université
PARIS-SACLAY

UE14 · Vérification déductive de programmes

Modèles mémoires

Master Informatique · FIIL · 2017 / 2018

1. Fin du cours 4

Boucles

$$\mathrm{WP}(\texttt{while } t \texttt{ invariant } J \texttt{ variant } s \cdot \prec \texttt{ do } e \texttt{ done}, Q) =$$
$$J \wedge$$
$$\forall w_1, \ldots, w_k.$$
$$(J \wedge t \rightarrow \mathrm{WP}(e, J \wedge s \prec s[\vec{x} \mapsto \vec{w}]))[\vec{x} \mapsto \vec{w}] \wedge$$
$$(J \wedge \neg t \rightarrow Q)[\vec{x} \mapsto \vec{w}]$$

Considérons une boucle `for(i = 0; i < n; i++);` :

- invariant J : `loop invariant 0 <= i <= n;`
- variant $s \cdot \prec$ pour la correction totale : `loop variant n-i;`
- variables $\vec{x}$ modifiées par la boucle : `loop assigns i;`

Exercice 5

1. Spécifier en ACSL et prouver avec WP la fonction suivante (fichier `code/m.c`) :

```
int m(int x, int y) {
  int res = 0;
  int a = 0;
  while (a < x) {
    res += y;
    a++;
  }
  return res;
}
```

facultatif et **très** difficile : prouver l'absence d'erreur à l'exécution (nécessite des prédicats intermédiaires, voir slide suivant).

Prédicats, lemmes et fonctions logiques

- lemme

```
/*@ lemma add_even: \forall integer x,y;
      x % 2 == 0 ==> y % 2 == 0 ==> (x+y) % 2 == 0; */
```

- prédicat : `/*@ predicate is_even(integer n) = n % 2 == 0; */`

- fonction logique
  - explicite : `/*@ logic integer next(integer n) = n+1; */`
  - axiomatisée :

```
/*@ axiomatic fib {
      logic integer fib(integer n);
      axiom fib0: fib(0) = 0;
      axiom fib1: fib(1) = 1;
      axiom fibn: \forall integer b;
        n >= 2 ==> fib(n) == fib(n-1) + fib(n-2);
    } */
```

Aide aux spécifications et aux preuves

Exercice 6

1. définir une axiomatique caractérisant $\sum_{i=0}^{n} i$
2. à l'aide de la question 1, spécifier et vérifier le programme suivant (fichier sum.c) :

```
int sum(int n) {
  int i = 0;
  int res = 0;
  for(int i = 0; i < n; i++) res += i;
  return res;
}
```

3. facultatif : prouver l'absence d'erreur à l'exécution (hint : de quelle hypothèse supplémentaire avez-vous besoin ?)

2. Bref rappel du cours 3

Problème d'alias

```
let double_incr (s1 s2: ref int): unit writes {s1,s2}
  ensures { !s1 = 1 + old !s1 /\ !s2 = 2 + old !s2 }
= s1 := 1 + !s1; s2 := 2 + !s2

let wrong () =
  let s = ref 0 in
  double_incr s s;   (* alias écriture/écriture *)
  assert { !s = 1 /\ !s = 2 }   (* en fait, !s = 3 *)
```

Problème d'alias

```
let double_incr (s1 s2: ref int): unit writes {s1,s2}
  ensures { !s1 = 1 + old !s1 /\ !s2 = 2 + old !s2 }
= s1 := 1 + !s1; s2 := 2 + !s2

let wrong () =
  let s = ref 0 in
  double_incr s s;   (* alias écriture/écriture *)
  assert { !s = 1 /\ !s = 2 }   (* en fait, !s = 3 *)
```

```
val g : ref int

let set_from_g (r: ref int): unit writes {r}
  ensures { !r = !g + 1 }
= r := !g + 1

let wrong () =
  set_from_g g;   (* alias lecture/écriture *)
  assert { !g = !g + 1 }   (* contradiction *)
```

Problème d'alias

La logique de Hoare, le calcul de WP exigent l'absence d'alias !

