Logiques de programmes, sixième cours

Logiques pour la mémoire faiblement cohérente

Xavier Leroy

2021-04-08

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

La cohérence séquentielle : un modèle idéalisé de programmation parallèle

La cohérence séquentielle (*sequential consistency*)

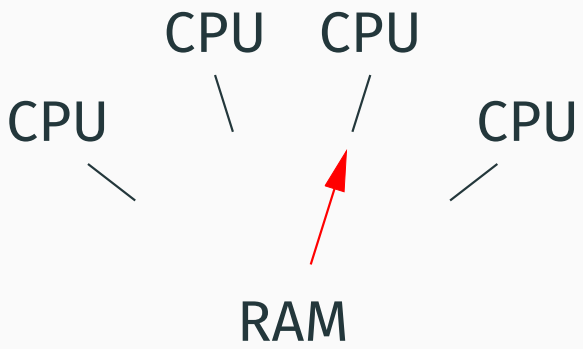
> *A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program*
>
> *(L. Lamport, 1978)*

Pour un système à mémoire partagée : l'état de la mémoire partagée est le résultat d'un entrelacement des opérations des différents processus.

Deux implémentations séquentiellement cohérentes (SC)

1. Temps partagé sur un unique processeur.

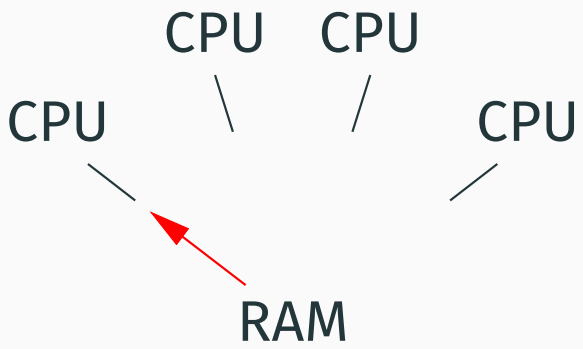
2. Système multiprocesseur avec un seul «guichet» d'accès à la mémoire.



Deux implémentations séquentiellement cohérentes (SC)

1. Temps partagé sur un unique processeur.

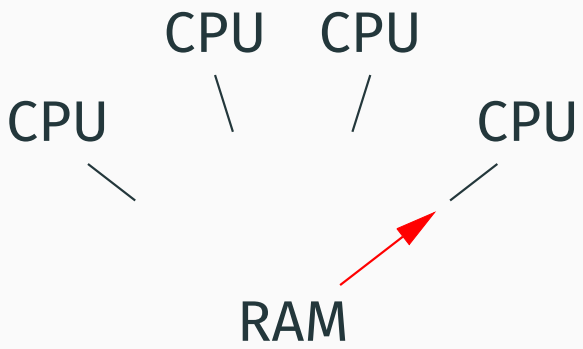
2. Système multiprocesseur avec un seul ≪guichet≫ d'accès à la mémoire.



Deux implémentations séquentiellement cohérentes (SC)

1. Temps partagé sur un unique processeur.

2. Système multiprocesseur avec un seul «guichet» d'accès à la mémoire.



Une sémantique formelle très naturelle

Rappel du cours 4 :

Sémantique de la construction $c_1 \parallel c_2$:
un entrelacement des réductions de c_1 et c_2.

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

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

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

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

Notre sémantique du parallélisme en PTR était SC sans le savoir !

Programmer dans un modèle SC

Le modèle SC définit précisément la sémantique des programmes parallèles même lorsqu'ils contiennent des courses critiques.

Cela ouvre la voie à des algorithmes parallèles qui utilisent des courses critiques de manière bien contrôlée.

Ces algorithmes sont utiles lorsque les instructions atomiques du processeur ou les sections critiques du langage ne sont pas disponibles ou sont trop coûteuses.

L'algorithme d'exclusion mutuelle de Peterson

$$flag : bool[2] = \{\texttt{false}, \texttt{false}\}; turn : \{0, 1\};$$

$flag[0] := \texttt{true};$	$flag[1] := \texttt{true};$
$turn := 1;$	$turn := 0;$
$\texttt{while}\ flag[1] \wedge turn = 1$	$\texttt{while}\ flag[0] \wedge turn = 0$
$\texttt{do skip done};$	$\texttt{do skip done};$
// début de section critique	// début de section critique
...	...
// fin de section critique	// fin de section critique
$flag[0] := \texttt{false}$	$flag[1] := \texttt{false}$

L'algorithme d'exclusion mutuelle de Peterson

$$flag : bool[2] = \{\texttt{false}, \texttt{false}\}; turn : \{0, 1\};$$

$flag[0] := \texttt{true};$	$flag[1] := \texttt{true};$
$turn := 1;$	$turn := 0;$
$\texttt{while}\ flag[1] \wedge turn = 1$	$\texttt{while}\ flag[0] \wedge turn = 0$
$\texttt{do skip done};$	$\texttt{do skip done};$
// début de section critique	// début de section critique
...	...
// fin de section critique	// fin de section critique
$flag[0] := \texttt{false}$	$flag[1] := \texttt{false}$

Par énumération des entrelacements SC, on peut montrer que les deux processus sont simultanément en section critique seulement si $flag[0] = flag[1] = \texttt{true} \wedge turn = 0 \wedge turn = 1$, ce qui est impossible.

Le verrouillage par ticket (*ticket lock*)

Un processus qui veut entrer en section critique

- Prend le prochain ticket (incrément atomique)
- Attend que le numéro sur son ticket soit affiché.

Ayant fini sa section critique, il fait afficher le numéro suivant.

Le verrouillage par ticket (*ticket lock*)

Deux variables globales : `next` et `now_serving`, initialement 0.

```
lock() {
    int t = fetch_and_add(&next, 1);      // incrément atomique
    while (now_serving != t) pause();     // lecture non atomique
}

unlock() {
    now_serving = now_serving + 1;        // incrément non atomique
}
```

L'incrément de `next` doit être atomique
(sinon deux processus pourraient avoir le même ticket).

Les accès à `now_serving` peuvent être non atomiques.

La réalité :
les modèles mémoire faiblement cohérents

Une épreuve de vérité (*litmus test*)

Un bout de l'algorithme de Dekkers pour l'exclusion mutuelle :

$$\begin{array}{l} \texttt{set}(X, 1); \\ \texttt{let}\ a = \texttt{get}(Y) \end{array} \quad \Big\| \quad \begin{array}{l} \texttt{set}(Y, 1); \\ \texttt{let}\ b = \texttt{get}(X) \end{array}$$

Initialement, $X = Y = 0$.

Dans le modèle SC :

- soit $\texttt{set}(X, 1)$ est exécuté en premier, et alors $b = 1$ à la fin ;
- soit $\texttt{set}(Y, 1)$ est exécuté en premier, et alors $a = 1$ à la fin.

