p \mapsto q \ast list(w, q)$.

$\{\ \mathtt{emp}\ \}$

`with` r `do`

$\{\ \exists q, w,\ p \mapsto q \ast list(w, q)\ \}$

  `let` $q =$ `get`(p) `in`

$\{\ p \mapsto q \ast \exists w,\ list(w, q)\ \}$

  `let` $a =$ `cons`(x, q) `in`

$\{\ a \mapsto x \ast a + 1 \mapsto q \ast p \mapsto q \ast \exists w,\ list(w, q)\ \}$

  `set`(p, a)

$\{\ p \mapsto a \ast a \mapsto x \ast a + 1 \mapsto q \ast \exists w,\ list(w, q)\ \}$

$\Rightarrow \{\ \exists q, w,\ p \mapsto q \ast list(w, q)\ \}$

`done`

$\{\ \mathtt{emp}\ \}$

Formalisation simplifiée : les sections atomiques (Vafeiadis (2011))

Commandes :

$$
\begin{aligned}
c ::= {} & \ldots \\
& \mid c_1 \parallel c_2 && \text{exécute } c_1 \text{ et } c_2 \text{ en parallèle} \\
& \mid \texttt{atomic}\ c && \text{exécute } c \text{ en une étape insécable}
\end{aligned}
$$

Une section «super-critique» : pendant l'exécution de `atomic` c, tous les autres processus sont bloqués et n'exécutent rien.

Pertinence pratique :

- Si on fait du temps partagé sur un unique processeur : section atomique $\approx$ bloquer temporairement la préemption.
- Modélise des instructions atomiques du processeur.

Modélisation des instructions atomiques des processeurs

Échange atomique (*swap*) et ses cas particuliers :

$$swap(p, n) \stackrel{def}{=} \texttt{atomic}(\texttt{let}\ x = \texttt{get}(p)\ \texttt{in}\ \texttt{set}(p, n); x)$$

$$test\_and\_set(p) \stackrel{def}{=} swap(p, 1)$$

$$read\_and\_clear(p) \stackrel{def}{=} swap(p, 0)$$

Incrément / décrément atomique :

$$fetch\_and\_add(p, d) \stackrel{def}{=} \texttt{atomic}(\texttt{let}\ x = \texttt{get}(p)\ \texttt{in}\ \texttt{set}(p, x + d); x)$$

Comparaison et échange (*compare and swap*) :

$$CAS(p, x, n) \stackrel{def}{=} \texttt{atomic}(\texttt{let}\ c = \texttt{get}(p)\ \texttt{in}\ \texttt{if}\ c = x\ \texttt{then}\ (\texttt{set}(p, n); 1)\ \texttt{else}\ 0)$$

Sémantique opérationnelle des sections atomiques

$$(\texttt{atomic}\ c)/h \to a/h' \qquad \text{si } c/h \xrightarrow{*} a/h'$$

$$(\texttt{atomic}\ c)/h \to \texttt{err} \qquad \text{si } c/h \xrightarrow{*} \texttt{err}$$

Note : $\texttt{atomic}\ c_1 \parallel \texttt{atomic}\ c_2$ est donc équivalent à $c_1; c_2$ ou $c_2; c_1$. Il n'y a pas d'entrelacement entre les étapes de réduction de c_1 et celles de c_2.

Note : si c/h diverge, $(\texttt{atomic}\ c)/h$ est bloquée. En pratique, c ne contient pas de boucles et termine toujours.

Un ≪triplet≫ pour le parallélisme à sections critiques

$$J \vdash \{ P \}\ c\ \{ Q \}$$

L'assertion J est un invariant sur la mémoire partagée (accessible uniquement dans des sections atomiques `atomic` c).

La précondition P et la postcondition Q décrivent la mémoire propre à la commande c.