- au moins, en ce qui concerne des valeurs modifiées
- Why3 relâche cette restriction grâce à un contrôle statique d'alias
- toute donnée mutable renvoyée par une fonction est fraîche

Problème d'alias

La logique de Hoare, le calcul de WP exigent l'absence d'alias !

- au moins, en ce qui concerne des valeurs modifiées
- Why3 relâche cette restriction grâce à un contrôle statique d'alias
- toute donnée mutable renvoyée par une fonction est fraîche

On doit indiquer les dépendances externes pour les fonctions :

- `val set_from_g (r: ref int): unit writes {r} reads {g}`
- sinon le contrôle statique n'a pas assez d'information

Problème d'alias

La logique de Hoare, le calcul de WP exigent l'absence d'alias !

- au moins, en ce qui concerne des valeurs modifiées
- Why3 relâche cette restriction grâce à un contrôle statique d'alias
- toute donnée mutable renvoyée par une fonction est fraîche

On doit indiquer les dépendances externes pour les fonctions :

- `val set_from_g (r: ref int): unit writes {r} reads {g}`
- sinon le contrôle statique n'a pas assez d'information

Pour vérifier des programmes avec des pointeurs arbitraires
on a besoin d'approches plus sophistiquées

- modèle mémoire (par exemple, tableaux « adresse $\longrightarrow$ valeur »)
- gérer les alias dans les VC :logique de séparation, *dynamic frames*...
- voir la 2^{e} partie du cours

Problème d'alias

La logique de Hoare, le calcul de WP exigent l'absence d'alias !

- au moins, en ce qui concerne des valeurs modifiées
- Why3 relâche cette restriction grâce à un contrôle statique d'alias
- toute donnée mutable renvoyée par une fonction est fraîche

On doit indiquer les dépendances externes pour les fonctions :

- `val set_from_g (r: ref int): unit writes {r} reads {g}`
- sinon le contrôle statique n'a pas assez d'information

Pour vérifier des programmes avec des pointeurs arbitraires
on a besoin d'approches plus sophistiquées

- modèle mémoire (par exemple, tableaux « adresse $\longrightarrow$ valeur »)
- gérer les alias dans les VC :logique de séparation, *dynamic frames*...
- voir la 2^{e} partie du cours

On y est :-) !

3. Modèle mémoire

Modèle mémoire : définition

- En programmation impérative, les programmes sont des séquences de commandes qui mettent à jour un état mémoire (ou simplement état ou mémoire).
- Par exemple, l'affectation `x := x+1;` modifie l'état pour associer à `x` son ancienne valeur incrémentée de 1.
- Un modèle mémoire est une spécification des états et des opérations associées (lectures et écritures).

Un modèle mémoire est (notamment) utile pour :

- formaliser la sémantique des langages impératifs
- vérifier des propriétés de programmes impératifs (ce cours !)
- prouver des transformations de programmes

Un premier modèle mémoire

- type $\texttt{pointer}_\tau$ d'un pointeur vers une valeur de type τ
- type d'un état mémoire $\texttt{memory}$
- fonction pour lire un pointeur :

$$\text{read} : \texttt{memory} \to \texttt{pointer}_\tau \to \tau$$

- fonction pour écrire un pointeur :

$$\text{write} : \texttt{memory} \to \texttt{pointer}_\tau \to \tau \to \texttt{memory}$$

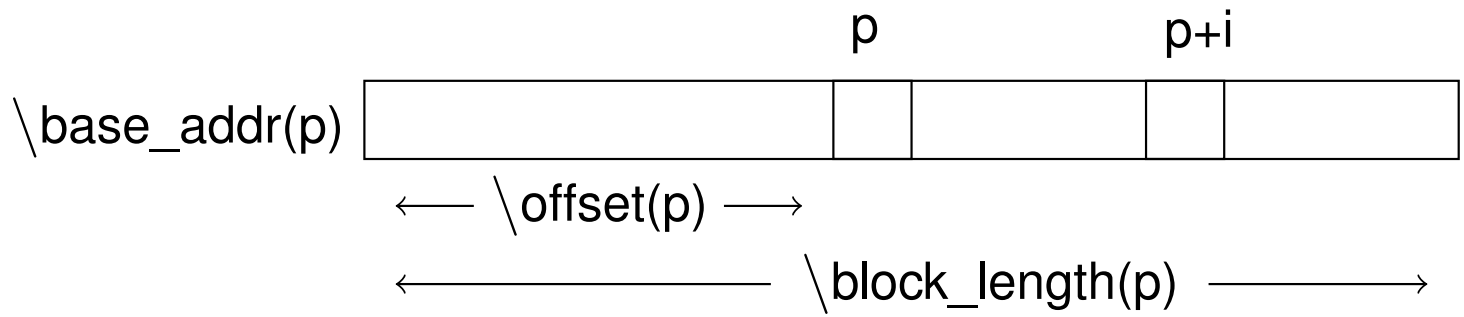
- lois algébriques de composition « read after write » :