Donc, l'état final $a = b = 0$ est impossible.

Réfutation expérimentale

On écrit le test en assembleur x86 (pour bien contrôler le code) :

```
X86_64 Dekker
{ want0=0; want1=0; }
  P0                  | P1 ;
  movl $1,(want0)     | movl $1,(want1) ;
  movl (want1),%eax   | movl (want0),%eax ;
exists (0:rax=0 /\ 1:rax=0)
```

On l'exécute avec l'outil `litmus7` :

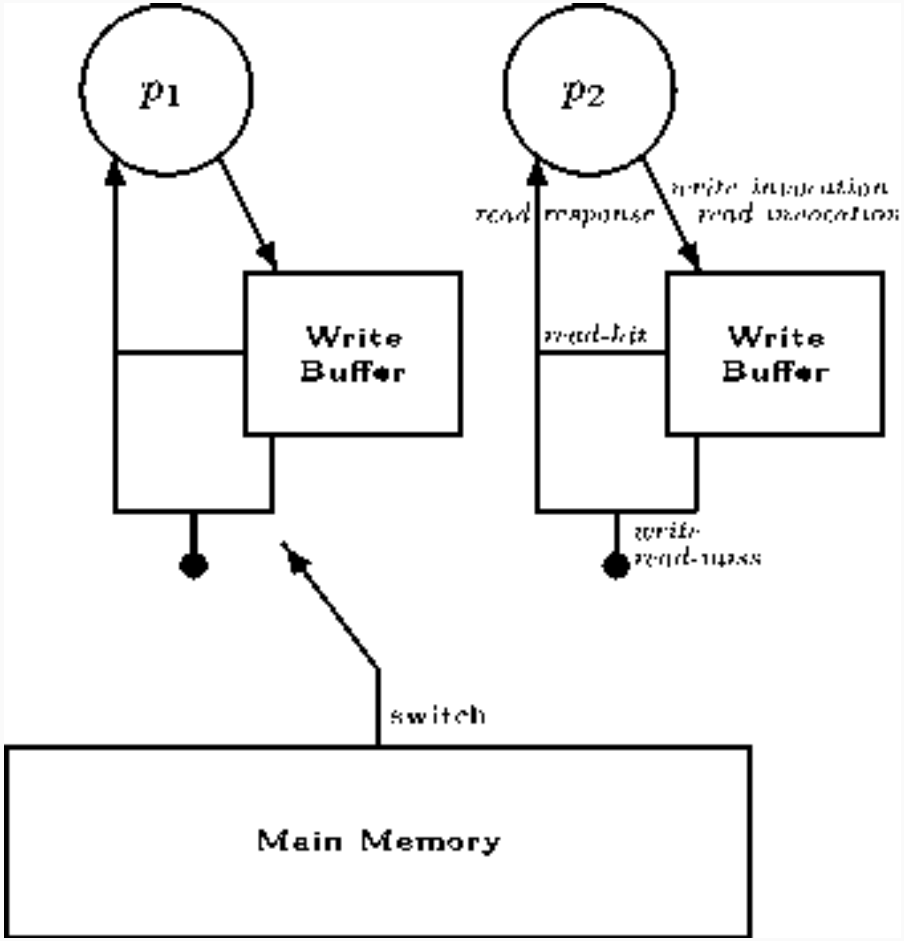
```
Test Dekker Allowed
Histogram (4 states)
178     *>0:rax=0; 1:rax=0;
1999870:>0:rax=1; 1:rax=0;
1999881:>0:rax=0; 1:rax=1;
71      :>0:rax=1; 1:rax=1;
```

La dure réalité

Le modèle de cohérence séquentielle n'est respecté

- ni par les architectures matérielles contemporaines
  (afin d'offrir des systèmes mémoire plus performants),
- ni par les compilateurs optimisants
  (afin d'améliorer les performances du code produit).

Matériel : les tampons en écriture (*write buffers, store buffers*)



(Higham, Jackson, Kawash, 2007)

Chaque processeur place ses écritures dans un tampon en attendant qu'elles soient transmises à la mémoire partagée.

Les écritures d'un processeur sont visibles immédiatement de ce processeur, mais pas immédiatement des autres processeurs.

Une exécution non-SC

$$\texttt{set}(X, 1); \texttt{let } a = \texttt{get}(Y) \;\Big\|\; \texttt{set}(Y, 1); \texttt{let } b = \texttt{get}(X)$$

	Processeur 1	Processeur 2
Temps $t = 0$	met $X \leftarrow 1$ dans son tampon	met $Y \leftarrow 1$ dans son tampon
Temps $t = 1$	lit $Y = 0$ depuis la mémoire partagée	
Temps $t = 2$		lit $X = 0$ depuis la mémoire partagée
Temps $t = 3$	transmet $X \leftarrow 1$ à la mémoire partagée	
Temps $t = 4$		transmet $Y \leftarrow 1$ à la mémoire partagée

Matériel : exécution dans le désordre (*out of order*) et spéculative

Le processeur peut réordonner au vol les instructions (*out-of-order execution*), afin de démarrer plus tôt les instructions qui prennent longtemps (p.ex. les lectures mémoire).

$$\text{écrire } X; \ldots; \text{lire } Y \rightarrow \text{lire } Y; \text{écrire } X; \ldots$$

Cette exécution dans le désordre est souvent spéculative : si le processeur se rend compte que $X = Y$, il annule la lecture anticipée de Y, ou la satisfait par l'écriture de X (*forwarding*).

Une autre exécution non-SC

Code machine :

$$\begin{array}{l} \texttt{set}(X, 1); \\ \texttt{let } a = \texttt{get}(Y) \end{array} \quad \Big\| \quad \begin{array}{l} \texttt{set}(Y, 1); \\ \texttt{let } b = \texttt{get}(X) \end{array}$$

Code exécuté par le processeur après réordonnancement au vol :

$$\begin{array}{l} \texttt{let } a = \texttt{get}(Y) \texttt{ in} \\ \texttt{set}(X, 1) \end{array} \quad \Big\| \quad \begin{array}{l} \texttt{let } b = \texttt{get}(X) \texttt{ in} \\ \texttt{set}(Y, 1) \end{array}$$

On peut évidemment terminer avec $a = b = 0$.