Les règles pour les sections atomiques

Exécution d'une section atomique :

$$\frac{\mathtt{emp} \vdash \{\, P \mathbin{\ast} J \,\}\ c\ \{\, \lambda v.\ Q\ v \mathbin{\ast} J \,\}}{J \vdash \{\, P \,\}\ \mathtt{atomic}\ c\ \{\, Q \,\}}$$

Partage d'une ressource J' :

$$\frac{J \mathbin{\ast} J' \vdash \{\, P \,\}\ c\ \{\, Q \,\}}{J \vdash \{\, P \mathbin{\ast} J' \,\}\ c\ \{\, \lambda v.\ Q\ v \mathbin{\ast} J' \,\}}$$

Encadrement sur l'invariant :

$$\frac{J \vdash \{\, P \,\}\ c\ \{\, Q \,\}}{J \mathbin{\ast} J' \vdash \{\, P \,\}\ c\ \{\, Q \,\}}$$

Les règles pour les structures de contrôle (rappel)

$$\frac{P \Rightarrow Q \; [\![a]\!]}{J \vdash \{\, P \,\}\; a \;\{\, Q \,\}}$$

$$\frac{J \vdash \{\, P \,\}\; c \;\{\, R \,\} \qquad \forall v,\; J \vdash \{\, R\, v \,\}\; c'[x \leftarrow v] \;\{\, Q \,\}}{J \vdash \{\, P \,\}\; \texttt{let}\; x = c\; \texttt{in}\; c' \;\{\, Q \,\}}$$

$$\frac{J \vdash \{\, \langle b \rangle \ast P \,\}\; c_1 \;\{\, Q \,\} \qquad J \vdash \{\, \langle \neg b \rangle \ast P \,\}\; c_2 \;\{\, Q \,\}}{\{\, P \,\}\; \texttt{if}\; b\; \texttt{then}\; c_1\; \texttt{else}\; c_2 \;\{\, Q \,\}}$$

$$\frac{J \vdash \{\, P_1 \,\}\; c_1 \;\{\, \lambda\_.\, Q_1 \,\} \qquad J \vdash \{\, P_2 \,\}\; c_2 \;\{\, \lambda\_.\, Q_2 \,\}}{J \vdash \{\, P_1 \ast P_2 \,\}\; c_1 \parallel c_2 \;\{\, \lambda\_.\, Q_1 \ast Q_2 \,\}}$$

Les ≪petites≫ règles pour les accès mémoire (rappel)

$$J \vdash \quad \{\,\mathtt{emp}\,\}\ \mathtt{alloc}(N)\ \{\,\lambda \ell.\, \ell \mapsto \_ \mathbin{\ast} \cdots \mathbin{\ast} \ell + N - 1 \mapsto \_\,\}$$

$$J \vdash \{\,[\![a]\!] \mapsto x\,\}\ \ \mathtt{get}(a)\ \ \{\,\lambda v.\, \langle v = x \rangle \mathbin{\ast} [\![a]\!] \mapsto x\,\}$$

$$J \vdash \{\,[\![a]\!] \mapsto \_\,\}\ \mathtt{set}(a, a')\ \{\,\lambda v.\, [\![a]\!] \mapsto [\![a']\!]\,\}$$

$$J \vdash \{\,[\![a]\!] \mapsto \_\,\}\ \ \mathtt{free}(a)\ \ \{\,\lambda v.\, \mathtt{emp}\,\}$$

Les règles structurelles (attention ! piège !)

$$\frac{J \vdash \{\, P \,\}\ c\ \{\, Q \,\}}{J \vdash \{\, P \ast R \,\}\ c\ \{\, \lambda v.\ Q\ v \ast R \,\}}\ \text{(encadrement)}$$

$$\frac{P \Rrightarrow P' \qquad J \vdash \{\, P' \,\}\ c\ \{\, Q' \,\} \qquad \forall v,\ Q'\ v \Rrightarrow Q\ v}{J \vdash \{\, P \,\}\ c\ \{\, Q \,\}}\ \text{(conséquence)}$$

$$\frac{J \vdash \{\, P \,\}\ c\ \{\, Q \,\} \qquad J \vdash \{\, P' \,\}\ c\ \{\, Q' \,\}}{J \vdash \{\, P \vee P' \,\}\ c\ \{\, \lambda v.\ Q\ v \vee Q'\ v \,\}}\ \text{(disjonction)}$$

$$\frac{J \text{ précise} \qquad J \vdash \{\, P \,\}\ c\ \{\, Q \,\} \qquad J \vdash \{\, P' \,\}\ c\ \{\, Q' \,\}}{J \vdash \{\, P \wedge P' \,\}\ c\ \{\, \lambda v.\ Q\ v \wedge Q'\ v \,\}}\ \text{(conjonction)}$$