$$\text{read}\ (\text{write}\ m\ p\ v)\ p = v$$
$$\text{read}\ (\text{write}\ m\ p\ v)\ p' = \text{read m p'} \qquad \text{si } p \neq p'$$

- La mémoire est vue comme un grand tableau d'octets (typés)

Représentation des pointeurs

- un pointeur permet d'accéder à un bloc mémoire de plusieurs octets
- pointeur = adresse de base + offset



- `\base_addr`, `\offset` et `\block_length` sont des fonctions logiques ACSL
- tableaux et arithmétique de pointeurs possibles

$$\begin{array}{rcll} e & ::= & \ldots & \text{mêmes expressions qu'avant} \\ & | & \star e & \text{accès à une valeur pointée} \end{array}$$

- présence d'un état mémoire global `mem: memory`
- rappel de logique de Hoare, règle d'affectation d'une variable :

$$\overline{\{P[x \mapsto e]\}\, x := e\, \{P\}}$$

- lors des substitutions, remplacer dans e les occurrences de $\star p$ par read $mem\ p$ (modélisation de la lecture d'une valeur pointée).
- règle d'affectation d'une valeur pointée (modélisation de l'écriture d'une valeur pointée) :

$$\overline{\{P[mem \mapsto \text{write}\ mem\ p\ e]\}\, \star p := e\, \{P\}}$$

Modèles mémoires simples : synthèse

- dans les premier cours :
  - variables du langage représentées par des variables logiques
  - pas de mémoire
  - aliasing pas exprimable
  - pas de pointeurs
  - obligations de preuve faciles à prouver
  - correspond au modèle « Hoare » du WP de Frama-C

Modèles mémoires simples : synthèse

- dans les premier cours :
  - variables du langage représentées par des variables logiques
  - pas de mémoire
  - aliasing pas exprimable
  - pas de pointeurs
  - obligations de preuve faciles à prouver
  - correspond au modèle « Hoare » du WP de Frama-C

- le début de ce cours :
  - une mémoire globale vue comme un grand tableau d'octets
  - pointeurs typés
  - aliasing entre pointeurs
  - problème :
    - dès qu'on modifie $\star\, p$, si on veut garantir que la valeur de $\star\, p'$ reste inchangé, il faut démontrer $p \neq p'$
    - très lourd en pratique
    - preuves souvent compliquées, non automatiques

Vers des modèles mémoires plus réalistes

En pratique (langage C) :

- adresse d'une variable
- types de données complexes (e.g. tableaux et structures)
- arithmétique de pointeurs
- allocations / dé-allocations
- casts
- opérations bit à bit, ...

Vers des modèles mémoires plus réalistes

En pratique (langage C) :

- adresse d'une variable
- types de données complexes (e.g. tableaux et structures)
- arithmétique de pointeurs
- allocations / dé-allocations
- casts
- opérations bit à bit, ...

Quel(s) modèle(s) mémoire(s) choisir ?