Matériel : accès mémoire fractionnés

Une donnée mal alignée peut être à cheval sur deux lignes de cache, et nécessiter deux accès mémoire pour l'écrire ou pour la lire.

$$\texttt{set}(X, 0x12345678) \;\Big\|\; \texttt{let}\ a = \texttt{get}(X)$$

peut être exécuté comme

$$\begin{array}{l} \texttt{set}(X_1, 0x1234); \\ \texttt{set}(X_2, 0x5678); \end{array} \;\Big\|\; \begin{array}{l} \texttt{let}\ a_1 = \texttt{get}(X_1)\ \texttt{in} \\ \texttt{let}\ a_2 = \texttt{get}(X_2)\ \texttt{in} \\ \texttt{let}\ a = a_1 << 16 \mid a_2 \end{array}$$

Une exécution non-SC avec valeurs *ex nihilo* (*values out of thin air*)

$$\texttt{set}(X, 0x12345678) \;\Big\|\; \texttt{let}\ a = \texttt{get}(X)$$

Partant de $X = 0$, il y a deux exécutions SC :
$a = 0$ ou $a = 0x12345678$.

$$\begin{array}{l} \texttt{set}(X_1, 0x1234); \\ \texttt{set}(X_2, 0x5678); \end{array} \;\Big\|\; \begin{array}{l} \texttt{let}\ a_1 = \texttt{get}(X_1)\ \texttt{in} \\ \texttt{let}\ a_2 = \texttt{get}(X_2)\ \texttt{in} \\ \texttt{let}\ a = a_1 << 16 \mid a_2 \end{array}$$

Après fractionnement des accès mémoire, un troisième résultat est possible : $a = 0x12340000$, provenant de $a_1 = 0x1234$ et $a_2 = 0$.

Note : la valeur $0x12340000$ n'apparaît pas dans le code initial !

Des remèdes possibles

Utiliser des instructions barrières qui empêchent le processeur de réordonner certains accès en mémoire :

- Barrière ≪forte≫ : préserve l'ordre entre les accès avant la barrière et ceux après la barrière.
- Barrière ≪faible≫ : préserve l'ordre entre les lectures avant la barrière et les accès après la barrière.

Autres instructions avec des comportements mémoire spéciaux :

- instructions ≪verrouillées≫ (préfixe `lock` en x86);
- *load-acquire* et *store-release* (Itanium, ARM);
- etc.

Optimisations du compilateur : réordonnancement d'instructions

Le compilateur peut réordonner des lectures et des écritures indépendantes (à des adresses X, Y garanties différentes).

Typiquement, les lectures sont anticipées et les écritures sont retardées.

$$\text{écrire } X; \ldots; \text{lire } Y \rightarrow \text{lire } Y; \text{écrire } X; \ldots$$

$\rightarrow$ Mêmes comportements non-SC que le réordonnancement dynamique fait par certains processeurs.

Optimisations du compilateur : factorisation des lectures

Un cas particulier de *Common Subexpression Elimination*.

$\texttt{let } a = \texttt{get}(X) \texttt{ in}$		$\texttt{let } a = \texttt{get}(X) \texttt{ in}$
$\ldots$		$\ldots$
(pas d'écriture de X)	$\rightsquigarrow$	$\ldots$
$\ldots$		$\ldots$
$\texttt{let } b = \texttt{get}(X) \texttt{ in}$		$\texttt{let } b = a \texttt{ in}$
$\ldots$		$\ldots$

Une exécution non-SC

$$\begin{array}{l} \texttt{let } a = \texttt{get}(X) \texttt{ in} \\ \texttt{let } b = \texttt{get}(Y) \texttt{ in} \\ \texttt{let } c = \texttt{get}(X) \end{array} \;\Big\|\; \begin{array}{l} \texttt{set}(X, 1); \\ \texttt{set}(Y, 1); \end{array}$$

Avec $X = Y = 0$ initialement, aucune exécution SC ne termine avec $(a, b, c) = (0, 1, 0)$.

Après factorisation du $\texttt{get}(X)$, on a $\texttt{let } c = a$ et le résultat $(a, b, c) = (0, 1, 0)$ est possible.

Optimisations du compilateur : déplacement de code invariant

Un même calcul effectué à chaque tour de boucle peut être effectué avant la boucle :

$$
\begin{array}{l}
\texttt{for } i = 0 \texttt{ to } 99 \texttt{ do} \\
\quad A[i] := i + j \times 10 \\
\texttt{done}
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{l}
t := j \times 10; \\
\texttt{for } i = 0 \texttt{ to } 99 \texttt{ do} \\
\quad A[i] := i + t \\
\texttt{done}
\end{array}
$$

Cela peut «casser» les codes qui font de l'attente active :

$$
\begin{array}{l}
\texttt{do} \\
\quad t := \texttt{get}(X) \\
\texttt{while } t = 0
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{l}
t := \texttt{get}(X); \\
\texttt{do} \\
\quad \texttt{skip} \\
\texttt{while } t = 0
\end{array}
$$

Des remèdes possibles

Désactiver les optimisations ? Jamais !

Informer le compilateur des accès mémoire qui constituent des communications inter-processus, afin de les compiler spécialement :

- modificateur `volatile` (C,C++, Java)
- bibliothèque d'opérations atomiques (C/C++ 2011)
- etc.

Les *low-level atomics* en C/C++ 2011

Diverses opérations atomiques : lecture, écriture, *fetch-and-add*, *compare-and-swap*, ...

Chaque opération est annotée par le mode de cohérence mémoire exigé :

`memory_order_seq_cst`	cohérence séquentielle
`memory_order_acq_rel`	juste assez pour le passage de messages
`memory_order_acquire`	
`memory_order_release`	
`memory_order_consume`	
`memory_order_relaxed`	aucune garantie hors l'atomicité

DRF + CSL = ♡

Logique de séparation concurrente et garantie DRF

La garantie DRF (*Data Race Free*)

Une propriété d'un modèle mémoire relâché :

> *Si un programme s'exécute dans le modèle SC sans faire de courses critiques,*
>
> *alors il s'exécute dans le modèle relâché exactement comme dans le modèle SC.*

En d'autres termes : pour un programme sans courses critiques, les relaxations du modèle mémoire n'ajoutent pas de comportements par rapport aux comportements SC.

Cette garantie DRF semble vraie de / est revendiquée par tous les modèles mémoires connus (de processeurs ou de langages).

La garantie DRF étendue

Si un programme comprenant des sections critiques, des opérations atomiques, et autres dispositifs de synchronisation s'exécute dans le modèle SC sans faire de courses critiques,

alors il s'exécute dans le modèle relâché exactement comme dans le modèle SC,

à condition que les dispositifs de synchronisation soient correctement implémentés.

≪Correctement implémentés≫ = avec les barrières mémoire qui suffisent pour garantir l'absence de comportements non-SC.