La règle de conjonction et le contre-exemple de Reynolds

Prenons $J = \mathtt{true}$ (l'assertion $\lambda h.\top$ vraie de toutes les mémoires).
Soit $\mathtt{one} = 1 \mapsto \_$. On a $\mathtt{one} \mathbin{✶} \mathtt{true} \Rightarrow \mathtt{true}$, donc

$$\mathtt{emp} \vdash \{\, \mathtt{one} \mathbin{✶} \mathtt{true} \,\}\ 0\ \{\, \lambda\_.\mathtt{emp} \mathbin{✶} \mathtt{true} \,\}$$

$$\mathtt{emp} \vdash \{\, \mathtt{one} \mathbin{✶} \mathtt{true} \,\}\ 0\ \{\, \lambda\_.\mathtt{one} \mathbin{✶} \mathtt{true} \,\}$$

et en appliquant la règle $\mathtt{atomic}$,

$$J \vdash \{\, \mathtt{one} \,\}\ \mathtt{atomic}\ 0\ \{\, \lambda\_.\mathtt{emp} \,\}$$

$$J \vdash \{\, \mathtt{one} \,\}\ \mathtt{atomic}\ 0\ \{\, \lambda\_.\mathtt{one} \,\}$$

Si la règle de conjonction était vraie pour tout J, on concluerait

$$J \vdash \{\, \mathtt{one} \wedge \mathtt{one} \,\}\ \mathtt{atomic}\ 0\ \{\, \lambda\_.\mathtt{emp} \wedge \mathtt{one} \,\}$$

or la postcondition $\mathtt{emp} \wedge \mathtt{one}$ est toujours fausse.

Les assertions précises

Intuitivement : une assertion P est précise si son empreinte mémoire est définie de manière unique.

Formellement : si P découpe un sous-tas h_1 dans un tas h donné, ce sous-tas h_1 est déterminé de manière unique :

$$h = h_1 \uplus h_2 = h_1' \uplus h_2' \quad \wedge \quad P\, h_1 \quad \wedge \quad P\, h_1' \quad \Rightarrow \quad h_1 = h_1'$$

Exemples d'assertions précises ou non

Assertions précises	Assertions non précises
$\texttt{emp}$	$\texttt{true}$
$\ell \mapsto \_$	$\exists \ell,\ \ell \mapsto \_$
$\ell \mapsto v$	$\exists \ell,\ \ell \mapsto v$
$\exists v, \ell \mapsto v \mathbin{\ast} R(v)$	
$P \mathbin{\ast} Q$	$P \mathbin{\ast} \texttt{true}$
$\langle b \rangle \mathbin{\ast} P \vee \langle \neg b \rangle \mathbin{\ast} Q$	$\texttt{emp} \vee \ell \mapsto \_$

(si P, Q, $R(v)$ sont précises)

Sémaphores binaires et applications

Codage des sémaphores binaires

Un sémaphore binaire = une case mémoire p qui contient 0 (signifiant «non disponible») ou 1 (signifiant «disponible»).

Les opérations P (prendre) et V (relâcher) :

$$V(sem) = \texttt{atomic}(\texttt{set}(sem, 1))$$
$$P(sem) = \texttt{let } x = swap(sem, 0) \texttt{ in}$$
$$\texttt{if } x = 1 \texttt{ then } 0 \texttt{ else } P(sem)$$

avec

$$swap(p, n) = \texttt{atomic}(\texttt{let } x = \texttt{get}(p) \texttt{ in } \texttt{set}(p, n); x)$$

Note : $P(sem)$ fait une attente active, et peut ne pas terminer, mais la boucle est en dehors de la section atomique.

