

Logique de Floyd-Hoare-Dijkstra en “correction partielle”

pour expliquer le plugin WP de FRAMA-C

Inventée par R. Floyd, T. Hoare et E. Dijkstra en 1967-1975

Formalise la vérification de propriétés notées “ $\{P\} S \{Q\}$ ”

*Sur tout état initial satisfaisant la **pré-condition** P ,
l'exécution du **code** S n'a **pas d'erreur** à l'exécution
et si elle termine, alors l'état final satisfait la **post-condition** Q .*

Qu'est-ce ? Sémantique formelle (*axiomatique*) de programmes adaptée pour la vérification de programmes (*logique de programme*).

Pour ? Expliquer/justifier les outils comme FRAMA-C (C) <https://frama-c.com/> ou CREUSOT (RUST) <https://github.com/creusot-rs/creusot>.

Comment ? Dérive des **obligations de preuves** complexes à partir d'un jeu réduit de règles d'inférence (axiomes).

Plan du Cours 2

Concepts de base

Règles pour les commandes primitives

Commandes gardées dérivées

Application pour expliquer le plugin WP de FRAMA-C

Conditions logiques (notées P , Q , I , C suivant le contexte)

Condition $\stackrel{def}{=}$ proposition en logique du 1er ordre représentant un *ensemble d'états* (où *état* : map fini variables \rightarrow valeurs)

Exemple

$$(\exists k, x = 2 \cdot k) \wedge (x \neq 0 \Rightarrow z = x \cdot y)$$

Exo Indiquer les états qui satisfont la condition ci-dessus

1. $\{x \mapsto 4, y \mapsto 5, z \mapsto 20, t \mapsto 17\} \dots$
2. $\{x \mapsto 3, y \mapsto 5, z \mapsto 15\} \dots$
3. $\{x \mapsto 4, y \mapsto 5, z \mapsto 15\} \dots$
4. $\{x \mapsto 0, y \mapsto 5, z \mapsto 20\} \dots$
5. $\{x \mapsto 4, y \mapsto 5, z \mapsto 20, k \mapsto 17\} \dots$

NB on rappelle que dans " $(\exists k, x = 2 \cdot k)$ ", k est **variable liée**
c-à-d. renommable avec un nom **frais** sans changer la signification !

Représentation intermédiaire du code (motivation)

FRAMA-C (pour C) et CREUSOT (pour RUST) ont un même *back-end* de vérification **WHY3** - <https://www.why3.org/>

Sépare ainsi

1. les principes généraux pour vérifier les programmes en logique de Hoare, implémentés par **WHY3** ;
2. l'encodage des sémantiques *complexes* (cf. TD1) de C et de RUST dans l'IR (*Intermediate Representation*) de **WHY3**

Dans ce cours une IR de **commandes gardées** à la Dijkstra avec syntaxe inspirée des **expressions régulières**.

Idée des commandes gardées (notées S pour *Statement*)

Commande gardée comme calcul **non-déterministe**

d'un *état initial* à *état final* ou à une **erreur**

Erreur uniquement provoquée par commande spéciale **“abort”**

Non-déterminisme pour représenter le *non-déterminé* :

résultat de `malloc`, interactions utilisateurs, choix `compilo`, etc

Exemples et quasi-analogie avec expressions régulières (ER)

$x := y + 1; x := x + x$ équivaut à $x := 2(y + 1)$
 “;” **quasi-analogue** au “.” des ER

$(x := x + 1)^*$ incrémente x d'un entier naturel n arbitraire

$x := x + 1 \sqcup \varepsilon$ incrément optionnel de x
 “ \sqcup ” **analogue** au “+” des ER

\emptyset ne termine pas (ou bloque sans état final)

Quasi-analogie : \emptyset et **abort** deux absorbants à **gauche** pour “;”
 donc aucun n'est absorbant à droite !

