Logiques de programmes, troisième cours

Pointeurs et structures de données : la logique de séparation

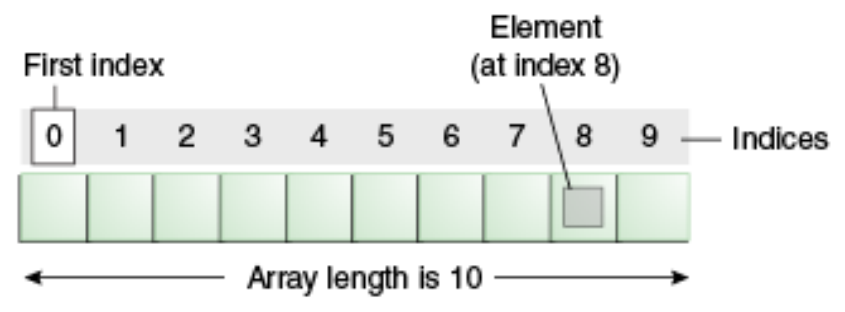
Xavier Leroy

2021-03-18

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

Prologue :
les tableaux en logique de Hoare

Les tableaux



Ajout des tableaux dans IMP

Expressions : $a ::= \ldots \mid T[a]$ lecture dans le tableau T

Commandes : $c ::= \ldots \mid T[a] := a'$ écriture dans le tableau T

Convention : les variables en majuscules T, U, sont des tableaux.

Une logique de Hoare pour les tableaux

Quel triplet de Hoare pour l'écriture dans un tableau ?

Incorrect : $\{ Q[T[a] \leftarrow a'] \}\ T[a] := a' \ \{ Q \}$ ✘

Il n'y a pas que $T[a]$ qui est modifié, mais aussi $T[a_1]$ pour toute expression a_1 qui a la même valeur que a. Exemple :

$$\{ 0 = 0 \wedge T[i] = 1 \}\ T[0] := 0 \ \{ T[0] = 0 \wedge T[i] = 1 \}$$

est faux si $i = 0$.

Correct : $\{ Q[T \leftarrow (T + a \mapsto a')] \}\ T[a] := a' \ \{ Q \}$ ✔

L'expression $T + a \mapsto a'$ dénote un tableau égal à T sauf que l'indice a a pour valeur a'. (*functional update*)

Raisonner sur les tableaux

On raisonne sur ces tableaux «fonctionnels» avec l'équation

$$(T + a \mapsto a')\,[i] = \begin{cases} a' & \text{si } i = a \\ T[i] & \text{si } i \neq a \end{cases}$$

Exemple (vérification d'une écriture dans `T[0]`)

$$\{\, i \neq 0 \wedge \mathtt{T}[i] = 1 \,\} \iff$$
$$\{\, 0 = 0 \wedge (i = 0 \;?\; 0 : \mathtt{T}[i]) = 1 \,\} \iff$$
$$\{\, (\mathtt{T} + 0 \mapsto 0)[0] = 0 \wedge (\mathtt{T} + 0 \mapsto 0)[i] = 1 \,\}$$
$$\mathtt{T}[0] := 0$$
$$\{\, \mathtt{T}[0] = 0 \wedge \mathtt{T}[i] = 1 \,\}$$

Exemple : l'initialisation d'un tableau

$i := 0;$

$\{\, i = 0 \,\}$

`while` $i < N$ `do`

$\{\, \forall j,\ 0 \leq j < i \Rightarrow \mathtt{T}[j] = j \times 2 \,\} \Rightarrow$

$\{\, \forall j,\ 0 \leq j < i + 1 \Rightarrow (\mathtt{T} + i \mapsto i \times 2)[j] = j \times 2 \,\}$

$\mathtt{T}[i] := i \times 2;$

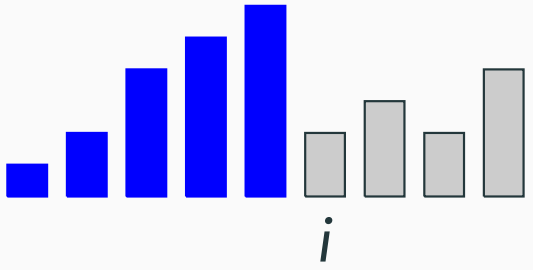
$\{\, \forall j,\ 0 \leq j < i + 1 \Rightarrow \mathtt{T}[j] = j \times 2 \,\}$

$i := i + 1$

$\{\, \forall j,\ 0 \leq j < i \Rightarrow \mathtt{T}[j] = j \times 2 \,\}$

`done`

Exemple : le tri par insertion



$i := 1;$

`while` $i < N$ `do`

$\{\, 0 < i < N \;\wedge\; \forall p, q,\ 0 \le p \le q < i \Rightarrow \mathtt{T}[p] \le \mathtt{T}[q] \,\}$

$j := i;$

`while` $j > 0 \wedge T[j-1] > T[j]$

$\{\, 0 \le j \le i \;\wedge\; \forall p, q,\ 0 \le p \le q \le i \wedge q \neq j \Rightarrow \mathtt{T}[p] \le \mathtt{T}[q] \,\}$

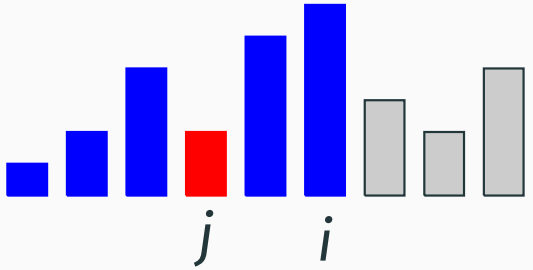
$\mathtt{swap}(T, j, j-1);$

$j := j - 1$

`done`

$i := i + 1$

`done`



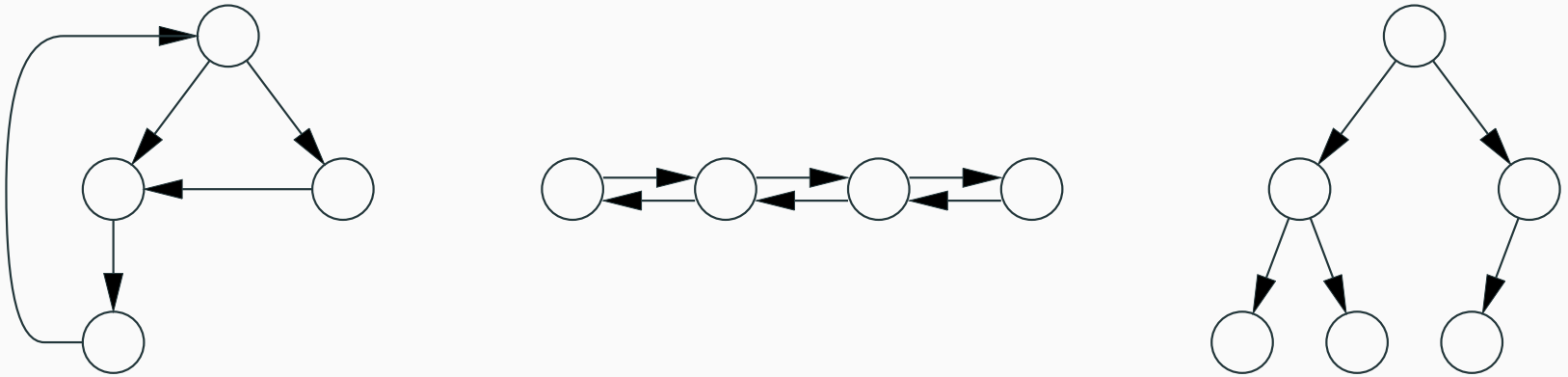
Plus un invariant : $\mathtt{T}$ est une permutation du tableau initial $\mathtt{T}_0$

Les pointeurs
et l'approche de Burstall-Bornat

Les pointeurs

Pointeurs : explicites (Algol-W, Pascal, C, C++) ou implicites via des objets manipulés par référence (Java, Lisp, Python, OCaml, …).

Très utilisés pour représenter et manipuler des graphes et des structures de données chaînées (listes, arbres, …).