Les règles pour les sémaphores binaires

Soit RI l'assertion décrivant les ressources associées au sémaphore. On suppose que RI est précise.
On prend comme invariant sur l'état partagé

$$J(sem, RI) \stackrel{def}{=} \exists n.\, sem \mapsto n \ast (\langle n = 0 \rangle \vee \langle n = 1 \rangle \ast RI)$$

c'est-à-dire «si le sémaphore est occupé, les ressources RI sont dans la mémoire partagée». On peut alors dériver :

$$J(sem, RI) \vdash \{\, RI \,\}\ V(sem)\ \{\, \mathtt{emp} \,\}$$
$$J(sem, RI) \vdash \{\, \mathtt{emp} \,\}\ P(sem)\ \{\, RI \,\}$$

Autrement dit : donner p c'est placer RI dans la mémoire partagée, et prendre p c'est récupérer RI depuis la mémoire partagée.

Synchronisation avec un sémaphore

On prend l'assertion $RI = \exists n, x \mapsto n \mathbin{\ast} \langle n \text{ premier} \rangle$,
«la variable x contient un nombre premier».

$$\{\, sem \mapsto 0 \mathbin{\ast} x \mapsto \_ \,\}$$

$$\begin{array}{l|l}
\{\, x \mapsto _ \,\} & \{\, \texttt{emp} \,\} \\
\quad \texttt{set}(x, 53); & P(sem); \\
\{\, x \mapsto 53 \,\} \Rightarrow \{\, RI \,\} & \{\, RI \,\} \\
\quad V(sem) & \texttt{let } n = \texttt{get}(x) \texttt{ in} \\
\{\, \texttt{emp} \,\} & \{\, x \mapsto n \mathbin{\ast} \langle n \text{ premier} \rangle \,\} \\
 & \texttt{print}(n)
\end{array}$$

Le P et le V assurent que le processus droit ne lit pas x avant que le processus gauche ne l'ait initialisé, et transfèrent la permission d'accéder à x du processus gauche au processus droit.

Synchronisation et transfer de ressources avec un sémaphore

On prend l'assertion $RI = \exists p, x \mapsto p \mathbin{\ast} p \mapsto \_$
«la variable x pointe vers une adresse valide».

$$\{\, sem \mapsto 0 \mathbin{\ast} x \mapsto \_ \,\}$$

$$\begin{array}{l|l}
\{\, x \mapsto _ \,\} & \{\, \mathtt{emp} \,\} \\
\quad \mathtt{let}\ p = \mathtt{alloc}(1)\ \mathtt{in} & P(sem); \\
\{\, x \mapsto _ \mathbin{\ast} p \mapsto _ \,\} & \{\, RI \,\} \\
\quad \mathtt{set}(x, p); & \mathtt{let}\ p = \mathtt{get}(x)\ \mathtt{in} \\
\{\, x \mapsto p \mathbin{\ast} p \mapsto _ \,\} \Rightarrow \{\, RI \,\} & \{\, x \mapsto p \mathbin{\ast} p \mapsto _ \,\} \\
\quad V(sem) & \mathtt{free}(p) \\
\{\, \mathtt{emp} \,\} & \{\, x \mapsto _ \,\}
\end{array}$$

La mémoire allouée par le processus gauche est transférée et désallouée sans risques par le processus droit.

Dérivation de la règle pour P

On rappelle l'invariant sur l'état partagé :

$$J(sem, RI) \stackrel{def}{=} \exists n.\, sem \mapsto n \ast (\langle n = 0 \rangle \vee \langle n = 1 \rangle \ast RI)$$

Pour $swap(sem, 0)$, on a le triplet

$$J(sem, RI) \vdash \{\, \mathtt{emp} \,\}\ swap(sem, 0)\ \{\, \lambda n.\, \langle n = 0 \rangle \vee \langle n = 1 \rangle \ast RI \,\}$$

$P(sem)$ itère $swap(sem, 0)$ jusqu'à ce que le résultat soit 1, d'où

$$J(sem, RI) \vdash \{\, \mathtt{emp} \,\}\ P(sem)\ \{\, RI \,\}$$

Dérivation de la règle pour V

$$J(sem, RI) \stackrel{def}{=} \exists n.\, sem \mapsto n \ast (\langle n = 0 \rangle \vee \langle n = 1 \rangle \ast RI)$$