Exemples d'implémentations de verrous

x86 : lock

```
     movl  $1, %edx
.L2: movl  %edx, %eax
     xchgb (%rdi), %al
     testb %al, %al
     jne   .L2
```

unlock

```
     movb    $0, (%rdi)
```

Power : lock

```
.L2: lbarx  9,0,3
     stbcx. 10,0,3
     bne    0,.L2
     isync
     andi.  9,9,0xff
     bne    0,.L2
```

unlock

```
     lwsync
     li  9,0
     stb 9,0(3)
```

Un point subtil de l'implémentation x86

Avant 1999, le noyau Linux implémentait `unlock` avec une instruction atomique

```
lock; btr $0, (...)
```

au lieu d'une simple écriture

```
movb $0, (...)
```

Une longue discussion a conclut qu'une simple écriture suffit car l'architecture x86 est de type TSO :

L'écriture de 0 dans le verrou n'est pas immédiatement visible des autres processus qui attendent de prendre le verrou, mais lorsqu'elle devient visible, toutes les écritures précédentes sont déjà visibles et les lectures précédentes sont déjà effectuées.

Compatibilité avec les optimisations de compilateurs

Un grand nombre d'optimisations à la compilation sont correctes pour les programmes sans courses critiques.

(Correctes = tous les comportements du programme optimisé sont des comportements possibles du programme source. L'optimisation n'a pas introduit de nouveaux comportements.)

Compatibilité avec les optimisations de compilateurs

Transformation	SC	DRF guarantee	JMM
Trace-preserving transformations	✓	✓	✓
Reordering normal memory accesses	×	✓	×
Redundant read after read elimination	✓	✓	×
Redundant read after write elimination	✓	✓	✓
Irrelevant read elimination	✓	✓	✓
Irrelevant read introduction	✓	?	×
Redundant write before write elimination	✓	✓	✓
Redundant write after read elimination	✓	✓	×
Roach-motel reordering	×(✓for locks)	✓	×
External action reordering	×	✓	×

J. Ševčik, *Program Transformations in Weak Memory Models*, PhD, 2008.

Comment montrer l'absence de courses critiques ?

L'hypothèse ≪le programme n'a pas de courses critiques dans le modèle SC≫ est forte ! Comment l'établir ?

- Démonstration *ad hoc*.
- Système de types (≈ Rust).
- Analyse statique (Infer, etc).
- Vérification déductive en logique de séparation concurrente !

Rappel du 4e cours : si $J \vdash \{ P \}\ c\ \{ Q \}$, alors c s'exécute sans courses critiques, dans une sémantique par entrelacements équivalente à SC.

La garantie CSL (*Concurrent Separation Logic*)

Si un programme est vérifiable en logique de séparation concurrente (incluant sections critiques, sections atomiques, etc),

alors il s'exécute correctement dans un modèle mémoire relâché qui offre la garantie DRF,

à condition que les sections critiques / atomiques soient correctement implémentées.

Une démonstration directe de la garantie CSL pour un modèle TSO

Nous allons montrer la correction sémantique de la logique de séparation concurrente pour une variante de notre langage PTR munie de tampons d'écriture (modèle appelé TSO) :

Tampons (*store buffers*) $\quad s ::= \varepsilon \mid (\ell, v) \cdot s$

On considère des configurations globales (du programme entier)

$$(\ (c_1/s_1) \parallel \cdots \parallel (c_n/s_n) \ ) \ / \ h$$

composées de n processus $c_1 \ldots c_n$, chacun avec son tampon s_i, le tout dans une mémoire globale h.

On considère aussi des configurations locales (d'un processus)

$$c/s/h$$

Réduction locales

À tout moment un processus peut effectuer l'écriture la plus ancienne de son tampon :

$$c/s \cdot (\ell, v)/h \to c/s/h[\ell \leftarrow v]$$

Les constructions de base ont leur sémantique habituelle :

$$\begin{aligned}
(\texttt{let}\ x = a\ \texttt{in}\ c)/s/h &\to c[x \leftarrow [\![a]\!]]/s/h \\
(\texttt{let}\ x = c_1\ \texttt{in}\ c_2)/s/h &\to (\texttt{let}\ x = c_1'\ \texttt{in}\ c_2)/s'/h' \\
&\qquad \text{si } c_1/s/h \to c_1'/s'/h' \\
(\texttt{let}\ x = c_1\ \texttt{in}\ c_2)/s/h &\to \texttt{err} \quad \text{si } c_1/s/h \to \texttt{err}
\end{aligned}$$

Réductions locales : constructions impératives

Les constructions impératives écrivent dans s (le tampon) et lisent dans $s \rhd h$, le tas h mis à jour d'après s :

$$\varepsilon \rhd h = h \qquad ((\ell, v) \cdot s) \rhd h = (s \rhd h)[\ell \leftarrow v]$$

$$\mathtt{get}(a)/s/h \to (s \rhd h)([\![a]\!])/s/h \quad \text{si } [\![a]\!] \in \mathit{Dom}(s \rhd h)$$

$$\mathtt{set}(a, a')/s/h \to 0/([\![a]\!], [\![a']\!]) \cdot s/h \quad \text{si } [\![a]\!] \in \mathit{Dom}(s \rhd h)$$

$$\mathtt{get}(a)/s/h \to \mathtt{err} \quad \text{si } [\![a]\!] \notin \mathit{Dom}(s \rhd h)$$

$$\mathtt{set}(a, a')/s/h \to \mathtt{err} \quad \text{si } [\![a]\!] \notin \mathit{Dom}(s \rhd h)$$

Réductions locales : sections atomiques

Comme dans PTR, les sections atomiques s'exécutent comme un seul ≪grand pas≫ :

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

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

Cependant, il faut que le tampon soit vide à la fin de la section atomique ($\approx$ on met une barrière en écriture à la fin).

Lorsqu'on crée un invariant de ressources, il faut aussi que le tampon soit vide, d'où la construction $\texttt{mkinv}(c)$:

$$\texttt{mkinv}(c)/\varepsilon/h \to c/\varepsilon/h$$

Réductions globales

À chaque étape on réduit localement un des processus c_i/s_i de la composision parallèle, les autres étant inchangés.

$$\frac{c_i/s_i/h \to c'/s'/h'}{(\cdots \parallel (c_i/s_i) \parallel \cdots) \,/\, h \to (\cdots \parallel (c'/s') \parallel \cdots) \,/\, h'}$$

$$\frac{c_i/s_i/h \to \texttt{err}}{(\cdots \parallel (c_i/s_i) \parallel \cdots) \,/\, h \to \texttt{err}}$$

Correction sémantique de la logique de séparation concurrente

Pour PTR dans le modèle SC, on avait décomposé l'état mémoire courant h 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.

Correction sémantique de la logique de séparation concurrente

Pour PTR dans le modèle TSO, on va décomposer l'état mémoire courant h et le tampon s de la commande c comme suit :

$$h = h_u \uplus h_j \qquad s \rhd h_u = h_1 \uplus h_f$$

La mémoire principale h se décompose en mémoire partagée h_j et mémoire non partagée h_u.