Exemple : les listes simplement chaînées

```
class List {
  int head;
  List tail;
}
```

```
typedef struct cell * list;
struct cell { int head; list tail; };
```

Les listes $[1; 2; 3]$ et $[4; 5]$:

Concaténation en place des listes `l1` et `l2` :

```
p = l1;
while (p->tail != NULL) p = p->tail;
p->tail = l2;
```

Exemple : les listes simplement chaînées

```
class List {
  int head;
  List tail;
}
```

```
typedef struct cell * list;
struct cell { int head; list tail; };
```

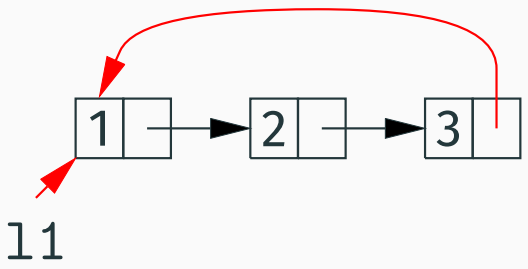
Les listes $[1; 2; 3]$ et $[4; 5]$:

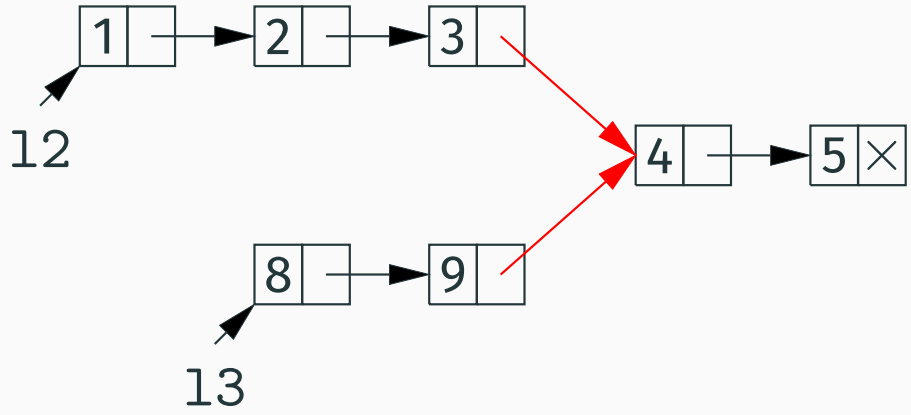
Concaténation en place des listes `l1` et `l2` :

```
p = l1;
while (p->tail != NULL) p = p->tail;
p->tail = l2;
```

Le problème du partage (*aliasing*)

Le partage est essentiel pour représenter les graphes,
mais souvent problématique pour des structures plus simples.





À gauche : `l1` est une liste cyclique (infinie) $1, 2, 3, 1, 2, 3, \ldots$

À droite : `l2` partage avec `l3` le suffixe $\ll 4, 5 \gg$.

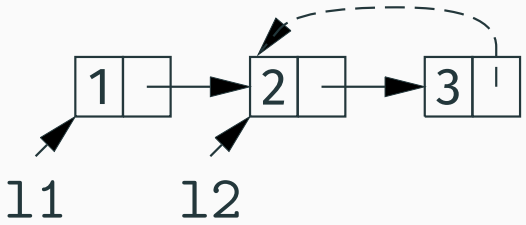
Le problème du partage (*aliasing*)

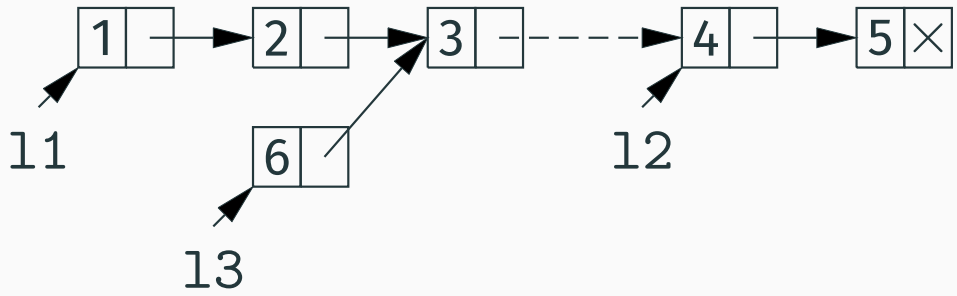
```
p = l1;
while (p->tail != NULL) p = p->tail;
p->tail = l2;
```

Si `l1` est cyclique, la concaténation ne termine pas.

Si `l1` partage avec `l2`, une liste cyclique est créée.

Toute liste `l3` qui partage avec `l1` est modifiée par effet de bord.





Modéliser les pointeurs et le tas mémoire (*memory heap*)

Le modèle naïf :

- Le tas mémoire = un grand tableau global M.
- Un pointeur p = un indice dans M.
- Un accès $p \to \texttt{f}$ = un accès $M[p + \texttt{offset(f)}]$.

Le modèle de Burstall-Bornat :

- Le tas mémoire = un tableau global par champ F_1, F_2, ...
- Un pointeur p = un indice dans les tableaux F_i.
- Un accès $p \to \texttt{f}$ = un accès $F[p]$.
- Une affectation $p \to \texttt{f} := a$ modifie $F[p]$ mais les autres tableaux $F' \neq F$ sont inchangés.

Les graphes dans le modèle Burstall-Bornat

```
struct node {
    bool mark;
    int arity;
    struct node * child[arity];
}
```

Trois tableaux globaux $\texttt{MARK}[p]$, $\texttt{ARITY}[p]$, $\texttt{CHILD}[p][i]$.

La relation d'accessibilité $path(p, q)$,
«le nœud q est accessible depuis le nœud p».

$$path(p, p) \qquad \frac{p \neq 0 \qquad 0 \leq i < \texttt{ARITY}[p] \qquad path(\texttt{CHILD}[p][i], q)}{path(p, q)}$$

Un algorithme générique de parcours de graphe

Marquer tous les nœuds atteignables depuis une racine r.

$$\{\, \forall x,\ \texttt{MARK}[x] = 0 \,\}$$

$W := \{r\}$

`while` $W \neq \emptyset$ `do`

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

pick $p \in W$; $\quad W := W \setminus \{p\}$;

`if` $\texttt{MARK}[p] = 0$ `then begin`

$\texttt{MARK}[p] = 1$;

$W := W \cup \{\texttt{CHILD}[p][i] \mid 0 \leq i < \texttt{ARITY}[p]\}$

`end`

`done`

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

(Note : l'affectation $\texttt{MARK}[p] = 1$ ne change pas les tableaux `CHILD` et `ARITY`, et donc préserve la relation *path*.)

Les listes simplement chaînées dans le modèle Burstall-Bornat

Deux tableaux globaux $\texttt{HEAD}$ et $\texttt{TAIL}$.

Un prédicat de représentation : $\mathit{lseg}(w, p, q)$,
« entre les pointeurs p et q se trouve la représentation de la liste (mathématique) w ».

$$\mathit{lseg}(\varepsilon, p, p) \qquad \frac{p \neq 0 \qquad \texttt{HEAD}[p] = n \qquad \mathit{lseg}(w, \texttt{TAIL}[p], q)}{\mathit{lseg}(n \cdot w, p, q)}$$

p pointe sur une liste bien formée (sans cycles) $=$

$$\exists w,\ \mathit{lseg}(w, p, \texttt{NULL}).$$

p et q pointent sur des listes disjointes (sans partage) =

$$\forall r, w, w',\ \mathit{lseg}(w, p, r) \wedge \mathit{lseg}(w', q, r) \Rightarrow r = \texttt{NULL}$$

Spécifier la concaténation de listes

On définit $\mathit{list}(w, p) \stackrel{def}{=} \mathit{lseg}(w, p, \texttt{NULL})$,

≪le pointeur p représente la liste w≫.

$$\{\, w \neq \varepsilon \wedge \mathit{list}(w, \texttt{l1}) \wedge \mathit{list}(w', \texttt{l2}) \wedge \mathit{disjoint}(\texttt{l1}, \texttt{l2}) \,\}$$
$$\texttt{concat}(\texttt{l1}, \texttt{l2})$$
$$\{\, \mathit{list}(w \cdot w', \texttt{l1}) \wedge \mathit{list}(w', \texttt{l2}) \,\}$$

Une spécification raisonnable mais encore incomplète :
il manque le fait que toute liste `l3` initialement disjointe de `l1` n'est pas modifiée.