Il suffit de montrer

$$\mathtt{emp} \vdash \{\, RI \ast J(sem, RI) \,\}\ \mathtt{set}(sem, 1)\ \{\, sem \mapsto 1 \ast RI \,\}$$

pour avoir $\mathtt{emp} \vdash \{\, RI \ast J(sem, RI) \,\}\ \mathtt{set}(sem, 1)\ \{\, J(sem, RI) \,\}$
et donc $J(sem, RI) \vdash \{\, RI \,\}\ V(sem)\ \{\, \mathtt{emp} \,\}$.

Mais on ne connait pas l'état du sémaphore (vide ou occupé) :

$$\mathtt{emp} \vdash \{\, RI \ast sem \mapsto 0 \,\}\ \mathtt{set}(sem, 1)\ \{\, sem \mapsto 1 \ast RI \,\}\ \text{(vide)}$$

$$\mathtt{emp} \vdash \{\, RI \ast sem \mapsto 1 \ast RI \,\}\ \mathtt{set}(sem, 1)\ \{\, sem \mapsto 1 \ast RI \,\}\ \text{(occupé)}$$

Dans le 2e cas il faut $RI \ast RI \Rightarrow RI$, qui est vrai si RI est précise.

Codage des sections critiques

On peut utiliser un sémaphore comme un verrou :
P prend le verrou, V le rend.

D'où un codage simple des sections critiques :

$$\texttt{with}\ r\ \texttt{do}\ c \quad \stackrel{def}{=} \quad P(r); c; V(r)$$

où chaque section critique r est identifiée par l'adresse mémoire d'un sémaphore, initialisé à 1.

Codage des sections critiques

Si RI_r est l'invariant de la ressource r, l'invariant de mémoire partagée s'obtient en prenant la conjonction des invariants des sémaphores :

$$J_{\mathcal{R}} \quad = \quad \bigstar_{r \in \mathcal{R}} J(r, RI_r)$$

Ce codage valide la règle pour les sections critiques :

$$\frac{r \in \mathcal{R} \qquad J_{\mathcal{R} \setminus \{r\}} \vdash \{\, RI_r \bigstar P \,\}\ c\ \{\, RI_r \bigstar Q \,\}}{J_{\mathcal{R}} \vdash \{\, P \,\}\ \texttt{with}\ r\ \texttt{do}\ c\ \{\, Q \,\}}$$

Codage des sections critiques conditionnelles

Dans notre langage PTR, la condition c_b d'une s.c.c. est nécessairement une commande qui s'évalue en un booléen.

$$\texttt{with}\ r\ \texttt{when}\ c_b\ \texttt{do}\ c \quad \stackrel{def}{=} \quad P(r);\ wait(r, c_b);\ c;\ V(r)$$

où *wait* est la boucle suivante :

$$\begin{aligned} wait(r, c_b) \quad = \quad & \texttt{let}\ b = c_b\ \texttt{in} \\ & \texttt{if}\ b\ \texttt{then}\ 0\ \texttt{else}\ (V(r);\ P(r);\ wait(r, c_b)) \end{aligned}$$

On dérive la règle suivante :

$$\frac{\begin{array}{c} r \in \mathcal{R} \\ J_{\mathcal{R}\setminus\{r\}} \vdash \{\, RI_r \mathbin{\ast} P \,\}\ c_b\ \{\, \lambda b.\ \langle b \rangle \mathbin{\ast} B \vee \langle \neg b \rangle \mathbin{\ast} RI_r \mathbin{\ast} P \,\} \\ J_{\mathcal{R}\setminus\{r\}} \vdash \{\, B \,\}\ c\ \{\, RI_r \mathbin{\ast} Q \,\} \end{array}}{J_{\mathcal{R}} \vdash \{\, P \,\}\ \texttt{with}\ r\ \texttt{when}\ c_b\ \texttt{do}\ c\ \{\, Q \,\}}$$

Le schéma producteur/consommateur

Une généralisation de l'exemple «synchronisation et transfer de ressources», où plusieurs ressources sont transférées successivement.