Triplets de Hoare (exos)

Le triplet de Hoare “ $\{P\} S \{Q\}$ ” signifie

*Si l'état initial satisfait la **précondition** P ,
alors aucune exécution de la **commande** S n'atteint **abort**
et tout état final satisfait la **postcondition** Q .*

Exo 1 trouver sur chaque triplet, la WP (*Weakest-Precondition*) P qui le rend valide, où “ P_1 plus faible que P_2 ” signifie $P_2 \Rightarrow P_1$

$\{P\} x := x + 1 \{x \geq 5\}$ $\{P\} \mathbf{abort} \{x \geq 5\}$ $\{P\} \emptyset \{x \geq 5\}$

...

...

...

Exo 2 dans chaque cas, si l'état initial satisfait précondition P ,
quelles garanties sur S ?

$\{P\} S \{\mathbf{true}\}$ $\{P\} S \{\mathbf{false}\}$ $\{\mathbf{false}\} S \{Q\}$

...

...

...

Variables de programmes comme variables logiques

Dans “ $\{x = 2 \cdot a\} x := x + 1 \{x = 2 \cdot a + 1\}$ ”

- ▶ “ x ” et “ a ” sont deux *variables logiques*
- ▶ “ x ” est aussi une *variable de programme* qui représente un **fragment non-aliasé** de l'état.

Par hypothèse du formalisme, ce “**fragment non-aliasé**” a :

- ▶ “ x ” comme **unique identificateur** ;
- ▶ il est **disjoint** de tous les autres “fragments de l'état”.

Exemple En FRAMA-C, une telle variable peut correspondre à

- ▶ une variable du source C jamais précédée par un “&”
- ▶ un bloc mémoire de la sémantique du C :
tableaux, champs de structures potentiellement aliasées, etc.

Règle d'inférence en logique de Hoare

Une *règle de Hoare* est un système formel de déduction formel noté

$$\frac{\{P_1\} S_1 \{Q_1\} \quad \dots \quad \{P_n\} S_n \{Q_n\}}{\{P\} S \{Q\}} \text{ "extra-conditions"}$$

qui signifie

Si $\{P_1\} S_1 \{Q_1\}$ et ... et $\{P_n\} S_n \{Q_n\}$
 et si "extra-conditions" (vérifiées "par ailleurs")
 alors $\{P\} S \{Q\}$.

Exemple de 2 règles primitives (sans extra-condition)

$$\frac{}{\{\text{false}\} \text{abort} \{Q\}} \quad \frac{\{P\} S_1 \{I\} \quad \{I\} S_2 \{Q\}}{\{P\} S_1; S_2 \{Q\}}$$

Inférence de Hoare et règle dérivée

Une *inférence de Hoare* est une combinaison valide des règles (qui établit donc une *preuve formelle*).

Exemple

$$\frac{\{P\} S \{false\} \quad \overline{\{false\} \mathbf{abort} \{Q\}}}{\{P\} S; \mathbf{abort} \{Q\}}$$

L'inférence ci-dessus établit une *règle dérivée* (c-à-d un *théorème formel*) marqué avec un D :

$$D \frac{\{P\} S \{false\}}{\{P\} S; \mathbf{abort} \{Q\}}$$

La règle de “*conséquence logique*”

Règle primitive (la seule avec extra-conditions non syntaxiques!)

$$\frac{\{P'\} S \{Q'\} \quad P \Rightarrow P'}{\{P\} S \{Q\} \quad Q' \Rightarrow Q}$$

Exemple d'inférence $\frac{\{P\} S \{\text{false}\}}{\{\text{true}\} S \{Q\}} P$

où extra-conditions “ $\text{true} \Rightarrow P$ ” et “ $\text{false} \Rightarrow Q$ ” simplifiées en extra-condition “ P ”.

Moralement, FRAMA-C *construit* une **inférence de Hoare** pour *chaque fonction source*. Et les **obligations de preuves** envoyées au solveur-SMT viennent des extra-conditions de cette règle!

Intuition sur les systèmes d'inférence

utilisées abondamment en logique et en sémantique formelle

Inférence = **arbre** de calculs (racine en bas - feuilles en haut)
avec **propagations d'attributs**

Systèmes d'inférence généralisent **grammaires attribuées**
sans ordre imposé de propagation parent/enfants par non-terminal
(et principes d'induction plus généraux)

Dans la suite, règles de Hoare sur $\{P\} S \{Q\}$:
pour **analyse** sur *non-terminal* S avec attributs P et Q ,
et une **unique règle primitive** par commande primitive.