> *The verification proofs are quite long for such a simple matter and very boring. We will not weary the reader with them; instead we will try to do better.*
>
> *(R. M. Burstall, 1972)*

Raisonnement local et empreintes mémoire

Le raisonnement local

Un principe de bon sens :

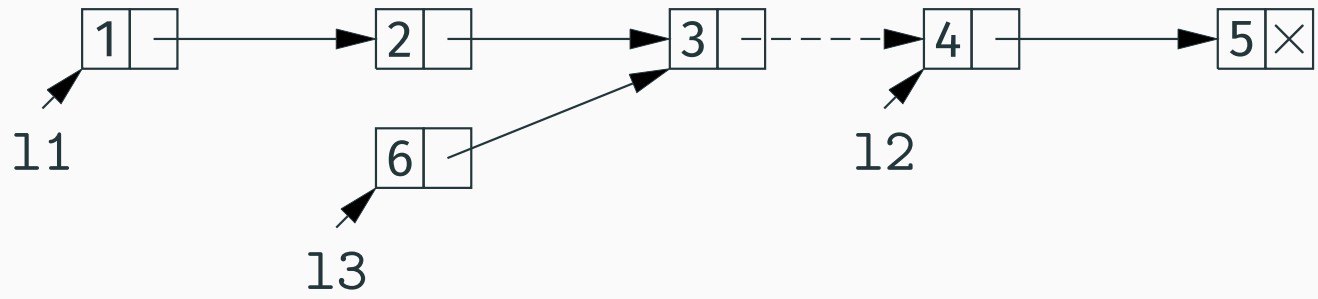
> *Tout ce qui n'est pas explicitement mentionné dans* $\{ P \}\ c\ \{ Q \}$ *est préservé par l'exécution de* c.

En logique de Hoare, ce principe se présente comme la règle d'encadrement (*frame rule*) suivante :

$$\frac{\{ P \}\ c\ \{ Q \} \qquad \text{aucune variable modifiée par } c \text{ n'apparaît dans } R}{\{ P \wedge R \}\ c\ \{ Q \wedge R \}}$$

Exemple : $\{ x = 0 \}\ x := x + 1\ \{ x = 1 \}$, et donc
$\{ x = 0 \wedge y = 8 \}\ x := x + 1\ \{ x = 1 \wedge y = 8 \}$.

Pointeur + partage = plus de raisonnement local ?



Prenons

$$P = \mathit{list}(1.2.3.\varepsilon, \mathtt{l1}) \wedge \mathit{list}(4.5.\varepsilon, \mathtt{l2}) \wedge \mathit{disjoint}(\mathtt{l1}, \mathtt{l2})$$

$$Q = \mathit{list}(1.2.3.4.5.\varepsilon, \mathtt{l1})$$

$$R = \mathit{list}(6.3.\varepsilon, \mathtt{l3})$$

On a bien $\{\, P \,\}\ \mathtt{concat}(\mathtt{l1}, \mathtt{l2})\ \{\, Q \,\}$

mais pas $\{\, P \wedge R \,\}\ \mathtt{concat}(\mathtt{l1}, \mathtt{l2})\ \{\, Q \wedge R \,\}$

(R est faux «après»).

Vers une meilleure règle d'encadrement

$$\frac{\{\,P\,\}\ c\ \{\,Q\,\} \qquad \text{aucune variable modifiée par } c \text{ n'apparaît dans } R \qquad \text{aucune case mémoire modifiée par } c \text{ n'est mentionnée dans } R}{\{\,P \wedge R\,\}\ c\ \{\,Q \wedge R\,\}}$$

Cette règle est plausible, mais la condition ≪aucune case mémoire modifiée par c n'est mentionnée dans R≫ n'est pas syntaxique.

Ce serait bien si la logique de programmes nous permettait de la vérifier !

Empreintes mémoire

À une assertion logique P, Q on associe une empreinte mémoire : l'ensemble des adresses (pointeurs) dont elle décrit le contenu.

Exemple

L'assertion $p \mapsto 0$, «à l'adresse p il y a la valeur 0»,
a pour empreinte $\{p\}$.

L'assertion $(b \wedge p \mapsto 0) \vee (\neg b \wedge q \mapsto 1)$
a pour empreinte $\{p\}$ si b est vrai, $\{q\}$ si b est faux.

Vers une meilleure règle d'encadrement

Intuition (presque vraie) : si $\{ P \}\ c\ \{ Q \}$, les cases mémoire modifiées pendant l'exécution de c sont mentionnées dans P ou dans Q, et donc appartiennent à leur empreinte.

$$\frac{\begin{array}{c}\{ P \}\ c\ \{ Q \} \\ \text{aucune variable modifiée par } c \text{ n'apparaît dans } R \\ \mathit{empreinte}(P) \cap \mathit{empreinte}(R) = \emptyset \\ \mathit{empreinte}(Q) \cap \mathit{empreinte}(R) = \emptyset\end{array}}{\{ P \wedge R \}\ c\ \{ Q \wedge R \}}$$

Conjonction séparante

L'énoncé ≪P et R sont vraies et leurs empreintes sont disjointes≫ revient si souvent qu'on lui donne un nom : la conjonction séparante, notée $P \ast R$.

Formellement : (assertions = prédicats sur l'état mémoire h)

$$(P \ast R)\ h \quad \stackrel{def}{=} \quad \exists h_1, h_2,\ P\ h_1 \wedge R\ h_2 \wedge h = h_1 \uplus h_2 \text{ (union disjointe)}$$

Exemple

$p \mapsto 0 \ast p \mapsto 0$	est toujours fausse.
$p \mapsto 0 \ast q \mapsto 0$	implique $p \neq q$.

Conjonction séparante et encadrement

La *frame rule* de la logique de séparation :

$$\frac{\{\, P \,\}\ c\ \{\, Q \,\} \qquad \text{aucune variable modifiée par } c \text{ n'apparaît dans } R}{\{\, P \mathbin{\color{red}{\ast}} R \,\}\ c\ \{\, Q \mathbin{\color{red}{\ast}} R \,\}}$$

Capture élégamment l'idée de raisonnement local :
P, Q décrivent les parties de la mémoire pertinentes pour l'exécution de c ; R décrit d'autres parties.

La logique de séparation

L'émergence de la logique de séparation

Burstall (1972) : les *Distinct Nonrepeating List Systems*
($\approx$ structures simplement chaînées sans aucun partage)
+ règles de raisonnement ad-hoc.

Reynolds (1999), *Intuitionistic Reasoning about Shared Mutable Data Structures*. Introduit la notion de conjonction séparante.

O'Hearn et Pym (1999), *The Logic of Bunched Implications*.
Pour raisonner sur des ressources utilisées de manière linéaire.

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

Reynolds (2002), *Separation Logic : A Logic for Shared Mutable Data Structures*. L'article qui lui a donné son nom.

Quel langage avec des pointeurs ?

Approche classique : IMP + opérations `alloc`, `get`, `set`, `free`.

Deux degrés de mutabilité : variables et cases mémoire.

Assertions = prédicats de l'état des variables (s, le *store*)
et de l'état des cases mémoire (h, le *heap*)

Dans ce cours : mini-ML + références
ou encore : lambda-calcul + monade d'état.

Variables immuables contenant des références (adresses) de cases mémoires mutables.

Assertions = prédicats de l'état des cases mémoire (h, le *heap*).

Le langage PTR

Commandes (expressions avec effets) :

$c ::= a$	expression pure
$\mid \texttt{let}\ x = c\ \texttt{in}\ c'$	séquence et liaison
$\mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2$	conditionnelle
$\mid \texttt{choose}(N)$	choix non déterministe
$\mid \texttt{alloc}(N)$	alloue N cases mémoire
$\mid \texttt{get}(a)$	lecture à l'adresse a
$\mid \texttt{set}(a, a')$	écriture à l'adresse a
$\mid \texttt{free}(a)$	désallocation de l'adresse a

Dans les exemples on utilisera aussi des fonctions récursives $\texttt{def}\ f\ x_1\ \cdots\ x_n = c$.

Exemples de programmes PTR

Allocation et initialisation d'une cellule de liste :

```
def cons hd tl =
    let a = alloc(2) in
    let _ = set(a, hd) in
    let _ = set(a + 1, tl) in a
```

N.B. adresses = entiers, avec arithmétique de pointeurs.