La mémoire non partagée h_u, mise à jour d'après le tampon s, se décompose en h_1, la mémoire propre à c, et h_f, l'encadrement.

Présentation équivalente :

$$s \rhd h = h_1 \uplus h_j \uplus h_f \quad \text{avec} \quad Dom(s) \cap Dom(h_j) = \emptyset$$

Le triplet sémantique

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

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

Comme au 4e cours, on a :

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

Le triplet sémantique

Le cas récursif : une étape de réduction de $c/s/h$.

$$\forall a, c \neq a$$

$$\forall s, h, h_j, h_f, s \rhd h = h_1 \uplus h_j \uplus h_f \wedge Dom(s) \cap Dom(h_j) = \emptyset \wedge J\ h_j \Rightarrow c/s/h \not\to \texttt{err}$$

$$\begin{aligned}\forall & s, h, h_j, h_f, c', s', h', \\ & s \rhd h = h_1 \uplus h_j \uplus h_f \wedge Dom(s) \cap Dom(h_j) = \emptyset \\ & \wedge J\ h_j \wedge c/s/h \to c'/s'/h' \\ & \Rightarrow \exists h'_1, h'_j,\ s' \rhd h' = h'_1 \uplus h'_j \uplus h_f \wedge Dom(s') \cap Dom(h'_j) = \emptyset \\ & \qquad \wedge J\ h'_j \wedge \texttt{Safe}^n\ c'\ h'_1\ Q\end{aligned}$$

$$\texttt{Safe}^{n+1}\ c\ h_1\ Q$$

Vérification des règles de la logique

Il reste à montrer que ce triplet sémantique $J \models \{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}$ satisfait les règles de la logique de séparation concurrente. Les deux cas intéressants concernent les sections atomiques.

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

À la fin de l'exécution de c on a une décomposition $\varepsilon \rhd h' = (h'_1 \uplus h'_j) \uplus \emptyset \uplus h_f$ que l'on réécrit en $\varepsilon \rhd h' = h'_1 \uplus h'_j \uplus h_f$. Il est crucial que le tampon final s' soit ε car sinon la contrainte $Dom(s') \cap Dom(h'_j) = \emptyset$ ne serait pas satisfaite.

Vérification des règles de la logique

Concernant l'ajout d'un invariant J' dans J :

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

Au début de l'exécution on a une décomposition
$s \rhd h = (h_1 \uplus h'_j) \uplus h_j \uplus h_f$ que l'on réécrit en
$s \rhd h = h_1 \uplus (h_j \uplus h'_j) \uplus h_f$.

Là aussi, il faut forcer $s = \varepsilon$ pour satisfaire
$Dom(s) \cap Dom(h_j \uplus h'_j) = \emptyset$ à coup sûr.

C'est la construction `mkinv` qui force à vider le tampon.

Une limitation et une force

Une limitation : notre formalisation ne permet pas de justifier l'implémentation de $\texttt{unlock}(\ell) = \texttt{atomic}(\texttt{set}(\ell, 0))$ par une écriture normale, sans vidage du tampon.

C'est un aspect de TSO que nous n'avons pas capturé.

Une force : cela rend notre démonstration adaptable à des modèles mémoires plus relâchés que TSO, en particulier PSO (*Partial Store Ordering*), où cette optimisation de `unlock` n'est pas valide.

De TSO à PSO

Dans le modèle PSO, les écritures à des adresses différentes peuvent être réordonnées, et donc ≪sortir≫ du tampon en écriture dans un ordre différent de l'ordre d'exécution du processus :

$$c/s_1 \cdot (\ell, v) \cdot s_2/h \to c/s_1 \cdot s_2/h[\ell \leftarrow v] \quad \text{si } \ell \notin Dom(s_2)$$

Ici, l'écriture en ℓ ≪passe devant≫ les écritures s_2.

Cela ne change rien à la correction sémantique de la logique, car

$$(s_1 \cdot (\ell, v) \cdot s_2) \rhd h = (s_1 \cdot s_2) \rhd (h[\ell \leftarrow v]) \quad \text{si } \ell \notin Dom(s_2)$$

Une logique pour *release-acquire*

Le modèle *release-acquire*

Une écriture de type *release* garantit que les écritures et lectures précédentes ont été effectuées.

(Pas de réordonnancement de $X; W_{rel}$ en $W_{rel}; X$.)

Une lecture de type *acquire* garantit que les lectures et écritures suivantes n'ont pas commencé.

(Pas de réordonnancement de $R_{acq}; X$ en $X; R_{acq}$.)

Utilisation pour le passage de messages

```
// préparation du message
// écritures non atomiques (na)
set_na(msg, ...);
set_na(msg + 1, ...);
set_na(msg + 2, ...);
// envoi du message
set_rel(ready, 1)
```

```
// attente du message
// lire ready jusqu'à ≠ 0
while get_acq(ready) = 0 do skip;
// lecture du message
// lectures non atomiques
let x = get_na(msg) in
...
```

Une forme légère de synchronisation et de transmission de ressources, sans exclusion mutuelle.

Utilisation pour implémenter les verrous

Libérer un verrou est une simple écriture de type *release* :

$$\mathit{unlock}(\ell) = \mathtt{set}_{\mathit{rel}}(\ell, 0)$$

Prendre un verrou nécessite une instruction atomique, p.ex. *Compare And Swap*, en mode *acquire* :

$$\mathit{lock}(\ell) = \mathtt{while}\ (\mathtt{CAS}_{\mathit{acq}}(\ell, 0, 1) \neq 0)\ \mathtt{do}\ \mathtt{skip}$$

On peut améliorer les performances avec une attente active utilisant des lectures de type *relaxed* :

$$\mathit{spin}(\ell) = \mathtt{while}\ (\mathtt{get}_{\mathit{rlx}}(\ell) \neq 0)\ \mathtt{do}\ \mathtt{skip}$$
$$\mathit{lock}(\ell) = \mathtt{do}\ \mathit{spin}(\ell)\ \mathtt{while}\ (\mathtt{CAS}_{\mathit{acq}}(\ell, 0, 1) \neq 0)$$

Implémentation du modèle *release/acquire*

Pour une architecture TSO comme x86 : rien à faire !

- Les écritures normales ont la sémantique *release*.
- Les lectures normales ont la sémantique *acquire*.

Pour une architecture plus relâchée (Power, ARM) :

- Des barrières mémoire moins coûteuses que les barrières qui garantissent SC.