`while true do`	`while true do`
calculer x;	`let` $y = \mathit{consume}()$ `in`
$\mathit{produce}(x)$;	utiliser y
`done`	`done`

Les ressources produites mais pas encore consommées sont stockées dans un tampon en mémoire partagée.

Note : on peut avoir plusieurs processus producteurs et plusieurs processus consommateurs.

Une solution avec un tampon de taille 1 et deux sémaphores

Trois variables en mémoire partagée :

- b : adresse du tampon (une case mémoire)
- s_1 : un sémaphore qui est à 1 lorsque le tampon est plein (contient une donnée produite pas encore consommée).
- s_0 : un sémaphore qui est à 1 lorsque le tampon est vide (ne contient pas de donnée produite et pas encore consommée)

Implémentation :

$$\mathit{produce}(b, s_0, s_1, x) = P(s_0);\ \texttt{set}(b, x);\ V(s_1)$$

$$\mathit{consume}(b, s_0, s_1) = P(s_1);\ \texttt{let}\ x = \texttt{get}(b)\ \texttt{in}\ V(s_0);\ x$$

Spécification et vérification du producteur/consommateur

On note $RI(x)$ l'invariant de ressources associé à la donnée x.

Spécification de *produce* et *consume* :

$$J(b) \vdash \{\, RI(x) \,\}\ produce(b, s_0, s_1, x)\ \{\, \mathtt{emp} \,\}$$
$$J(b) \vdash \{\, \mathtt{emp} \,\}\ consume(b, s_0, s_1)\ \{\, \lambda x.\ RI(x) \,\}$$

La vérification «passe» en prenant comme invariant J sur l'état partagé

$$J(b) \stackrel{def}{=} J(s_0, b \mapsto \_) \ast J(s_1, \exists x,\ b \mapsto x \ast RI(x))$$

En d'autres termes : quand le sémaphore s_0 est à 1, b est valide (on peut écrire dedans); quand le sémaphore s_1 est à 1, b contient une donnée x qui satisfait $RI(x)$.

Correction sémantique

Correction sémantique de la logique de séparation concurrente

La démonstration originale de Brookes (2004) :

- Sémantique dénotationnelle des commandes sous forme de traces d'actions.
- Une sémantique ≪locale≫ des actions et des traces qui met en évidence la possession des ressources et les transfers de ressources lors des sections critiques.
- Une hypothèse : tous les invariants de ressource sont précis.

La démonstration simplifiée de Vafeiadis (2011) :

- Raisonnement direct et élémentaire sur les suites de réductions, à l'aide d'un prédicat compté $\mathtt{Safe}^n\ c\ h$.
- Seule la règle de conjonction exige des invariants précis.

Quelques intuitions

$$J \vdash \{ P \} \, c \, \{ Q \}$$

Intuition déductive : c'est comme $\{ P \ast J \} \, c \, \{ Q \ast J \}$
avec en plus J invariant, c.à.d. tous les triplets apparaissant dans la dérivation sont de cette forme.

Intuition opérationnelle : à chaque étape de calcul, l'état mémoire courant h se décompose en trois parties disjointes :

$$h = h_1 \uplus h_j \uplus h_f$$

h_1 est la mémoire propre à c.
h_j est la mémoire partagée accessible aux sections atomiques.
h_f est la mémoire ≪encadrante≫, incluant les mémoires propres des processus s'exécutant en parallèle avec c.