Concaténation en place de deux listes :

```
def concat_rec l1 l2 =
    let tl = get(l1 + 1) in
    if tl = 0 then set(l1 + 1, l2) else concat_rec tl l2
def concat l1 l2 =
    if l1 = 0 then l2 else let _ = concat_rec l1 l2 in l1
```

Les assertions

Les assertions sont des prédicats sur l'état mémoire h.

Assertions pures (P est une proposition) :

$$\langle P \rangle \stackrel{def}{=} \lambda h.\, P \wedge Dom(h) = \emptyset$$

≪La mémoire est vide≫

$$emp \stackrel{def}{=} \langle \top \rangle = \lambda h.\, Dom(h) = \emptyset$$

≪L'adresse ℓ contient la valeur v≫

$$\ell \mapsto v \stackrel{def}{=} \lambda h.\, Dom(h) = \{\ell\} \wedge h(\ell) = v$$

≪L'adresse ℓ est valide≫

$$\ell \mapsto \_ \stackrel{def}{=} \exists v,\, \ell \mapsto v = \lambda h.\, Dom(h) = \{\ell\}$$

La conjonction séparante

La conjonction séparante $P \mathbin{\ast} Q$ dit qu'on peut partitionner l'état en deux parties, l'une satisfaisant P et l'autre satisfaisant Q.

$$P \mathbin{\ast} Q \stackrel{def}{=} \lambda h.\, \exists h_1, h_2,\ P\, h_1 \wedge Q\, h_2 \wedge h = h_1 \uplus h_2$$

Quelques propriétés :

$$P \mathbin{\ast} Q = Q \mathbin{\ast} P \qquad (P \mathbin{\ast} Q) \mathbin{\ast} R = P \mathbin{\ast} (Q \mathbin{\ast} R)$$

$$\mathrm{emp} \mathbin{\ast} P = P \mathbin{\ast} \mathrm{emp} = P \qquad \langle A \rangle \mathbin{\ast} \langle B \rangle = \langle A \wedge B \rangle$$

Les règles de la logique de séparation

Définissent des triplets $\{\, P \,\}\ c\ \{\, Q \,\}$.

Les commandes renvoyant une valeur v, la postcondition Q est une fonction $\lambda v \ldots$ des valeurs dans les assertions.

$$\frac{P \Rightarrow Q\ [\![a]\!]}{\{\, P \,\}\ a\ \{\, Q \,\}} \qquad \frac{\forall n \in [0, N[,\ P \Rightarrow Q\ n}{\{\, P \,\}\ \texttt{choose}(N)\ \{\, Q \,\}}$$

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

$$\frac{\{\, \langle b \rangle \ast P \,\}\ c_1\ \{\, Q \,\} \quad \{\, \langle \neg b \rangle \ast P \,\}\ c_2\ \{\, Q \,\}}{\{\, P \,\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{\, Q \,\}}$$

Les règles structurelles

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

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

$$\frac{A \Rightarrow \{\, P \,\}\ c\ \{\, Q \,\}}{\{\, \langle A \rangle \mathbin{\ast} P \,\}\ c\ \{\, Q \,\}}\ \text{(pure-elim)} \qquad \frac{\forall x, \{\, P \,\}\ c\ \{\, Q \,\}}{\{\, \exists x,\ P \,\}\ c\ \{\, Q \,\}}\ \text{(}\exists\text{-elim)}$$

Les ≪petites≫ règles pour les opérations sur la mémoire

≪Petites≫ signifie ≪avec la plus petite empreinte mémoire≫.

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

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

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

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

Des règles ≪plus grosses≫ s'obtiennent par encadrement, p.ex.

$$\{\,P\,\}\ \mathtt{alloc}(2)\ \{\,\lambda\ell.\,P\ast\ell\mapsto \_\ast\ell+1\mapsto \_\,\}$$

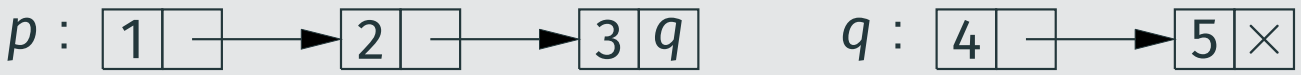
Structures de données et prédicats de représentation

Listes simplement chaînées

Prédicats de représentation :

$$\begin{aligned} lseg(\varepsilon, p, q) &= \langle p = q \rangle \\ lseg(x \cdot w, p, q) &= \exists p', p \mapsto x \ast p + 1 \mapsto p' \ast lseg(w, p', q) \\ list(w, p) &= lseg(w, p, \mathtt{NULL}) \end{aligned}$$

Exemple



On a $lseg(1.2.3.\varepsilon, p, q) \ast list(4.5.\varepsilon, q)$ ce qui équivaut à $list(1.2.3.4.5.\varepsilon, p)$.

Spécifier des fonctions sur les listes simplement chaînées

$$\{\, \mathit{list}(w, p) \,\} \quad \texttt{length}(p) \quad \{\, \lambda r.\, \langle r = |w| \rangle \mathbin{\ast} \mathit{list}(w, p) \,\}$$

$$\{\, \mathit{list}(w, p) \,\} \quad \texttt{copy}(p) \quad \{\, \lambda r.\, \mathit{list}(w, r) \mathbin{\ast} \mathit{list}(w, p) \,\}$$

$$\{\, \mathit{list}(w, p) \,\} \quad \texttt{dispose}(p) \quad \{\, \lambda r.\, \mathit{emp} \,\}$$

$$\{\, \mathit{list}(w, p) \mathbin{\ast} \mathit{list}(w', q) \,\} \quad \texttt{concat}(p, q) \quad \{\, \lambda r.\, \mathit{list}(w \cdot w', r) \,\}$$

$$\{\, \mathit{list}(w, p) \,\} \quad \texttt{reverse}(p) \quad \{\, \lambda r.\, \mathit{list}(\mathit{rev}(w), r) \,\}$$

Spécifier des fonctions sur les listes simplement chaînées

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{length}(p)\ \ \{\, \lambda r.\, \langle r = |w| \rangle \mathbin{\ast} \mathit{list}(w,p) \,\}$$

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{copy}(p)\ \ \{\, \lambda r.\, \mathit{list}(w,r) \mathbin{\ast} \mathit{list}(w,p) \,\}$$

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{dispose}(p)\ \ \{\, \lambda r.\, \mathit{emp} \,\}$$

$$\{\, \mathit{list}(w,p) \mathbin{\ast} \mathit{list}(w',q) \,\}\ \ \texttt{concat}(p,q)\ \ \{\, \lambda r.\, \mathit{list}(w \cdot w', r) \,\}$$

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{reverse}(p)\ \ \{\, \lambda r.\, \mathit{list}(\mathit{rev}(w), r) \,\}$$

Contrôle du partage :

- Pour $\texttt{copy}(p)$, la postcondition $\mathit{list}(w,r) \mathbin{\ast} \mathit{list}(w,p)$ garantit que le résultat ne partage aucune cellule avec l'argument.
- Pour $\texttt{concat}(p,q)$, la précondition $\mathit{list}(w,p) \mathbin{\ast} \mathit{list}(w',q)$ exige que les deux arguments ne partagent aucune cellule.