Relaxed Separation Logic (Vafeiadis & Narayan, 2013)

Une logique de séparation concurrente pour une partie des *low-level atomics* de C/C++ 2011.

c	$::= a$	expression pure
	$\mid \texttt{let}\ x = c\ \texttt{in}\ c'$	séquencement et liaison
	$\mid \texttt{if}\ a\ \texttt{then}\ c_1\ \texttt{else}\ c_2$	conditionnelle
	$\mid \texttt{repeat}\ c$	répétition tant que $= 0$
	$\mid c_1 \parallel c_2$	exécution parallèle
	$\mid \texttt{alloc}()$	allocation
	$\mid \texttt{get}_X(a)$	lecture mémoire
	$\mid \texttt{set}_Y(a, a')$	écriture mémoire
	$\mid \texttt{CAS}_{Z,X}(a, a', a'')$	*Compare And Swap*
X	$::= sc \mid acq \mid rlx \mid na$	type de lecture
Y	$::= sc \mid rel \mid rlx \mid na$	type d'écriture
Z	$::= sc \mid rel\_acq \mid acq \mid rel$	type de CAS

Les assertions de la logique RSL

Assertions, préconditions :

$$P ::= \langle A \rangle \mid \texttt{true} \mid P \mathbin{\ast} P'$$

$\mid \ell \overset{\pi}{\mapsto} v$	la case ℓ contient v avec permission π
$\mid \mathit{Acq}(\ell, \Phi) \mid \mathit{Rel}(\ell, \Phi)$	invariants de ressources
$\mid \mathit{RMWAcq}(\ell, \Phi)$	invariant de ressource
$\mid \mathit{Init}(\ell)$	l'invariant a été établi
$\mid \mathit{Uninit}(\ell)$	la case ℓ vient d'être allouée

Postconditions, invariants de ressources :

$$Q, \Phi ::= \lambda v.\ P$$

Accès non atomiques

Les accès non atomiques sont comme en logique de séparation standard :

$$\begin{array}{rll} \{\, \mathtt{emp} \,\} & \mathtt{alloc}() & \{\, \lambda \ell.\ \mathit{Uninit}(\ell) \,\} \\ \{\, \ell \overset{\pi}{\mapsto} v \,\} & \mathtt{get}_{na}(\ell) & \{\, \lambda x.\ \langle x = v \rangle \mathbin{\ast} \ell \overset{\pi}{\mapsto} v \,\} \\ \{\, \mathit{Uninit}(\ell) \vee \ell \overset{1}{\mapsto} \_ \,\} & \mathtt{set}_{na}(\ell, v) & \{\, \lambda \_.\ \ell \overset{1}{\mapsto} v \,\} \end{array}$$

(Le rôle de *Uninit* est d'empêcher de lire depuis une case mémoire allouée mais non initialisée.)

Écritures en mode *release*, lectures en mode *acquire*

$\mathit{Rel}(\ell, \Phi)$ donne la permission d'écrire à l'adresse ℓ une valeur v satisfaisant $\Phi\ v$.

$\mathit{Acq}(\ell, \Phi)$, en conjonction avec $\mathit{Init}(\ell)$, donne la permission de lire l'adresse ℓ, obtenant une valeur v et la ressource $\Phi\ v$.

$$\begin{array}{rcl} \{\,\mathtt{emp}\,\} & \mathtt{alloc}() & \{\,\lambda \ell.\ \mathit{Rel}(\ell, \Phi) \mathbin{\ast} \mathit{Acq}(\ell, \Phi)\,\} \\ \{\,\mathit{Rel}(\ell, \Phi) \mathbin{\ast} \Phi\ v\,\} & \mathtt{set}_{\mathit{rel}}(\ell, v) & \{\,\mathit{Rel}(\ell, \Phi) \mathbin{\ast} \mathit{Init}(\ell)\,\} \\ \{\,\mathit{Acq}(\ell, \Phi) \mathbin{\ast} \mathit{Init}(\ell)\,\} & \mathtt{get}_{\mathit{acq}}(\ell) & \{\,\lambda v.\ \Phi\ v \mathbin{\ast} \mathit{Acq}(\ell, \Phi[v \leftarrow \mathtt{emp}])\,\} \end{array}$$

On peut lire plusieurs fois la même valeur, mais les lectures suivantes ne transfèrent pas de ressources :

$$\Phi[v \leftarrow \mathtt{emp}] \overset{\mathit{def}}{=} \lambda v'.\ \mathtt{if}\ v' = v\ \mathtt{then}\ \mathtt{emp}\ \mathtt{else}\ \Phi\ v'.$$

Exemple de transfert de ressources

On a un tampon b (non atomique) et un drapeau x (atomique).
On prend $\Phi = \lambda v.\ \texttt{if}\ v = 0\ \texttt{then}\ \texttt{emp}\ \texttt{else}\ b \overset{1}{\mapsto} 53$.

$$\texttt{let}\ x = \texttt{alloc}()\ \texttt{in}\ \texttt{let}\ b = \texttt{alloc}()\ \texttt{in}\ \texttt{set}_{rel}(x, 0);$$
$$\{\ \mathit{Uninit}(b) \ast \mathit{Rel}(x, \Phi) \ast \mathit{Init}(x) \ast \mathit{Acq}(x, \Phi)\ \}$$

$$\{\ \mathit{Uninit}(b) \ast \mathit{Rel}(x, \Phi)\ \}$$
$$\texttt{set}_{na}(b, 53);$$
$$\{\ b \overset{1}{\mapsto} 53 \ast \mathit{Rel}(x, \Phi)\ \}$$
$$\Rightarrow \{\ \Phi\ 1 \ast \mathit{Rel}(x, \Phi)\ \}$$
$$\texttt{set}_{rel}(x, 1)$$

$\Big\|$

$$\{\ \mathit{Init}(x) \ast \mathit{Acq}(x, \Phi)\ \}$$
$$\texttt{repeat}\ \texttt{get}_{acq}(x);$$
$$\{\ \exists v \neq 0,\ \Phi\ v\ \} \Rightarrow \{\ b \overset{1}{\mapsto} 53\ \}$$
$$\texttt{let}\ n = \texttt{get}_{na}(b)\ \texttt{in}$$
$$\{\ b \overset{1}{\mapsto} 53 \ast \langle n = 53 \rangle\ \}$$

Plusieurs lecteurs, plusieurs écrivains

Les permissions pour écrire sont duplicables :

$$\mathit{Init}(\ell) = \mathit{Init}(\ell) \mathbin{\ast} \mathit{Init}(\ell) \qquad \mathit{Rel}(\ell, \Phi) = \mathit{Rel}(\ell, \Phi) \mathbin{\ast} \mathit{Rel}(\ell, \Phi)$$