- aucun n'est parfait
- compromis entre expressivité et facilité des preuves

Représentation d'un état mémoire

- les variables globales (notamment) sont allouées dans des blocs distincts. On peut ainsi effectuer une hypothèse de séparation :

$$
\begin{array}{l}
\forall \text{ variables globales } G_1, G_2, \\
\forall i, j, \\
\quad 0 \leq i < \backslash block\_length(\&G_1) \implies \\
\quad 0 \leq j < \backslash block\_length(\&G_2) \implies \\
\quad \backslash base\_addr(\&G_1) + i \neq \backslash base\_addr(\&G_2) + j
\end{array}
$$

- une mémoire n'est plus juste un tableau d'octets mais un ensemble de tableaux indexés par les adresses de base des variables globales
- propriété d'isolation : modifier un sous-tableau laisse inchangés les autres, i.e., modifier un offset de G_1 ne modifie pas ceux de G_2
- preuves facilitées

Quelques optimisations possibles

- modèle de Burstall-Bornat
  - les champs de structures (les attributs des objets en POO) sont aussi séparés 2 à 2
  - facilite les raisonnements sur les structures et les objets
- modèle « Typed » du WP de Frama-C
  - jusqu'à présent, la mémoire est typée mais elle contient des valeurs de différentes sortes (des entiers, des flottants, des pointeurs)
  - il faut donc faire des conversions lors de lecture/écriture mémoire
  - problème : ces conversions peuvent gêner les prouveurs
  - solution : une mémoire par sorte
- modèle « Typed+ref » du WP de Frama-C
  - si une fonction *f* accède toujours à un pointeur *p* via $*$p, *f* ne crée pas d'alias vers *p*.
  - si on suppose que *p* n'est pas aliasé lors des appels à *f*, alors il est correct d'utiliser le modèle « Hoare » pour ces pointeurs
  - c'est ce que fait automatiquement Typed+ref
  - en C, de tels pointeurs sont notamment utilisés pour mimer le passage par référence et sont donc très fréquents

Exercice 1

1. Montrer que le triplet de Hoare suivant est prouvable (en utilisant le premier modèle du cours)

$$\{\text{read } mem\ p = 0\}\ \star p := \star p + 1\ \{\text{read } mem\ p = 1\}$$

2. définir une loi algébrique caractérisant la propriété d'isolation des adresses de base, i.e. que, si p et q sont 2 pointeurs ayant des adresses de base différentes, alors écrire dans p ne modifie pas $\star\ q$.

Exercice 1

Correction

1. Soit $I \equiv \star p := \star p + 1$ et $m \equiv mem$

$$\dfrac{\dfrac{\dfrac{}{\{\text{read}\ (\text{write}\ m\ p\ (\text{read}\ m\ p + 1))\ p = 1\}\ I\ \{\text{read}\ m\ p = 1\}}\ (3)}{\{\text{read}\ m\ p + 1 = 1\}\ I\ \{\text{read}\ m\ p = 1\}}\ (2)}{\{\text{read}\ m\ p = 0\}\ I\ \{\text{read}\ m\ p = 1\}}\ (1)$$

   (1) par application de la règle de conséquence (car notamment $\models \text{read}\ m\ p + 1 = 1 \rightarrow \text{read}\ m\ p = 0$)
   (2) par application de la règle de conséquence avec la loi $\text{read}\ (\text{write}\ m\ p\ v)\ p = v$
   (3) par application de la règle d'affectation d'une valeur pointée

2. $\text{read}\ (\text{write}\ m\ p\ v)\ q = \text{read}\ m\ q$ si $\backslash base\_addr(p) \neq \backslash base\_addr(q)$

4. Modèles et le WP de Frama-C

Les modèles du WP de Frama-C

- calcul WP générique par rapport aux modèles
- modèles mémoires du moins expressif au plus expressif :
  - Hoare
  - Typed+ref
  - Typed (par défaut)
- modèles arithmétiques :
  - Natural (par défaut) : opérations entières (hors casts) sont les opérations mathématiques habituelles. L'absence d'overflow doit être vérifié par ailleurs.
  - Machine : opérations C sur des entiers calculées modulo la taille du type. Preuves plus compliquées.
  - Real (par défaut) : opérations flottantes transformées en opérations réelles, sans arrondi. Incorrect par rapport aux normes C et IEEE.
  - Float : opérations flottantes définies comme des opérations mathématiques avec arrondi. Correct par rapport à la norme IEEE. Pas de preuves automatiques (sauf dans certains cas aves Gappa)