Spécifier des fonctions sur les listes simplement chaînées

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{length}(p)\ \ \{\, \lambda r.\ \langle r = |w| \rangle \mathbin{\ast} \mathit{list}(w,p) \,\}$$

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{copy}(p)\ \ \{\, \lambda r.\ \mathit{list}(w,r) \mathbin{\ast} \mathit{list}(w,p) \,\}$$

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{dispose}(p)\ \ \{\, \lambda r.\ \mathit{emp} \,\}$$

$$\{\, \mathit{list}(w,p) \mathbin{\ast} \mathit{list}(w',q) \,\}\ \ \texttt{concat}(p,q)\ \ \{\, \lambda r.\ \mathit{list}(w \cdot w', r) \,\}$$

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{reverse}(p)\ \ \{\, \lambda r.\ \mathit{list}(\mathit{rev}(w), r) \,\}$$

Gestion des ressources :

- Certaines listes sont préservées (`length`, `copy`)
- D'autres sont allouées (`copy`) ou détruites (`dispose`)
- D'autres sont recomposées en de nouvelles listes (`concat`)

Spécifier des fonctions sur les listes simplement chaînées

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{length}(p)\ \ \{\, \lambda r.\ \langle r = |w| \rangle \mathbin{\ast} \mathit{list}(w,p) \,\}$$

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{copy}(p)\ \ \{\, \lambda r.\ \mathit{list}(w,r) \mathbin{\ast} \mathit{list}(w,p) \,\}$$

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{dispose}(p)\ \ \{\, \lambda r.\ \mathit{emp} \,\}$$

$$\{\, \mathit{list}(w,p) \mathbin{\ast} \mathit{list}(w',q) \,\}\ \ \texttt{concat}(p,q)\ \ \{\, \lambda r.\ \mathit{list}(w \cdot w', r) \,\}$$

$$\{\, \mathit{list}(w,p) \,\}\ \ \texttt{reverse}(p)\ \ \{\, \lambda r.\ \mathit{list}(\mathit{rev}(w), r) \,\}$$

Permissions :

- Après $\texttt{dispose}(p)$ ou $\texttt{concat}(p,q)$, on perd le droit d'accéder à p et q comme listes bien formées.
- Après $\texttt{concat}(p,q)$, on gagne le droit d'accéder à la valeur résultat comme une liste bien formée.

Un exemple de vérification

$\{\, \mathit{list}(w, p) \mathbin{\ast} \mathit{list}(w', q) \,\}$

```
def rev_append p q =
  if p = NULL then                    // w = ε
    q
  else                                // w est de la forme x · w1
    let t = get(p + 1) in
    let _ = set(p + 1, q) in
    rev_append t p
```

$\{\, \lambda r.\ \mathit{list}(\mathit{rev}(w) \cdot w', r) \,\}$

Un exemple de vérification

$\{\, list(w, p) \mathbin{\ast} list(w', q) \,\}$

```
def rev_append p q =
  if p = NULL then                    // w = ε
```

$\{\, \langle p = \texttt{NULL} \rangle \mathbin{\ast} list(w', q) \,\}$

```
    q
  else                                // w est de la forme x · w1
```

$\{\, \exists p', p \mapsto x \mathbin{\ast} p + 1 \mapsto p' \mathbin{\ast} list(w_1, p') \mathbin{\ast} list(w', q) \,\}$

```
    let t = get(p + 1) in
```

$\{\, p \mapsto x \mathbin{\ast} p + 1 \mapsto t \mathbin{\ast} list(w_1, t) \mathbin{\ast} list(w', q) \,\}$

```
    let _ = set(p + 1, q) in
```

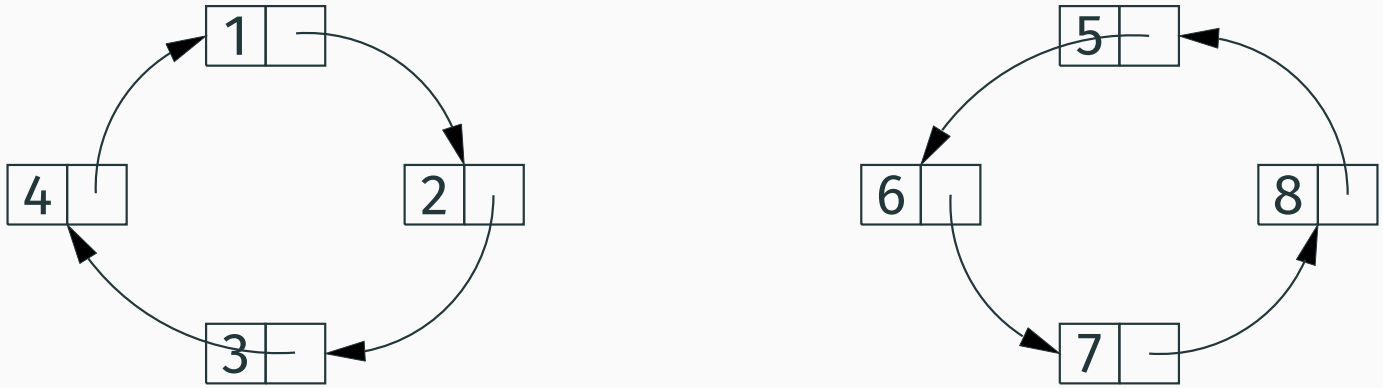
$\{\, list(w_1, t) \mathbin{\ast} p \mapsto x \mathbin{\ast} p + 1 \mapsto q \mathbin{\ast} list(w', q) \,\}$

```
    rev_append t p
```

$\{\, \lambda r.\ list(rev(w_1).x \cdot w', r) \,\}$

$\{\, \lambda r.\ list(rev(w) \cdot w', r) \,\}$

Listes circulaires



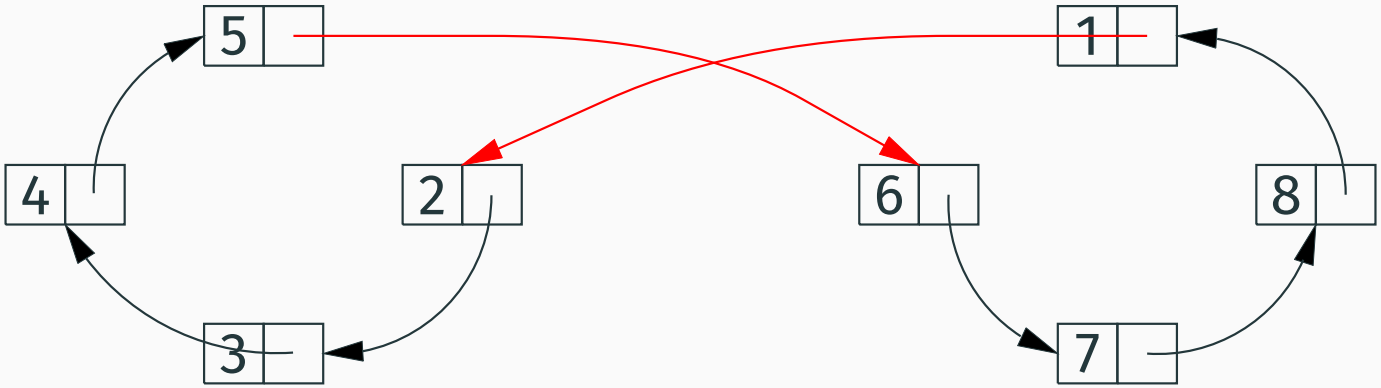
Prédicat de représentation :

$$circlist(w, p) = \langle w \neq \varepsilon \rangle \mathbin{\ast} lseg(w, p, p)$$

Concaténation de deux listes circulaires :

$$\{\, circlist(w, p) \mathbin{\ast} circlist(w', q) \,\}$$
$$\texttt{swap}(p, q); \texttt{swap}(p + 1, q + 1)$$
$$\{\, circlist(w \cdot w', q) \,\}$$

Listes circulaires



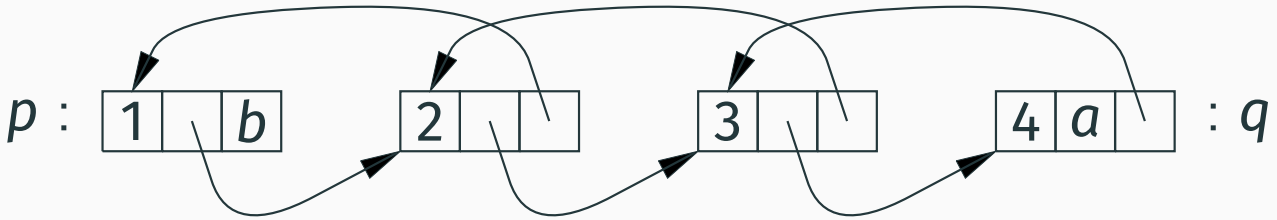
Prédicat de représentation :

$$circlist(w, p) = \langle w \neq \varepsilon \rangle \mathbin{\ast} lseg(w, p, p)$$

Concaténation de deux listes circulaires :

$$\{\, circlist(w, p) \mathbin{\ast} circlist(w', q) \,\}$$
$$\texttt{swap}(p, q); \texttt{swap}(p + 1, q + 1)$$
$$\{\, circlist(w \cdot w', q) \,\}$$