Les permissions pour lire sont sécables :

$$\mathit{Acq}(\ell, \lambda v.\ \Phi_1\ v \mathbin{\ast} \Phi_2\ v) = \mathit{Acq}(\ell, \Phi_1) \mathbin{\ast} \mathit{Acq}(\ell, \Phi_2)$$

Exemple : un écrivain, deux lecteurs.

$$\begin{array}{l} \mathtt{set}_{na}(a, 13); \\ \mathtt{set}_{na}(b, 17); \\ \mathtt{set}_{rel}(x, 1); \end{array} \;\Big\|\; \begin{array}{l} \mathtt{repeat}\ \mathtt{get}_{acq}(x); \\ \{\, a \overset{1}{\mapsto} 13 \,\} \end{array} \;\Big\|\; \begin{array}{l} \mathtt{repeat}\ \mathtt{get}_{acq}(x); \\ \{\, b \overset{1}{\mapsto} 17 \,\} \end{array}$$

Compare And Swap

$$\{\,\texttt{emp}\,\}\ \texttt{alloc}()\ \{\,\lambda \ell.\ \mathit{Rel}(\ell, \Phi) \mathbin{\ast} \mathit{RMWAcq}(\ell, \Phi)\,\}$$

Les permissions *RMWAcq* sont duplicables :

$$\mathit{RMWAcq}(\ell, \Phi) \mathbin{\ast} \mathit{RMWAcq}(\ell, \Phi) = \mathit{RMWAcq}(\ell, \Phi)$$

La règle pour $\mathit{CAS}_{X,rlx}$:

$$\frac{\begin{array}{c} P \Rightarrow \mathit{Init}(\ell) \mathbin{\ast} \mathit{RMWAcq}(\ell, \Phi) \mathbin{\ast} \texttt{true} \\ P \mathbin{\ast} \Phi\ v \Rightarrow \mathit{Rel}(\ell, \Psi) \mathbin{\ast} \Psi\ v' \mathbin{\ast} R\ 1 \\ P \Rightarrow R\ 0 \\ X \in \{\mathit{rel}, \mathit{rlx}\} \Rightarrow \Phi\ v = \texttt{emp} \qquad X \in \{\mathit{acq}, \mathit{rlx}\} \Rightarrow \Psi\ v' = \texttt{emp} \end{array}}{\{\,P\,\}\ \texttt{CAS}_{X,rlx}(\ell, v, v')\ \{\,R\,\}}$$

Une logique pour les accès relâchés

Écritures et lectures relâchées

Intuition : c'est comme une écriture *release* ou une lecture *acquire* mais sans transfert de ressources.

$$\{\, \mathit{Acq}(\ell, \Phi) \,\}\ \mathtt{get}_{\mathit{rlx}}(\ell)\ \{\, \lambda v.\ \langle \Phi\ v \neq \mathtt{false} \rangle \,\}$$

$$\frac{\Phi\ v = \mathtt{emp}\ \text{(c.à.d. } \Phi\ v \text{ est pure et vraie)}}{\{\, \mathit{Rel}(\ell, \Phi) \,\}\ \mathtt{set}_{\mathit{rlx}}(\ell, v)\ \{\, \mathit{Rel}(\ell, \Phi) \,\}}$$

Une modeste application : contrôler l'ensemble des valeurs possibles pour la case ℓ, p.ex. $\Phi = \lambda v.\ \langle 0 \leq v \leq 10 \rangle$.

Mode relâché et valeurs *ex nihilo*

Vafeiadis & Narayan observent que ces règles sont incorrectes pour C/C++ 2011, car les accès relâchés ont le droit de produire des valeurs *ex nihilo* (*out of thin air*).

$$\begin{array}{l} \texttt{let } a = \texttt{get}_{rlx}(X) \texttt{ in} \\ \texttt{set}_{rlx}(Y, a) \end{array} \quad \Big\| \quad \begin{array}{l} \texttt{let } b = \texttt{get}_{rlx}(Y) \texttt{ in} \\ \texttt{set}_{rlx}(X, b) \end{array}$$

En partant de $X = Y = 0$, on peut (d'après le standard) finir avec $X = Y = 1$.

D'après les règles de Vafeiadis & Narayan, $\Phi = \lambda v.\ \langle v = 0 \rangle$ est un invariant correct pour X et pour Y.

Le problème des valeurs *ex nihilo*

Un risque majeur : cassent la sûreté du typage et introduisent des failles de sécurité.

Un risque théorique : aucune architecture ou compilateur connu n'exhibe ce comportement.

Un problème de spécification : les définitions axiomatiques (par structures d'événements) des modèles mémoire comme C11 n'arrivent pas à distinguer entre

- comportements avec valeurs *ex nihilo* ;
- comportements spéculatifs (de type *load buffering*) corrects.

Une sémantique prometteuse

Kang et al (2017) décrivent une sémantique opérationnelle pour les atomiques de C/C++ 2011, de la forme (simplifiée) suivante :

1- La mémoire partagée M = un ensemble de messages d'écriture

Un message est de la forme $\langle \ell : v \,@\, t \rangle$, représentant l'écriture de la valeur v à l'adresse ℓ à la date t.
On a au plus un message $\langle \ell : v \,@\, t \rangle$ pour un ℓ et un t donné.

Une sémantique prometteuse

Kang et al (2017) décrivent une sémantique opérationnelle pour les atomiques de C/C++ 2011, de la forme (simplifiée) suivante :

1- La mémoire partagée M = un ensemble de messages d'écriture

2- Un processus = une vue V de la mémoire ...

Une vue = une fonction adresse $\to$ date à laquelle le contenu de cette adresse a été observé pour la dernière fois.

Lire l'adresse ℓ = observer un message $\langle \ell : v \,@\, t \rangle$ avec $t \geq V(\ell)$.

Écrire v en ℓ = envoyer un message $\langle \ell : v \,@\, t \rangle$ avec $t > V(\ell)$ frais.

Dans les deux cas, V est mis à jour : $V[\ell \leftarrow t]$.

Une sémantique prometteuse

Kang et al (2017) décrivent une sémantique opérationnelle pour les atomiques de C/C++ 2011, de la forme (simplifiée) suivante :

1- La mémoire partagée M = un ensemble de messages d'écriture

2- Un processus = une vue V de la mémoire ...

3- ... plus un ensemble P de promesses.

Une promesse = une écriture spéculative = un message déjà dans la mémoire partagée, mais qui devra être réalisé par une vraie écriture plus tard dans l'exécution du processus.

Une sémantique prometteuse

Kang et al (2017) décrivent une sémantique opérationnelle pour les atomiques de C/C++ 2011, de la forme (simplifiée) suivante :

1- La mémoire partagée M = un ensemble de messages d'écriture

2- Un processus = une vue V de la mémoire …

3- …plus un ensemble P de promesses.