Un triplet sémantique faible avec comptage de pas

On définit le triplet sémantique $J \models \{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}$:

$$J \models \{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\} \quad \stackrel{def}{=} \quad \forall n, h,\ P\ h \Rightarrow \mathtt{Safe}^n\ c\ h\ Q\ J$$

Le prédicat inductif $\mathtt{Safe}^n\ c\ h\ Q\ J$ signifie que les exécutions de c dans la mémoire propre h
– ne font pas d'erreur dans les n premières étapes d'exécution ;
– satisfont Q si elles terminent en au plus n étapes ;
– préservent l'invariant J sur la mémoire partagée.

$$\mathtt{Safe}^0\ c\ h\ Q\ J \qquad \frac{Q\ [\![a]\!]\ h}{\mathtt{Safe}^{n+1}\ a\ h\ Q\ J} \qquad \frac{(\forall a, c \neq a) \qquad \cdots}{\mathtt{Safe}^{n+1}\ c\ h\ Q\ J}$$

Un triplet sémantique faible avec comptage de pas

$$\frac{\begin{array}{l}\forall a, c \neq a \\ \forall h_j, h_f,\ J\ h_j \Rightarrow c/h_1 \uplus h_j \uplus h_f \not\to \texttt{err} \\ \forall h_j, h_f, c', h',\ J\ h_j \wedge c/h_1 \uplus h_j \uplus h_f \to c'/h' \Rightarrow \\ \qquad \exists h'_1, h'_j,\ h' = h'_1 \uplus h'_j \uplus h_f \wedge J\ h'_j \wedge \texttt{Safe}^n\ c'\ h'_1\ Q\end{array}}{\texttt{Safe}^{n+1}\ c\ h_1\ Q}$$

Le cas récursif : c dans h_1 est sûre pour $n+1$ étapes si

Un triplet sémantique faible avec comptage de pas

$$\frac{\begin{array}{l}\forall a, c \neq a \\ \forall h_j, h_f,\ J\ h_j \Rightarrow c/h_1 \uplus h_j \uplus h_f \not\to \texttt{err} \\ \forall h_j, h_f, c', h',\ J\ h_j \wedge c/h_1 \uplus h_j \uplus h_f \to c'/h' \Rightarrow \\ \qquad \exists h'_1, h'_j,\ h' = h'_1 \uplus h'_j \uplus h_f \wedge J\ h'_j \wedge \texttt{Safe}^n\ c'\ h'_1\ Q\end{array}}{\texttt{Safe}^{n+1}\ c\ h_1\ Q}$$

Le cas récursif : c dans h_1 est sûre pour $n+1$ étapes si

- dans tout état h de la forme $h_1 \uplus h_j \uplus h_f$ avec h_j satisfaisant J, c/h ne fait pas d'erreurs, et …

Un triplet sémantique faible avec comptage de pas

$$\frac{\begin{array}{l}\forall a, c \neq a \\ \forall h_j, h_f,\ J\ h_j \Rightarrow c/h_1 \uplus h_j \uplus h_f \not\to \texttt{err} \\ \forall h_j, h_f, c', h',\ J\ h_j \wedge c/h_1 \uplus h_j \uplus h_f \to c'/h' \Rightarrow \\ \qquad \exists h'_1, h'_j,\ h' = h'_1 \uplus h'_j \uplus h_f \wedge J\ h'_j \wedge \texttt{Safe}^n\ c'\ h'_1\ Q\end{array}}{\texttt{Safe}^{n+1}\ c\ h_1\ Q}$$

Le cas récursif : c dans h_1 est sûre pour $n+1$ étapes si

- dans tout état h de la forme $h_1 \uplus h_j \uplus h_f$ avec h_j satisfaisant J, c/h ne fait pas d'erreurs, et …
- pour toute réduction $c/h \to c'/h'$, l'état h' se décompose en $h'_1 \uplus h'_j \uplus h_f$ avec h'_j satisfaisant J,
  et de plus c' dans h'_1 est sûre pour n étapes.

Correction sémantique et décompositions de l'état mémoire

On montre sans trop de peine que ce triplet sémantique $J \models \{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}$ satisfait les règles de la logique de séparation concurrente.