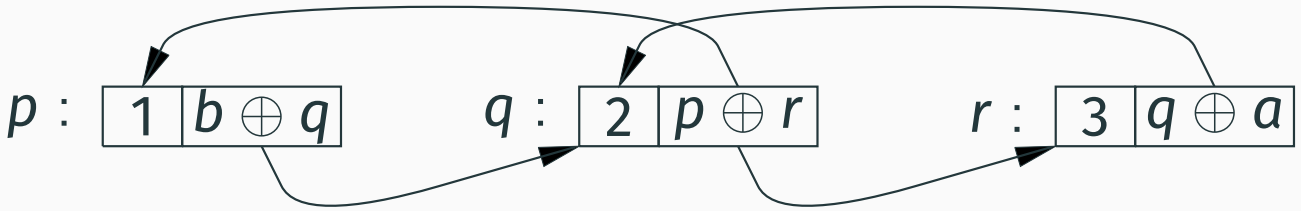
Listes doublement chaînées



Chaînage en avant de p jusqu'à a, en arrière de q jusqu'à b.

$$
\begin{aligned}
dlseg(\varepsilon, p, a, q, b) &= \langle p = a \wedge q = b \rangle \\
dlseg(x \cdot w, p, a, q, b) &= \exists p', p \mapsto x \mathbin{\ast} p + 1 \mapsto p' \mathbin{\ast} p + 2 \mapsto b \\
&\qquad \mathbin{\ast} dlseg(w, p', a, q, p') \\
dlist(w, p, q) &= dlseg(w, p, \mathtt{NULL}, q, \mathtt{NULL})
\end{aligned}
$$

L'astuce du ≪ou exclusif≫



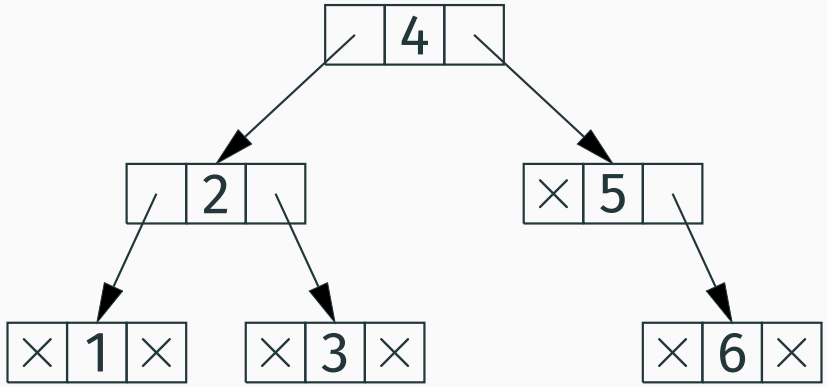
Dans chaque cellule on stocke le ≪ou exclusif≫ du pointeur vers l'avant et du pointeur vers l'arrière.

Parcours en avant : on a p et q et on retrouve $r = (p \oplus r) \oplus p$.

Parcours en arrière : on a r et q et on retrouve $p = (p \oplus r) \oplus r$.

$$\begin{aligned} dlseg(\varepsilon, p, a, q, b) &= \langle p = a \wedge q = b \rangle \\ dlseg(x \cdot w, p, a, q, b) &= \exists p', p \mapsto x \mathbin{\ast} p + 1 \mapsto b \oplus p' \mathbin{\ast} \\ &\qquad \mathbin{\ast} dlseg(w, p', a, q, p') \\ dlist(w, p, q) &= dlseg(w, p, \mathtt{NULL}, q, \mathtt{NULL}) \end{aligned}$$

Arbres binaires



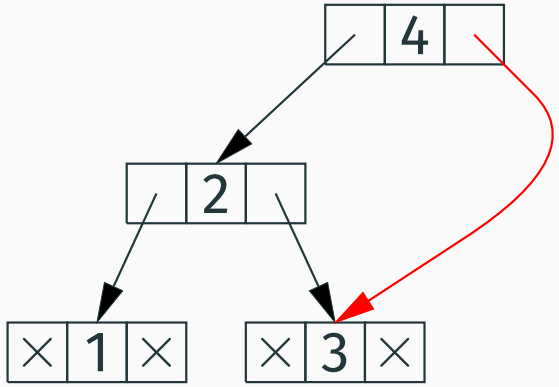
Prédicat de représentation :

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

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

Note : pas de partage interne possible, les sous-arbres doivent être disjoints.

Arbres binaires avec partage interne ($\approx$ dags)



Possible avec une *overlapping conjunction* : (Hobor et Villard, 2013)

$$(P \circledast Q)\ h = \exists h_1, h_2, h_3, h = h_1 \uplus h_2 \uplus h_3 \wedge P\ (h_1 \uplus h_2) \wedge Q\ (h_1 \uplus h_3)$$

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

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

Correction sémantique de la logique de séparation

Correction sémantique de la logique de séparation

Nous avons donné des règles définissant un triplet $\{ P \}\ c\ \{ Q \}$.

Si $\{ P \}\ c\ \{ Q \}$ est dérivable par ces règles, toutes les exécutions possibles de c valident-elles le contrat énoncé par ce triplet ?

La sémantique par réductions de PTR

On réduit des configurations c/h où h : adresses $\xrightarrow{fin}$ valeurs est l'état mémoire courant (*memory heap*).

Les règles pour les constructions pures :

$$(\texttt{let}\ x = a\ \texttt{in}\ c)/h \to c[x \leftarrow [\![a]\!]]/h$$

$$(\texttt{let}\ x = c_1\ \texttt{in}\ c_2)/h \to (\texttt{let}\ x = c_1'\ \texttt{in}\ c_2)/h' \quad \text{si } c_1/h \to c_1'/h'$$

$$(\texttt{let}\ x = c_1\ \texttt{in}\ c_2)/h \to \texttt{err} \quad \text{si } c_1/h \to \texttt{err}$$

$$(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)/h \to c_1/h \quad \text{si } [\![b]\!] \text{ est vrai}$$

$$(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)/h \to c_2/h \quad \text{si } [\![b]\!] \text{ est faux}$$

$$\texttt{choose}(N)/h \to n/h \quad \text{pour tout } n \in [0, N[$$

La sémantique par réductions de PTR

Les règles pour les constructions impératives :

$$\texttt{alloc}(N)/h \to \ell/h[\ell \leftarrow 0, \ell+1 \leftarrow 0, \ldots, \ell+N-1 \leftarrow 0]$$
$$\text{pour tout } \ell \text{ tel que } \{\ell, \ldots, \ell+N-1\} \cap \mathit{Dom}(h) = \emptyset$$
$$\texttt{get}(a)/h \to h([\![a]\!])/h \quad \text{si } [\![a]\!] \in \mathit{Dom}(h)$$
$$\texttt{set}(a, a')/h \to 0/h[\,[\![a]\!] \leftarrow [\![a']\!]\,] \quad \text{si } [\![a]\!] \in \mathit{Dom}(h)$$
$$\texttt{free}(a)/h \to 0/(h \setminus [\![a]\!]) \quad \text{si } [\![a]\!] \in \mathit{Dom}(h)$$

Règles d'erreur :

$$\texttt{get}(a)/h \to \texttt{err} \quad \text{si } [\![a]\!] \notin \mathit{Dom}(h)$$
$$\texttt{set}(a, a')/h \to \texttt{err} \quad \text{si } [\![a]\!] \notin \mathit{Dom}(h)$$
$$\texttt{free}(a)/h \to \texttt{err} \quad \text{si } [\![a]\!] \notin \mathit{Dom}(h)$$

Énoncer la correction sémantique

Même approche que pour la logique de Hoare forte.

On définit le prédicat inductif $\texttt{Term}\ c\ h\ Q$, ≪la commande c démarrée dans l'état h termine toujours sans erreurs et dans un état satisfaisant Q≫.

$$\frac{Q\ [\![a]\!]\ h}{\texttt{Term}\ a\ h\ Q}$$

$$\frac{(\forall a,\ c \neq a) \qquad c/h \not\to \texttt{err} \qquad (\forall c', h',\ c/h \to c'/h' \Rightarrow \texttt{Term}\ c'\ h'\ Q)}{\texttt{Term}\ c\ h\ Q}$$

Correction sémantique

Le triplet sémantique : ≪si l'état initial satisfait P, la commande c termine dans un état satisfaisant Q≫

$$\{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\} \quad \stackrel{def}{=} \quad \forall h,\ P\ h \Rightarrow \texttt{Term}\ c\ h\ Q$$

On montre que cette définition satisfait les axiomes et les règles d'inférences de la logique de séparation :

- Si $P \Rightarrow Q\ [\![a]\!]$ alors $\{\!\{ P \}\!\}\ a\ \{\!\{ Q \}\!\}$
- $\{\!\{ [\![a]\!] \mapsto \_ \}\!\}\ \texttt{set}(a, a')\ \{\!\{ \lambda v.\ [\![a]\!] \mapsto [\![a']\!] \}\!\}$
- etc.