Plan du Cours 2

Concepts de base

Règles pour les commandes primitives

Commandes gardées dérivées

Application pour expliquer le plugin WP de FRAMA-C

Erreur et séquence

Règles primitives vues en diapo 8.

$$\frac{}{\{false\} \mathbf{abort} \{Q\}} \qquad \frac{\{P\} S_1 \{I\} \quad \{I\} S_2 \{Q\}}{\{P\} S_1; S_2 \{Q\}}$$

On *abrège* les preuves de $\{I_0\} S_1; \dots; S_n \{I_n\}$ sous la forme

-- I_0
 S_1 ; -- I_1
 ...
 S_n ; -- I_n

quand chaque preuve $\frac{}{\{I_{k-1}\} S_k \{I_k\}}$ est une feuille !

NB L'arbre peut **toujours** être *implicite*, via notation diapo 23.
 L'arbre n'est utile que pour formaliser la notation !

Affectation “ $x := T$ ” où “ T ” est un *terme logique*

Règle primitive

$$\overline{\{Q[x \leftarrow T]\} x := T \{Q\}}$$

où $Q[x \leftarrow T]$ est la substitution des occurrences x par T dans Q , sans capture des variables libres de T par une variable liée de Q .

Exo Dériver

$$D \frac{}{\{x = a\} x := x - 1; x := x + x \{x = 2 \cdot a - 2\}}$$

$$D \frac{}{\{P\} x := T \{\exists x_0, P[x \leftarrow x_0] \wedge x = T[x \leftarrow x_0]\}} \quad x_0 \text{ frais pour } P, T \text{ et } x$$

Choix non-déterministe “ $S_1 \sqcup S_2$ ”

Règle primitive
$$\frac{\{P\} S_1 \{Q\} \quad \{P\} S_2 \{Q\}}{\{P\} S_1 \sqcup S_2 \{Q\}}$$

Intuition triplet garanti quelque soit le choix à l'exécution

Exo Dériver

$$D \frac{\{P_1\} S_1 \{Q\} \quad \{P_2\} S_2 \{Q\}}{\{P_1 \wedge P_2\} S_1 \sqcup S_2 \{Q\}}$$

$$D \frac{\{P\} S_1 \{Q_1\} \quad \{P\} S_2 \{Q_2\}}{\{P\} S_1 \sqcup S_2 \{Q_1 \vee Q_2\}}$$

$$D \frac{}{\{x = a\} x := x + 1; (y := x \sqcup y := 2 \cdot x) \{y = a + 1 \vee y = 2 \cdot a + 2\}}$$

Garde “**assume C**” où C est une condition pour contrôler les choix non-déterministes !

Si C , alors “**assume C**” ne change rien, sinon boucle infiniment.

Règle primitive

$$\overline{\{C \Rightarrow Q\} \text{ assume } C \{Q\}}$$

Définitions $\varepsilon \stackrel{\text{def}}{=} \text{assume true}$ $\emptyset \stackrel{\text{def}}{=} \text{assume false}$

Exo Dériver

$$D \overline{\{P\} \text{ assume } C \{P \wedge C\}}$$

$$D \overline{\{I\} \varepsilon \{I\}}$$

$$D \overline{\{P\} \emptyset \{\text{false}\}}$$

Itération finie non-déterministe “ S^* ”

La commande “ S^* ” est équivalente à “ $\varepsilon \sqcup (S; S^*)$ ”.

Règle primitive

$$\frac{\{I\} S \{I\}}{\{I\} S^* \{I\}}$$

Exo Dériver

$$D \frac{\{I\} S \{I\}}{\{I\} \varepsilon \sqcup (S; S^*) \{I\}}$$

Déclaration de variable locale “**var** $x(S)$ ”

La commande “**var** $x(S)$ ”

1. alloue une nouvelle variable de programme “ x ” (masquant un éventuel “ x ” englobant) avec une valeur choisie arbitrairement ;
2. exécute S ;
3. désalloue x (l'éventuel x englobant devient à nouveau visible).