Voici une illustration de la décomposition $h = h_1 \uplus h_j \uplus h_f$ à utiliser pour chacune des principales règles :

$$\frac{\mathtt{emp} \vdash \{ P \star J \}\ c\ \{ Q \star J \}}{J \vdash \{ P \}\ \mathtt{atomic}\ c\ \{ Q \}} \qquad \frac{(h_1 \uplus h_j) \uplus \emptyset \uplus h_f}{h_1 \uplus h_j \uplus h_f}$$

$$\frac{J \star J' \vdash \{ P \}\ c\ \{ Q \}}{J \vdash \{ P \star J' \}\ c\ \{ \lambda v.\, Q\, v \star J' \}} \qquad \frac{h_1 \uplus (h_j \uplus h_2) \uplus h_f}{(h_1 \uplus h_2) \uplus h_j \uplus h_f}$$

Correction sémantique et décompositions de l'état mémoire

$$\frac{J \vdash \{\, P_1 \,\}\ c_1\ \{\, \lambda\_.\ Q_1 \,\} \qquad J \vdash \{\, P_2 \,\}\ c_2\ \{\, \lambda\_.\ Q_2 \,\}}{J \vdash \{\, P_1 \mathbin{\ast} P_2 \,\}\ c_1 \parallel c_2\ \{\, \lambda\_.\ Q_1 \mathbin{\ast} Q_2 \,\}}$$

$$\frac{h_1 \uplus h_j \uplus (h_f \uplus h_2) \quad \text{ou} \quad h_2 \uplus h_j \uplus (h_f \uplus h_1)}{(h_1 \uplus h_2) \uplus h_j \uplus h_f}$$

$$\frac{J \vdash \{\, P \,\}\ c\ \{\, Q \,\}}{J \mathbin{\ast} J' \vdash \{\, P \,\}\ c\ \{\, Q \,\}}$$

$$\frac{h_1 \uplus h_j \uplus (h_f \uplus h'_j)}{h_1 \uplus (h_j \uplus h'_j) \uplus h_f}$$

$$\frac{J \vdash \{\, P \,\}\ c\ \{\, Q \,\}}{J \vdash \{\, P \mathbin{\ast} R \,\}\ c\ \{\, \lambda v.\ Q\ v \mathbin{\ast} R \,\}}$$

$$\frac{h_1 \uplus h_j \uplus (h_f \uplus h_2)}{(h_1 \uplus h_2) \uplus h_j \uplus h_f}$$

Absence de courses critiques

$$(c_1 \parallel c_2)/h \to \texttt{err} \quad \text{si} \quad Acc(c_1) \cap Acc(c_2) \neq \emptyset$$

Si on ajoute la règle d'erreur ci-dessus et que l'on prend

$$Acc(\texttt{atomic}\ c) = \emptyset,$$

la démonstration de correction sémantique ≪passe≫ encore, ce qui montre :

> *Toute commande c prouvable en logique de séparation concurrente ne contient aucune course critique entre accès mémoire non atomiques.*

Note : $\texttt{atomic}(\texttt{set}(p, 1)) \parallel \texttt{atomic}(\texttt{set}(p, 2))$ est prouvable mais n'est pas considéré comme une course critique.

Point d’étape

Point d'étape

Après l'éclair de la logique de séparation (2001), le coup de tonnerre de la logique de séparation concurrente (2004).

Par rapport aux logiques précédentes (p.ex. Owiki & Gries, 1976), un énorme progrès pour montrer des propriétés de sûreté des calculs parallèles :

- absence de courses critiques ;
- sûreté de la mémoire ; (pas d'accès après `free`, pas de double `free`)
- intégrité des structures de données ;
- Transfers de données entre processus.

Encore du progrès à faire sur la correction fonctionnelle, p.ex.

$$\{ x = 0 \}\ \texttt{atomic}(x := x + 1) \parallel \texttt{atomic}(x := x + 1)\ \{ x = 2 \}$$

Bibliographie

Bibliographie

Un livre de référence sur le parallélisme à mémoire partagée :

- M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming*, Morgan Kaufman, 2012.

L'article fondateur de la logique de séparation concurrente (version révisée) :

- P. O'Hearn, *Resources, Concurrency and Local Reasoning*, Theor. Comp. Sci, 2007.

La démonstration simple de la correction sémantique :

- V. Vafeiadis, *Concurrent separation logic and operational semantics*, MFPS 2011

Mécanisations :

- Le développement Coq correspondant à ce cours : `https://github.com/xavierleroy/cdf-program-logics`
- L'infrastructure Iris : `https://iris-project.org/`