Invariants de classe et systèmes d'ownership

Romain Bardou

Stage de Master à l'INRIA Futurs, avec Claude Marché

Mars – Juillet 2007

Plan

1. Contexte, invariants de classe
2. Méthodologie à la Boogie (Spec#, Microsoft Research)
3. Adapter à Jessie (INRIA Futurs)
4. Autres contributions
5. Conclusion

Plan

1. Contexte, invariants de classe
2. Méthodologie à la Boogie
3. Adapter aux mémoires séparées
4. Autres contributions
5. Conclusion

Contexte (1)

Spécification et preuve de programmes impératifs.

```
/*@ ensures
  @   \result >= x &&
  @   \result >= y &&
  @   \forall int z; z >= x && z >= y => z >= \result
  @*/
int max(int x, int y) {
  if (x > y) return x; else return y;
}
```

Contexte (2)

Méthode de preuve : génération d'*obligations*.

Programme annoté

$\Downarrow$

VCG

$\Downarrow$

Obligations de preuve

$\Downarrow$

Outil de démonstration automatique (Simplify, Ergo...)

Assistant de preuve (Coq, Isabelle...)

Invariants : exemples (1)

Classes Java.

```
class Purse {
  int euros;
  int cents;
  invariant purse_inv(p) =
    p.cents >= 0 &&
    p.cents <= 99;
}
```

Invariants : exemples (2)

Structures C.

```
struct Purse {
  int euros;
  int cents;
  invariant purse_inv(p) =
    p.cents >= 0 &&
    p.cents <= 99;
}
```

Invariants : exemples (3)

Tableaux, composants.

```
struct Bank {
  Purse[] clients;
  int count;
  invariant bank_allocation(x) =
    \forall i, 0 <= i && i <= x.count-1 =>
      \valid(x.clients[i]);
}
```

Invariants : difficultés (1)

Sémantique non immédiate.

```
void add(Purse source, Purse dest)
{
  dest.euros += source.euros;
  dest.cents += source.cents;
  // invariant violé
  if (dest.cents >= 100) { dest.cents -= 100; dest.euros++; }
  // invariant restauré
}
```

Invariants : difficultés (2)

Propriété d'une structure de données vraie "à tout moment".

"à tout moment" :
- en tout point de programme ?
- hors des méthodes de la structure ?
- spécifié par le programmeur ?
- ...

Invariants : difficultés (3)

Sémantique JML :

$$\text{invariant} \iff \begin{array}{l}\text{pré-condition et post-condition}\\ \text{pour chaque objet “accessible”}\end{array}$$

“objet accessible” : paramètres, leurs champs...

Ensemble d'objets difficilement exprimable.

Sémantique mal adaptée à la preuve.

Invariants : problématique

Outil pour spécifier les programmes.

Trouver une bonne définition / sémantique.

Fournir de quoi les utiliser dans les preuves.

Plan

1. Contexte, invariants de classe
2. Méthodologie à la Boogie
3. Adapter aux mémoires séparées
4. Autres contributions
5. Conclusion

Boogie

Spec# (C# annoté)

$\Downarrow$

Boogie

$\Downarrow$

Obligations de preuve

Boîtes

Sémantique d'invariants basée sur une notion d'*ownership*.

Boîtes et invariants

$$\text{Objet fermé} \Rightarrow \begin{cases} \text{Invariant vérifié} \\ \text{Non modifiable} \end{cases}$$

$$\text{Objet ouvert} \Rightarrow \begin{cases} \text{Invariant : ?} \\ \text{Modifiable} \end{cases}$$

Ownership

Une boîte peut contenir d'autres boîtes.

Une boîte fermée ne contient que des boîtes fermées.

Ouvrir ou fermer une boîte

On peut ouvrir une boîte fermée.

Pour fermer une boîte, on vérifie :
– son invariant, et
– qu'elle ne contient pas de boîte ouverte.

Méthodologie de Boogie

Deux champs pour décrire l'état d'un objet.

- `inv` : l'objet est-il fermé ?
- `committed` : l'objet est-il dans une boîte fermée ?

Ne pas modifier un objet x si $x.\texttt{inv} = \texttt{true}$.

Décrire l'état de ces champs pour les paramètres avant et après l'appel d'une fonction.

Exemple

```
void add(Purse source, Purse dest)


{

  dest.euros += source.euros;
  dest.cents += source.cents;
  if (dest.cents >= 100) { dest.cents -= 100; dest.euros++; }

}
```

Exemple

```
void add(Purse source, Purse dest)
  requires source.inv && dest.inv;

{

  dest.euros += source.euros;
  dest.cents += source.cents;
  if (dest.cents >= 100) { dest.cents -= 100; dest.euros++; }

}
```

Exemple

```
void add(Purse source, Purse dest)
  requires source.inv && dest.inv;

{
  unpack(dest);
  dest.euros += source.euros;
  dest.cents += source.cents;
  if (dest.cents >= 100) { dest.cents -= 100; dest.euros++; }
  pack(dest); // prouver l'invariant de dest
}
```

Exemple

```
void add(Purse source, Purse dest)
  requires source.inv && dest.inv && not dest.committed;

{
  unpack(dest);
  dest.euros += source.euros;
  dest.cents += source.cents;
  if (dest.cents >= 100) { dest.cents -= 100; dest.euros++; }
  pack(dest); // prouver l'invariant de dest
}
```

Exemple

```
void add(Purse source, Purse dest)
  requires source.inv && dest.inv && not dest.committed;
  ensures source.inv && dest.inv && not dest.committed;
{
  unpack(dest);
  dest.euros += source.euros;
  dest.cents += source.cents;
  if (dest.cents >= 100) { dest.cents -= 100; dest.euros++; }
  pack(dest); // prouver l'invariant de dest
}
```

Résultat principal

Supposons prouvées les obligations de preuve générées par la méthodologie.