Règle primitive

$$\frac{\{P\} S[x \leftarrow x_0] \{Q\}}{\{P\} \mathbf{var} x(S) \{Q\}} \quad x_0 \text{ frais pour } P, Q \text{ et } S$$

Exo Dériver

$$D \frac{}{\{x = 7 \wedge y = 2\} \mathbf{var} x(x := 1; y := y + x); x := x + 1 \{x = 8 \wedge y = 3\}}$$

$$D \frac{}{\{7 \leq y \leq a\} \mathbf{var} x(\mathbf{assume} y \leq x \leq 3 \cdot y; y := x) \{7 \leq y \leq 3 \cdot a\}}$$

Résumé des commandes et règles primitives

$S ::= \mathbf{abort} \mid x := T \mid \mathbf{assume} C \mid S_1; S_2 \mid S_1 \sqcup S_2 \mid S^* \mid \mathbf{var} x(S)$

$$\frac{\{P'\} S \{Q'\}}{\{P\} S \{Q\}} \quad P \Rightarrow P' \quad Q' \Rightarrow Q$$

$$\frac{}{\{\mathbf{false}\} \mathbf{abort} \{Q\}}$$

$$\frac{}{\{Q[x \leftarrow T]\} x := T \{Q\}}$$

$$\frac{}{\{C \Rightarrow Q\} \mathbf{assume} C \{Q\}}$$

$$\frac{\{P\} S_1 \{I\} \quad \{I\} S_2 \{Q\}}{\{P\} S_1; S_2 \{Q\}}$$

$$\frac{\{P\} S_1 \{Q\} \quad \{P\} S_2 \{Q\}}{\{P\} S_1 \sqcup S_2 \{Q\}}$$

$$\frac{\{I\} S \{I\}}{\{I\} S^* \{I\}}$$

$$\frac{\{P\} S[x \leftarrow x_0] \{Q\}}{\{P\} \mathbf{var} x(S) \{Q\}} \quad x_0 \text{ "frais"}$$

Plan du Cours 2

Concepts de base

Règles pour les commandes primitives

Commandes gardées dérivées

Application pour expliquer le plugin WP de FRAMA-C

Commande “**assert C**” avec C condition

La commande utilisée pour les “assert” RTE de FRAMA-C!

$$\mathbf{assert\ } C \stackrel{def}{=} (\mathbf{assume\ } \neg C; \mathbf{abort}) \sqcup \varepsilon$$

Exo Dériver

$$D \frac{}{\{C \wedge Q\} \mathbf{assert\ } C \{Q\}}$$

$$D \frac{}{\{I\} \mathbf{assert\ } C \{I\}} I \Rightarrow C$$

If-then-else et boucle while

if C **then** S_1 **else** $S_2 \stackrel{\text{def}}{=} (\text{assume } C; S_1) \sqcup (\text{assume } \neg C; S_2)$
while C **do** $S \stackrel{\text{def}}{=} (\text{assume } C; S)^*; (\text{assume } \neg C)$

Exo Dériver

$$D \frac{\{P \wedge C\} S_1 \{Q\} \quad \{P \wedge \neg C\} S_2 \{Q\}}{\{P\} \text{if } C \text{ then } S_1 \text{ else } S_2 \{Q\}}$$

$$D \frac{\{P_1\} S_1 \{Q\} \quad \{P_2\} S_2 \{Q\}}{\{(C \Rightarrow P_1) \wedge (\neg C \Rightarrow P_2)\} \text{if } C \text{ then } S_1 \text{ else } S_2 \{Q\}}$$

$$D \frac{\{I \wedge C\} S \{I\}}{\{I\} \text{while } C \text{ do } S \{I \wedge \neg C\}}$$