Combiner les modèles avec WP

- chaque propriété peut être prouvée avec un modèle différent
- attention aux hypothèses de chacun à vérifier par ailleurs
- par exemple, hypothèse de non-aliasing pour Typed+ref
- utiliser le greffon RTE pour garantir l'absence d'erreurs à l'exécution
- choix semi-automatique du modèle par le WP (choix automatique d'un modèle plus simple si possible)
- le noyau de Frama-C fait une synthèse des propriétés prouvées avec quels greffons/modèles

ACSL : fonctions et prédicats pour la mémoire

- constructions prédéfinies pour exprimer facilement des propriétés sur les pointeurs
- fonctions logiques (rappel) :
  - `\base_addr(p)` : addresse de base d'un pointeur
  - `\block_length(p)` : longueur du bloc associé au pointeur (en octets)
  - `\offset(p)` : décalage d'un pointeur par rapport à sa base (en octets)
  - ...
- prédicats logiques :
  - `\valid(p)` : *p* est-il un pointeur valide ?
  - `\initialized(p)` : la valeur pointée par *p* a-t-elle été initialisée ?
  - `\separated(p, q)` : les pointeurs *p* et *q* pointent vers 2 blocs mémoires différents.
  - ...
- aucun modèle mémoire imposé par ACSL
- son choix est laissé aux outils (e.g. WP)

Valeur déportée

- il est souvent utile de parler d'une valeur d'un terme t ou d'un prédicat P en un point de programme différent du point courant
- `\at(t,L)` et `\at(P,L)` où L est un label du programme ou un label prédéfini
- labels prédéfinis
  - `Here` : point de programme courant
  - `Pre` (resp. `Post`) : point de programme juste avant (resp. après) une fonction. Ces points sont appelés pre-state et post-state.
  - `Old` : désigne le pré-state dans un `ensures` ou un `assigns`
  - `LoopEntry` (resp. `LoopCurrent`) : dans une boucle, point de programme juste avant d'entrer dans la boucle (resp. ce tour de boucle)
- `\old(t)` = `\at(t,Old)`

Exercice 2

1. Spécifier et prouver la fonction suivante (fichier `code/swap.c`) :

```
/* swap the contents of [a] and [b]. */
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Exercice 3

1. Spécifier la fonction suivante (fichier `code/reset_swap.c`) :

```
/* put *a into *b, and reset *a to 0. */
void reset_swap(int *a, int *b) {
  int tmp = *a;
  *a = 0;
  *b = tmp;
}
```

2. Prouver ce contrat et les annotations de la fonction `main` de ce fichier avec le modèle Typed+ref
3. Même question avec le modèle Typed
4. Expliquer les résultats observés. Vous pouvez éventuellement vous aider des résultats fournis par la ligne de commandes suivante :

```
$ frama-c -wp -wp-model Typed+ref -wp-print-separation \
    reset_swap.c
```

Exercice 4

1. Spécifier et prouver la fonction suivante (fichier `code/find_min.c`) :

```
/* returns the index of the minimal element
   of the given array [a] of size [length] */
int find_min(int* a, int length) {
  int min, min_idx;
  min_idx = 0;
  min = a[0];
  for (int i = 1; i < length; i++) {
    if (a[i] < min) {
      min_idx = i;
      min = a[i];
    }
  }
  return min_idx;
}
```

Ensembles

- ensembles : citoyens de 1ère classe en ACSL
- ensembles homogènes : tous les éléments de l'ensemble doivent être du même type (éventuellement en convertissant des pointeurs vers des types différents en `char *`)
- souvent utile de faire référence à une place de zones mémoires
- $x = \{x\}$
- $a..b = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$
- $t_1 + t_2 = \{x + y \mid x \in t_1, y \in t_2\}$
- \initialized(p+(0..9)) : les valeurs $\star(p+i)$ avec $i \in [0..9]$ sont initialisées.
- ...
- simplifie les spécifications

Exercice 5

1. Modifier le contrat de la fonction `find_min` de l'exercice précédant pour utiliser une notation ensembliste.

2. Spécifier et prouver la fonction suivante (fichier `code/parity_reset.c`) :

```c
/* put 0 in each cell of [t] with an even index
   and 1 in the other cells.
   [n] is the length of [t]. */
void parity_reset(int *t, int n) {
  int i;
  for(i = 0; i < n; i++)
    t[i] = i % 2;
}
```