Invariant maintenu à chaque étape de réduction : il est toujours possible de réduire pour réaliser toutes les promesses. Ceci empêche les comportements *ex nihilo*.

$$\begin{array}{ccccc} (c_1, V_1, P_1)/M_1 & \longrightarrow & (c_2, V_2, P_2)/M_2 & \longrightarrow & \cdots \\ \searrow^{*} & & \searrow^{*} & & \\ (\_, \_, \emptyset)/\_ & & (\_, \_, \emptyset)/\_ & & \end{array}$$

SLR : une logique de séparation pour la sémantique prometteuse

(Svendsen *et al*, 2018.)

Une extension de RSL avec de nouvelles assertions :

$O(\ell, v, t)$ (généralise $\mathit{Init}(\ell)$)

J'ai observé la valeur v à l'adresse ℓ à la date t.

$W^{\pi}(\ell, X)$ (généralise $\ell \overset{\pi}{\mapsto} v$)

Je détiens la permission π sur l'adresse ℓ.

$X = \{(v_1, t_1), \ldots, (v_n, t_n)\}$ est un ensemble d'écritures datées à cette adresse.

Si $\pi = 1$ (permission exclusive), X contient toutes les écritures effectuées en ℓ.

Si $\pi < 1$, X est un sous-ensemble de ces écritures.

Quelques propriétés de l'assertion W

L'assertion est sécable :

$$W^{\pi_1+\pi_2}(\ell, X_1 \cup X_2) = W^{\pi_1}(\ell, X_1) \mathbin{\ast} W^{\pi_2}(\ell, X_2)$$

Les écritures sont cohérentes (valeur unique à une date donnée) :

$$W^{\pi}(\ell, X) \mathbin{\ast} \langle (v, t) \in X \wedge (v', t') \in X \wedge v \neq v' \rangle \Rightarrow W^{\pi}(\ell, X) \mathbin{\ast} \langle t \neq t' \rangle$$

Toutes les écritures sont observées :

$$W^{\pi}(\ell, X) \mathbin{\ast} \langle (v, t) \in X \rangle \Rightarrow W^{\pi}(\ell, X) \mathbin{\ast} O(\ell, v, t)$$

La réciproque est vraie si la possession est exclusive :

$$W^{1}(\ell, X) \mathbin{\ast} O(\ell, v, t) \Rightarrow W^{1}(\ell, X) \mathbin{\ast} O(\ell, v, t) \mathbin{\ast} \langle (v, t) \in X \rangle$$

Raisonner sur les accès atomiques relâchés

$$\frac{\Phi\ v = \mathtt{emp}\ \text{(c.à.d. } \Phi\ v \text{ est pure et vraie)}}{\left\{\begin{array}{l} W^\pi(\ell, X) \\ \ast\ \mathit{Rel}(\ell, \Phi) \\ \ast\ O(\ell, \_, t) \end{array}\right\}\ \mathtt{set}_{rlx}(\ell, v)\ \left\{\begin{array}{l} \lambda\_.\ \exists t' > t, \\ \qquad W^\pi(\ell, \{(v, t')\} \cup X) \end{array}\right\}}$$

Comme dans RSL, une écriture relâchée ne transfère pas de ressources ($\Phi\ v = \mathtt{emp}$).

L'écriture est reflétée dans l'assertion $W^\pi(\ell, X)$, qui n'a pas besoin d'être exclusive. ($\pi < 1$ est possible !)

$O(\ell, \_, t)$ montre que ℓ est initialisée et donne une borne inférieure pour la nouvelle date t'.

Raisonner sur les accès atomiques relâchés

$$\left\{ \begin{array}{l} \mathit{Acq}(\ell, \Phi) \\ \ast\ \mathit{O}(\ell, \_, t) \end{array} \right\} \mathtt{get}_{\mathit{rlx}}(\ell) \left\{ \begin{array}{l} \lambda v.\ \exists t' \geq t, \\ \qquad \mathit{Acq}(\ell, \Phi) \ast \mathit{O}(\ell, v, t') \ast \nabla(\Phi\ v) \end{array} \right\}$$

Une lecture non-atomique de la valeur v donne accès à la partie pure $\nabla(\Phi\ v)$ de l'invariant de ressources $\Phi\ v$, ainsi qu'à une nouvelle observation $\mathit{O}(\ell, v, t')$.

Si on détient la permission complète sur ℓ, la valeur lue est déterminée par la plus récente des écritures.

$$\left\{ \begin{array}{l} \mathit{Acq}(\ell, \Phi) \\ \ast\ \mathit{W}^1(\ell, X) \end{array} \right\} \mathtt{get}_{\mathit{rlx}}(\ell) \left\{ \begin{array}{l} \lambda v.\ \exists t, \langle (v, t) = \max(X) \rangle \\ \qquad \ast\ \mathit{Acq}(\ell, \Phi) \ast \mathit{W}^1(\ell, X) \ast \nabla(\Phi\ v) \end{array} \right\}$$

Point d’étape

Point d'étape

Un constat : les modèles mémoires comme ceux de Java ou de C/C++ 2011 sont compliqués et pas encore parfaitement compris.

> *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies*
>
> (C. A. R. Hoare)

Point d'étape

Un constat : les modèles mémoires comme ceux de Java ou de C/C++ 2011 sont compliqués et pas encore parfaitement compris.

Un espoir : cela ne concerne qu'une poignée de bibliothèques ; le gros des codes parallèles est toujours écrit avec des synchronisations classiques.

Point d'étape

Un constat : les modèles mémoires comme ceux de Java ou de C/C++ 2011 sont compliqués et pas encore parfaitement compris.

Un espoir : cela ne concerne qu'une poignée de bibliothèques ; le gros des codes parallèles est toujours écrit avec des synchronisations classiques.

Un outil fort nécessaire : les logiques de programmes !

- Pour abstraire une partie de la complexité du modèle mémoire (cf. RSL, SLR).
- Pour combiner raisonnement en logique de séparation classique et raisonnement spécifique au modèle mémoire.

Bibliographie

Bibliographie

Une introduction aux modèles mémoires faiblement cohérents :

- S. V. Adve, H. J. Boehm, *Memory models : a case for rethinking parallel languages and hardware*, Comm. ACM, 2010.

Les logiques RSL et SLR :

- V. Vafeiadis, C. Narayan, *Relaxed separation logic : a program logic for C11 concurrency*, OOPSLA 2013.
- K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, V. Vafeiadis, *A Separation Logic for a Promising Semantics*, ESOP 2018.

La sémantique ≪prometteuse≫ pour C/C++ 2011 :

- J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, D. Dreyer, *A promising semantics for relaxed-memory concurrency*, POPL 2017.