Stratégie de preuve et recherche d'invariant de boucle

P et Q étant fixés - quel invariant I pour prouver " $\{P\} L_3 - L_{15} \{Q\}$ " ?

```

1  -- P
2  -- J
3  x1 := T1; -- I
4  #[loop invariant I]
5  while C1 do -- I1
6    if C2 then -- I2
7      -- J2
8      x2 := T2 -- I
9    else -- I3
10     -- J3
11     x3 := T3 -- I
12   end -- I
13 end; -- I4
14 -- J4
15 x4 := T4 -- Q
16 -- Q

```

Structure de la preuve :

$$\frac{\frac{\overline{\{J\} L_3 \{I\}}}{\{P\} L_3 \{I\}} \quad P \Rightarrow J \quad \frac{\quad \dagger}{\{I\} L_5 - L_{13} \{I_4\}} \quad \frac{\overline{\{J_4\} L_{15} \{Q\}}}{\{I_4\} L_{15} \{Q\}}}{\{P\} L_3 - L_{13} \{I_4\} \quad \{I_4\} L_{15} \{Q\}} \quad I_4 \Rightarrow J_4}{\{P\} L_3 - L_{15} \{Q\}}$$

où $P \Rightarrow J$ est l'initialisation de l'invariant

et $I_4 \Rightarrow J_4$ est la preuve de la postcondition de la boucle

et \dagger est la preuve de **préservation de l'invariant** ci-dessous :

$$\frac{\frac{\overline{\{J_8\} L_6 \{I\}}}{\{I_2\} L_8 \{I\}} \quad I_2 \Rightarrow J_2 \quad \frac{\overline{\{J_3\} L_{11} \{I\}}}{\{I_3\} L_{11} \{I\}} \quad I_3 \Rightarrow J_3}{\{I_1\} L_6 - L_{12} \{I\}}$$

stratégie de
propagation

gardés en avant

"=" en arrière

avec $I_1 \stackrel{def}{=} I \wedge C_1$; $I_2 \stackrel{def}{=} I_1 \wedge C_2$; $I_3 \stackrel{def}{=} I_1 \wedge \neg C_2$; $I_4 \stackrel{def}{=} I \wedge \neg C_1$

$J \stackrel{def}{=} I[x_1 \leftarrow T_1]$; $J_2 \stackrel{def}{=} I[x_2 \leftarrow T_2]$; $J_3 \stackrel{def}{=} I[x_3 \leftarrow T_3]$; $J_4 \stackrel{def}{=} Q[x_4 \leftarrow T_4]$

Méthode : trouver le I qui rend valide les 4 **extra-conditions** ci-dessus.

Problème indécidable D'où la nécessité d'annoter les boucles avec des invariants !

Procédures (éventuellement récursives)

Etant donné une déf. de procédure “**pr** $foo(\vec{x}, \&\text{mut } \vec{y})[P, Q, S]$ ” où

- ▶ paramètres \vec{x} (resp. \vec{y}) “par copie” (resp. “par variable”);
- ▶ précondition P sur valeurs initiales des \vec{x} et \vec{y} ;
- ▶ postcondition Q sur valeurs initiales des \vec{x} , sur valeurs **finales** des \vec{y} , et sur valeurs initiales des \vec{y} **écrites** “ $@\vec{y}$ ”;
- ▶ le corps de S n’a aucune variable libre en dehors des \vec{x} et \vec{y} (e.g. passe explicitement var. globales en params par variable).

Vérif. sur la définition de foo avec $@\vec{x}$ “frais” (pour P , Q et S)
 $\{P[\vec{x} \leftarrow @\vec{x}, \vec{y} \leftarrow @\vec{y}]\} (\vec{x} := @\vec{x}; \vec{y} := @\vec{y}); S \{Q[\vec{x} \leftarrow @\vec{x}]\}$

NB \vec{x} modifiables localement dans S , mais invisible pour l’appelant.