Théorème (Correction sémantique de la logique de séparation)

Si $\{ P \}\ c\ \{ Q \}$ *est dérivable, alors* $\{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}$ *est vrai.*

Validité sémantique de la règle d'encadrement

La principale difficulté est de montrer que la règle d'encadrement est sémantiquement valide :

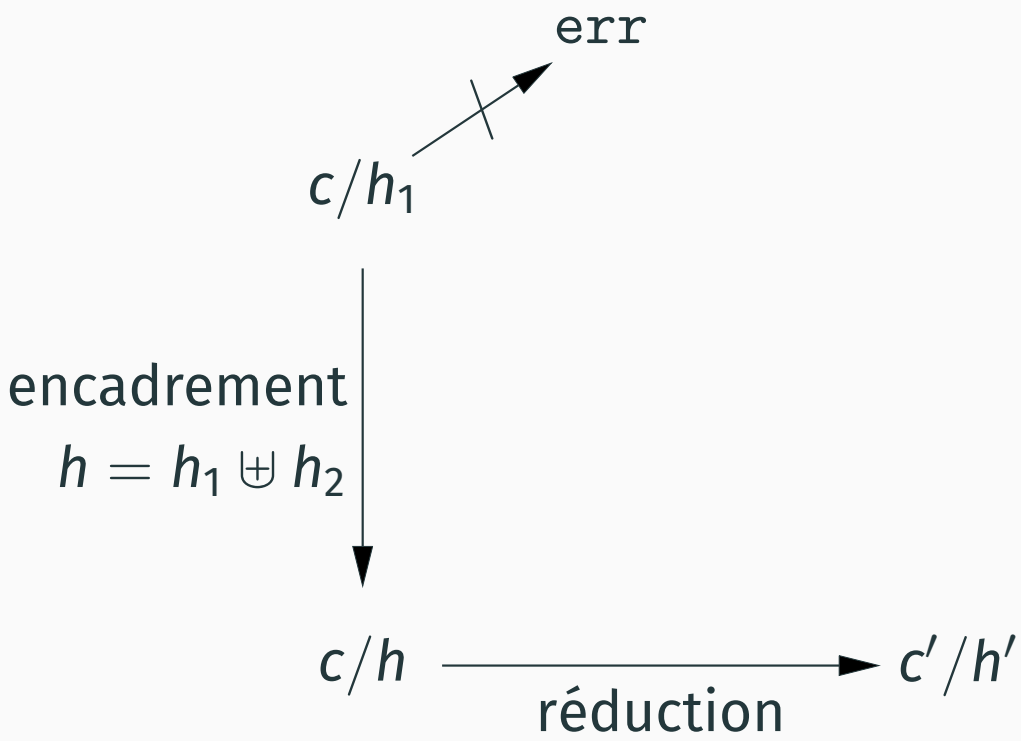
$$\text{Si } \{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\} \text{ alors } \{\!\{ P \mathbin{\ast} R \}\!\}\ c\ \{\!\{ \lambda v.\ Q\ v \mathbin{\ast} R \}\!\}.$$

Pour cela, il faut un lemme d'encadrement sur le prédicat `Term` :

$$\text{Si } \texttt{Term}\ c\ h_1\ Q \text{ et } R\ h_2 \text{ alors } \texttt{Term}\ c\ (h_1 \uplus h_2)\ (\lambda v.\ Q\ v \mathbin{\ast} R).$$

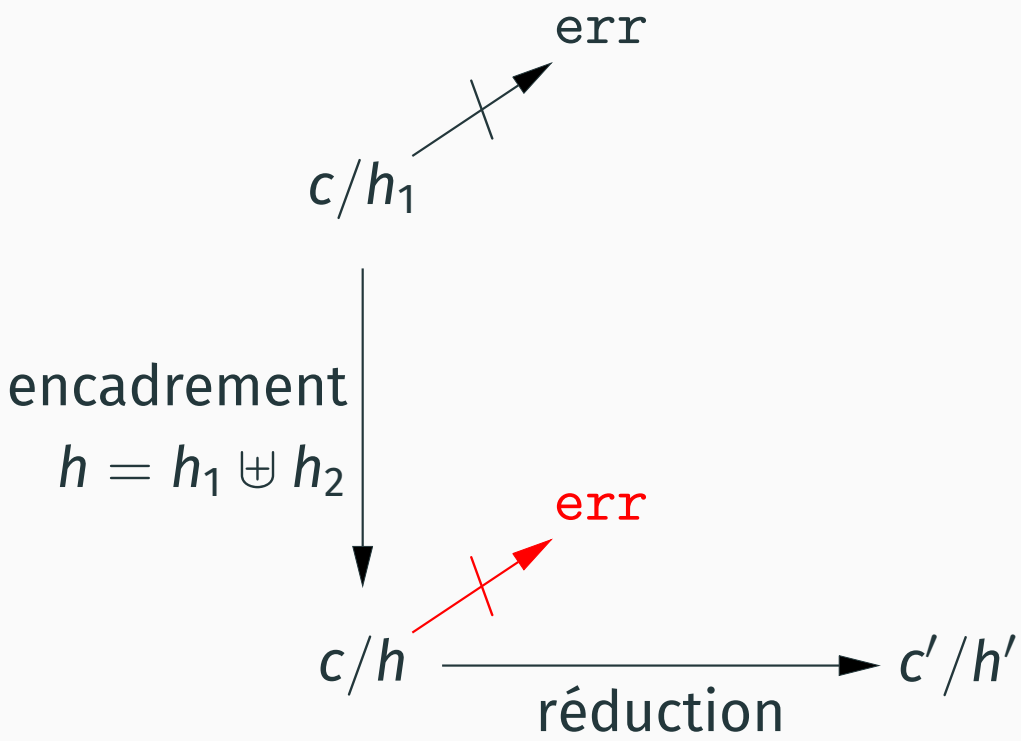
C'est vrai en raison d'une jolie propriété de la sémantique opérationnelle : si une commande ne fait pas d'erreurs dans un ≪petit≫ état, toute réduction dans un ≪grand≫ état est simulée par une réduction dans le ≪petit≫ état.