Pour tout type T,

$$\forall x : T,\ x.\texttt{inv} \Rightarrow Inv_T(x)$$

Valide en tout point de programme.

Pour un porte-monnaie :

$$\forall x : \texttt{Purse},\ x.\texttt{inv} \Rightarrow 0 \leq x.\texttt{cents} \leq 99$$

Relie `inv` et `cents`.

Plan

1. Contexte, invariants de classe
2. Méthodologie à la Boogie
3. Adapter aux mémoires séparées
4. Autres contributions
5. Conclusion

Jessie

C, Java annoté

$\Downarrow$

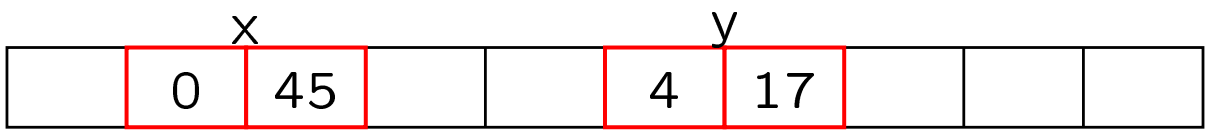
Jessie / Why

$\Downarrow$

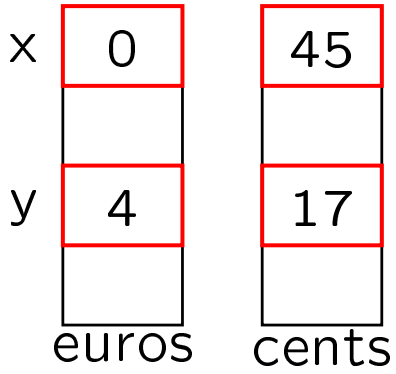
Obligations de preuve

Jessie : mémoires séparées

Modèle physique :



Modèle de Jessie :



Utiliser la propriété comme axiome ?

Quantifier sur les mémoires :

$$\forall x : \texttt{Purse}, \forall \texttt{inv}, \texttt{cents},\ x.\texttt{inv} \Rightarrow 0 \leq x.\texttt{cents} \leq 99$$

est incohérent.

Avec `inv` tel que $x.\texttt{inv} = \texttt{true}$:

$$\forall \texttt{cents},\ 0 \leq x.\texttt{cents} \leq 99$$

Ma solution : relier les mémoires

$assoc(p, m) \iff$ la mémoire m est reliée au point de programme p

L'axiome redevient cohérent :

$$\forall p, x, \texttt{inv}, \texttt{cents},\ assoc(p, \texttt{inv}) \wedge assoc(p, \texttt{cents}) \Rightarrow x.\texttt{inv} \Rightarrow x.\texttt{cents} \geq 0$$

Génération des $assoc$

On génère des instances du prédicat $assoc$.

- Au début d'une fonction.
- A la modification d'une mémoire `m`.
  - Uniquement si `m` apparait dans un invariant.
  - Uniquement pour les mémoires apparaissant dans ces invariants.

Se passer de $assoc$

Au lieu de générer des instances de $assoc(p, \cdots)$, instancier l'axiome des invariants en p.

Exemple : si on modifie $x.\texttt{cents}$, au lieu de générer :

$$assoc(p, \texttt{inv}) \wedge assoc(p, \texttt{cents})$$

on génère, au point de programme p, l'hypothèse :

$$\forall x : \texttt{Purse},\ x.\texttt{inv} \Rightarrow 0 \leq x.\texttt{cents} \leq 99$$

Propriété

Sous les hypothèses générées comme précédemment, si :
– on ne modifie pas une boîte fermée,
– on n'ouvre pas une boîte possédée,
– on prouve l'invariant d'une boîte quand on la ferme,
alors pour tout type T,

$$\forall x : T,\ x.\texttt{inv} \Rightarrow Inv_T(x)$$

En tout point de programme.

De plus, inductivement, les hypothèses générées sont valides.

Plan

1. Contexte, invariants de classe
2. Méthodologie à la Boogie
3. Adapter aux mémoires séparées
4. Autres contributions
5. Conclusion

Arithmétique de pointeurs

– Indispensable en C
– Tableaux en Java

Modifier les opérations d'ouverture et de fermeture : prendre en compte les *décalages* des champs (plus d'objets possédés).

Ajouter des opérations pour ouvrir ou fermer une rangée de boîtes.

```
pack_block(s.data, 0, s.size-1);
```

Modèle mémoire de Jessie réalisé en Coq

Modèle mémoire de Jessie en Why :
- déclarations de prédicats, de fonctions...,
- axiomes décrivant leur comportement.

Réalisation en Coq :
- définir les prédicats, les fonctions...,
- prouver les axiomes à partir de ces définitions.

Assure la cohérence du modèle mémoire.

Plan

1. Contexte, invariants de classe
2. Méthodologie à la Boogie
3. Adapter aux mémoires séparées
4. Autres contributions
5. Conclusion

Conclusion

Solution aussi flexible et expressive que celle de Boogie, mais adaptée aux mémoires séparées.

Implémentée dans Jessie.

Autres contributions :
- extension aux tableaux,
- réalisation en Coq du modèle mémoire de Jessie,
- description des invariants à l'aide d'un prédicat inductif exprimable en Coq.

Limitations, travaux futurs

Structure arborescente des composants.
– Listes circulaires ?

Préciser l'état de `inv` et `committed` pour chaque fonction.
– Valeurs par défaut ?
– Inférence ?

Ownership : solution à un problème plus général : la preuve de propriétés de *séparation*.