Commande pour l’appel à foo avec \vec{y}_f “frais” (pour \vec{T} , Q et \vec{z})

$foo(\vec{T}, \&\text{mut } \vec{z}) \stackrel{\text{def}}{=} \text{assert } P[\vec{x} \leftarrow \vec{T}, \vec{y} \leftarrow \vec{z}];$
 $\quad \text{var } \vec{y}_f \left(\text{assume } Q[\vec{x} \leftarrow \vec{T}, @\vec{y} \leftarrow \vec{z}, \vec{y} \leftarrow \vec{y}_f]; \vec{z} := \vec{y}_f \right)$

NB \vec{x} et \vec{y} peuvent apparaître dans termes \vec{T} et variables \vec{z} .

Exercice (cf. en TD)

1) Vérifier la définition ci-dessous

```

pr rem( $x$ , &mut  $y$ ) [
     $x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x \neq 0$ ,
     $y = @y \bmod x$ ,
    while  $x \leq y$  do  $y := y - x$ 
]
  
```

c-à-d que “rem(x , &mut y)” **simule**

“**assert** ($x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x \neq 0$); $y := y \bmod x$ ”

2) Vérifier (sur la commande gardée d'appel de procédure)

$$D \frac{}{\{x = 19\} \text{rem}(x - 15, \&\text{mut } x) \{x = 3\}}$$

Commande **return** et fonctions

sans param. par variable pour simplifier...

Defs de fonction “**fn** $f(\vec{x})[P_f, Q_f, S_f]$ ” où

- ▶ précondition P_f sur params \vec{x} “par copie” ;
- ▶ postcondition Q_f sur \vec{x} et variable spéciale $\backslash \mathbf{r}$ du résultat ;
- ▶ corps de S_f sans variable libre en dehors de \vec{x}
- ▶ **nouvelle** commande **return**
avec règle primitive selon fonction f englobante

$$\overline{\{Q_f[\vec{x} \leftarrow @\vec{x}, \backslash \mathbf{r} \leftarrow T]\} \mathbf{return} T \{Q\}}$$

Vérif. sur la définition de f avec $@\vec{x}$ “frais” (pour P_f , Q_f et S_f)

$$\{P_f[\vec{x} \leftarrow @\vec{x}]\} \vec{x} := @\vec{x}; S_f \{false\}$$

Commande pour l'appel à f avec r_0 “frais” (pour \vec{T} , Q_f et y)

$$y \leftarrow f(\vec{T}) \stackrel{def}{=} \mathbf{assert} P_f[\vec{x} \leftarrow \vec{T}; \\ \mathbf{var} r_0 \left(\mathbf{assume} Q_f[\vec{x} \leftarrow \vec{T}, \backslash \mathbf{r} \leftarrow r_0]; y := r_0 \right)$$

Plan du Cours 2

Concepts de base

Règles pour les commandes primitives

Commandes gardées dérivées

Application pour expliquer le plugin WP de FRAMA-C

Encodage d'un langage dans l'IR de vérification

Principe général

1. on encode chaque construction du langage dans l'IR
2. cet encodage induit une règle dérivée
3. on vérifie que la règle dérivée est bien correcte vis-à-vis de la "sémantique attendue"

NB Pour un vrai langage comme C, c'est complexe !

L'IR de WHY3, plus sophistiquée que les commandes gardées, permet de gérer break, continue, goto, etc.

Comment “éviter” les “erreurs” dans les termes logiques ?

Par exemple, en C, dans contexte “**int** x, y;” pour raisonner sur expressions “x+y” et “x/y” avec UB potentiels...

Solution pour “x+y”

1. vérifier que le contexte garantit “x+y” entier signé 32 bits via “**assert** ($-2^{31} \leq x + y < 2^{31}$)” pour + dans \mathbb{Z} .
2. dans conditions logiques, raisonner sur “x+y” avec + de \mathbb{Z} , sans avoir besoin de modéliser davantage le + de **int**.

Termes indéterminés dans les conditions

Pour “ x/y ”, RTE introduit un “**assert** ($y \neq 0$)”.

Mais aussi besoin représenter le résultat de x/y par un terme x/y .

Solution axiomatisation où le terme x/y existe, mais avec valeur **indéterminée** si $y = 0$.