Encadrer les réductions



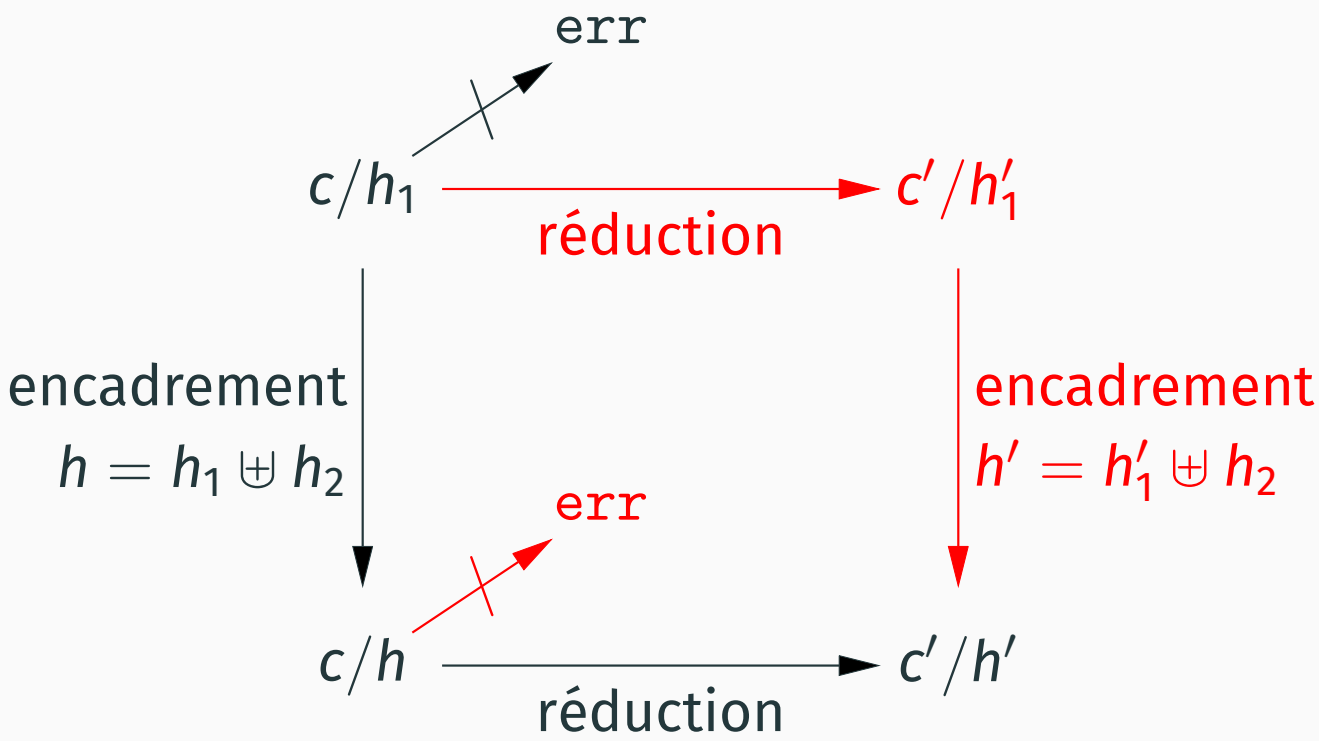
Si $c/h_1 \not\to \texttt{err}$, alors $c/h_1 \uplus h_2 \not\to \texttt{err}$. Si de plus $c/h_1 \uplus h_2 \to c'/h'$, il existe h'_1 tel que $h' = h'_1 \uplus h_2$ et $c/h_1 \to c'/h'_1$.

Encadrer les réductions



Si $c/h_1 \not\to \texttt{err}$, alors $c/h_1 \uplus h_2 \not\to \texttt{err}$. Si de plus $c/h_1 \uplus h_2 \to c'/h'$, il existe h'_1 tel que $h' = h'_1 \uplus h_2$ et $c/h_1 \to c'/h'_1$.

Encadrer les réductions



Si $c/h_1 \not\to \texttt{err}$, alors $c/h_1 \uplus h_2 \not\to \texttt{err}$. Si de plus $c/h_1 \uplus h_2 \to c'/h'$, il existe h'_1 tel que $h' = h'_1 \uplus h_2$ et $c/h_1 \to c'/h'_1$.

Encadrer les réductions

Cette propriété est vraie dans notre langage PTR parce que la règle de réduction des allocations est non-déterministe : l'adresse ℓ allouée peut être choisie parmi les adresses libres.

$$\texttt{alloc}(N)/h \to \ell/h[\ell \leftarrow 0, \ell+1 \leftarrow 0, \ldots, \ell+N-1 \leftarrow 0]$$
$$\text{pour tout } \ell \text{ tel que } \{\ell, \ldots, \ell+N-1\} \cap \mathit{Dom}(h) = \emptyset$$

Ce ne serait plus vrai si ℓ était une fonction de l'état mémoire :

$$\texttt{alloc}(N)/h \to \ell/h[\ell \leftarrow 0, \ell+1 \leftarrow 0, \ldots, \ell+N-1 \leftarrow 0]$$
$$\text{avec } \ell = \texttt{firstfree}(h, N)$$

car, en général, $\texttt{firstfree}(h_1 \uplus h_2, N) \neq \texttt{firstfree}(h_1, N)$.

Plan B : valider la règle d'encadrement par construction

Si l'allocation est déterministe, ou si on n'a pas envie de montrer la propriété d'encadrement des réductions, voici une alternative.

1. Définir le triplet sémantique de Hoare usuel :

$$\{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}_{Hoare} \overset{def}{=} \forall h,\ P\ h \Rightarrow \mathtt{Term}\ c\ h\ Q$$

2. Définir le triplet sémantique de séparation en quantifiant sur tous les encadrements possibles :

$$\{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}_{Sep} \overset{def}{=} \forall R,\ \{\!\{ P \star R \}\!\}\ c\ \{\!\{ \lambda v.\ Q\ v \star R \}\!\}_{Hoare}$$

Propriétés du triplet sémantique de Hoare

3. Montrer que le triplet sémantique $\{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}_{Hoare}$ satisfait

   - les ≪grandes≫ règles pour les constructions impératives

$$\{\!\{ P \}\!\}\ \texttt{alloc}(N)\ \{\!\{ \lambda \ell.\ \ell \mapsto \_ \ast \cdots \ast \ell + N - 1 \mapsto \_ \ast P \}\!\}_{Hoare}$$
$$\{\!\{ [\![a]\!] \mapsto x \ast P \}\!\}\ \ \texttt{get}(a)\ \ \{\!\{ \lambda v.\ \langle v = x \rangle \ast [\![a]\!] \mapsto x \ast P \}\!\}_{Hoare}$$
$$\{\!\{ [\![a]\!] \mapsto \_ \ast P \}\!\}\ \texttt{set}(a, a')\ \{\!\{ \lambda v.\ [\![a]\!] \mapsto [\![a']\!] \ast P \}\!\}_{Hoare}$$
$$\{\!\{ [\![a]\!] \mapsto \_ \ast P \}\!\}\ \ \texttt{free}(a)\ \ \{\!\{ \lambda v.\ P \}\!\}_{Hoare}$$

   - les règles usuelles pour les structures de contrôle :
     - si $P \Rightarrow Q\ [\![a]\!]$ alors $\{\!\{ P \}\!\}\ a\ \{\!\{ Q \}\!\}_{Hoare}$
     - si $\{\!\{ P \}\!\}\ c\ \{\!\{ R \}\!\}_{Hoare}$ et $\forall v, \{\!\{ Rv \}\!\}\ c'[x \leftarrow v]\ \{\!\{ Q \}\!\}_{Hoare}$ alors $\{\!\{ P \}\!\}\ \texttt{let}\ x = c\ \texttt{in}\ c'\ \{\!\{ Q \}\!\}_{Hoare}$
     - etc.
   - mais pas la règle d'encadrement.

Propriétés du triplet sémantique de séparation

$$\{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}_{Sep} \stackrel{def}{=} \forall R,\ \{\!\{ P \star R \}\!\}\ c\ \{\!\{ \lambda v.\ Q\ v \star R \}\!\}_{Hoare}$$

4. Observer que le triplet sémantique de séparation satisfait
   - les ≪petites≫ règles pour les constructions impératives ;
   - les règles pour les structures de contrôle ;
   - la règle d'encadrement.
5. Conclure que $\{ P \}\ c\ \{ Q \}$ implique $\{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}_{Sep}$ et donc aussi $\{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}_{Hoare}$.

Point d’étape

Point d'étape

L'apparition des logiques de séparation au début des années 2000 a entièrement renouvelé le domaine des logiques de programmes et de la vérification déductive.

De très nombreuses extensions, (→ cours du 01/04)
notamment vers le parallélisme à mémoire partagée.
(→ cours du 25/03 et du 08/04)

De nombreuses mises en application :

- vérification déductive + démonstration automatique (Smallfoot, Infer, VeriFast) (→ séminaire du 25/03)
- plongements dans des assistants à la démonstration (CFML, VST, Bedrock, IRIS) (→ séminaires du 01/04 et du 08/04)
- systèmes de types comme celui du langage Rust.

Bibliographie

Bibliographie

Une introduction à la logique de séparation :

- Peter O'Hearn, *Separation Logic*, Comm. ACM 62(2), 2019.

Un des articles fondateurs, toujours de référence 19 ans après :

- John C. Reynolds, *Separation Logic : A Logic for Shared Mutable Data Structures*, LICS 2002.

Mécanisations de la logique de séparation :

- Le développement Coq correspondant à ce cours : `https://github.com/xavierleroy/cdf-program-logics`
- A. Charguéraud, *Foundations of Separation Logic*, 2021, `https://www.charglueraud.org/teach/verif/`