Axiome de typage de la division entière

$$\forall x \forall y, x \in \mathbb{Z} \wedge y \in \mathbb{Z} \wedge y \neq 0 \Rightarrow x/y \in \mathbb{Z}$$

Axiome sémantique de la division entière

$$\forall x \forall y, x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge 0 < y \Rightarrow 0 \leq x - (y \cdot (x/y)) < y$$

NB dans une telle axiomatisation, la propriété suivante est valide

$$\forall x \forall a \forall b, (x = a + b \wedge a \in \mathbb{Z}) \Rightarrow x/(a - a) = (a + b)/0$$

elle est vraie quelque soit l'interprétation de la division par zéro.

Admettre cette propriété ne peut pas nous cacher un bug!

Tableaux et pointeurs

Tableaux représentés par des termes logiques (cf. TD2) avec

- ▶ “ $\text{select}(t, i)$ ” représentant la valeur à l'indice i dans le tableau t
- ▶ “ $\text{store}(t, i, v)$ ” représentant la **copie** du tableau t où la case d'indice i vaut v .

Pour “**int** $t[N]$;” l'affectation “ $t[i]=t[j]+1$;” encodée en :

```
assert( $0 \leq i < N \wedge 0 \leq j < N$ );  
 $t := \text{store}(t, i, \text{select}(t, j) + 1)$ 
```

FRAMA-C encode la mémoire complète des programmes C comme une collection de tableaux disjoints, où les pointeurs sont des indices dans ces tableaux.

Encodage *simplifié* d'une spécification à la ACSL

ACSL = la logique de FRAMA-C, cf. variante de div en TD1

```
/*@ requires P; assigns L; ensures Q; */
void div(int x, int *q, int *y) { *q=*y/x; *y=*y%x; }
```

```
pr div(x, q, y, &mut m)[
  x ∈ INT ∧ T ∧ Φ(P),
  T ∧ A ∧ Φ(Q[\old(*q) ← select(@m, q), \old(*y) ← select(@m, y)]),
  (assert(x ≠ 0); m := store(m, q, select(m, y)/x);
    m := store(m, y, select(m, y) mod x))
] où
```

m bloc mémoire des pointeurs de type int* potentiellement aliés
 INT ensemble des entiers signés
 \mathcal{T} condition de typage " $\text{select}(m, q) \in \text{INT} \wedge \text{select}(m, y) \in \text{INT}$ "
 $\Phi(P)$ condition " $P[*q \leftarrow \text{select}(m, q), *y \leftarrow \text{select}(m, y)]$ "
 \mathcal{A} encode assigns par " $\forall i, i \notin L [*q \leftarrow q, *y \leftarrow y] \Rightarrow \text{select}(m, i) = \text{select}(@m, i)$ "

Vers un modèle mémoire plus réaliste

Modèle mémoire simplifié de la diapo précédente : correct uniquement pour allocation statique globale !

Nécessité de prendre en compte les (dés)allocations dynamiques dans la pile et dans le tas, via des **permissions** elles-mêmes stockées en mémoire (dans le modèle sémantique).

Exemples

```
free(p);           supprime les permissions d'accès à p.  
char *p = "bonjour";    données de p en lecture seule.
```

En ACSL, on a typiquement :

- ▶ `\valid_read(p)` requis sur les accès `*p` en lecture.
- ▶ `\valid(p)` (qui implique `\valid_read(p)`) requis sur les accès `*p` en écriture.

Preuve correction fonctionnelle du `binary_search`

Via “`frama-c-gui -wp -wp-rte signed_binary_search_full.c`”

```
/*@ requires 0 <= n && \valid_read(t+(0..n-1))
   @   && \forall int i,j; 0 <= i <= j < n ==> t[i] <= t[j];
   @ assigns \nothing;
   @ ensures (0 <= \result < n && t[\result] == v) ||
   @   (\result == -1 &&
   @     \forall int k; 0 <= k < n ==> t[k] != v); */
int binary_search(long t[], int n, long v)
```

Par rapport à `signed_binary_search_rte.c`

la spécification ajoute précondition que tableau est trié

Postcondition garantit absence de `v` dans le tableau sur résultat `-1`

Nécessite invariant de boucle exprimant absence de `v` à tours de boucle précédents.