

Preuve formelle de micro-noyau

A. Stalain

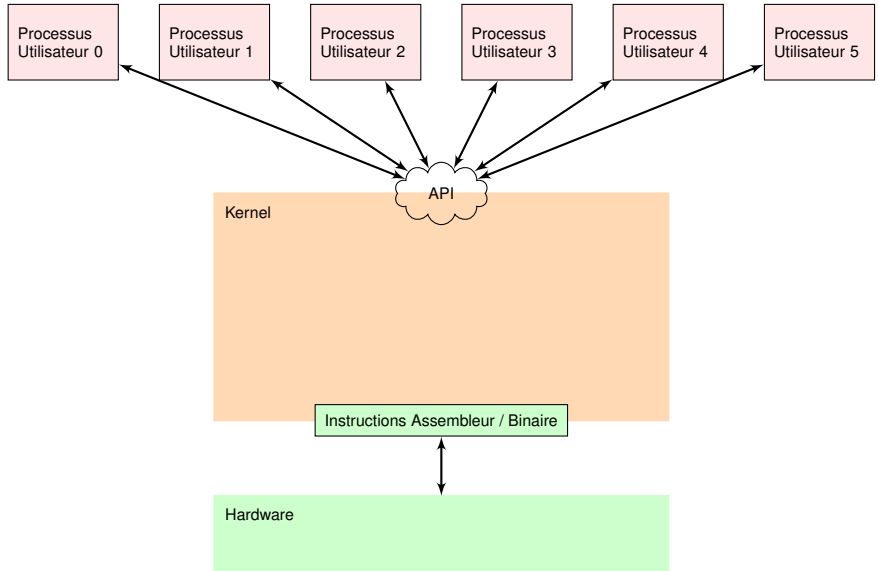
Orange Labs - Stagiaire

Tuteur - V. Sanchez Leighton

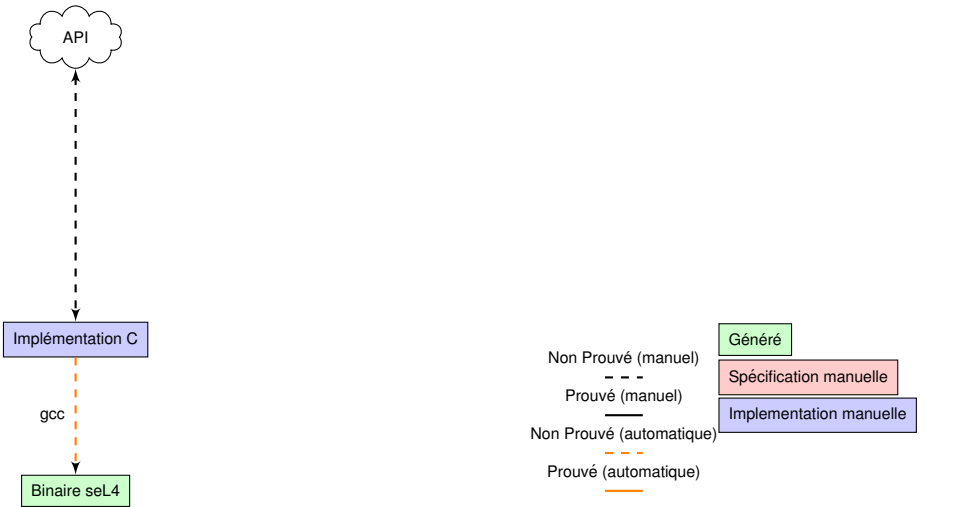
May 13, 2019

SeL4

- Micro-noyau, de la famille de L4
 - Minimaliste
- Sécurisé
- Performant
- Vérifié formellement



La preuve de SeL4



API



La spécification informelle du fonctionnement du système.

Exemple

La fonction `list_insert_front` doit insérer un élément en tête de liste chaînée.

Implémentation C

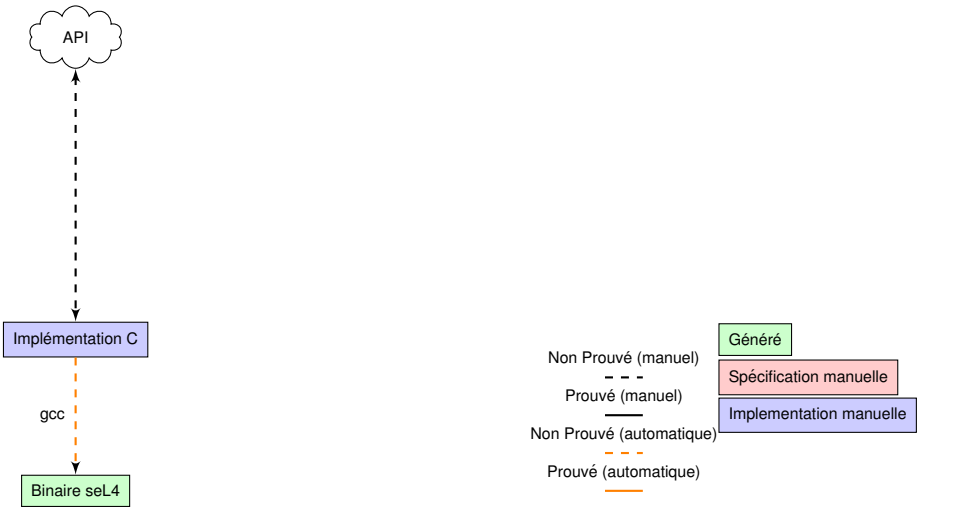
Implémentation C

Le Code source C

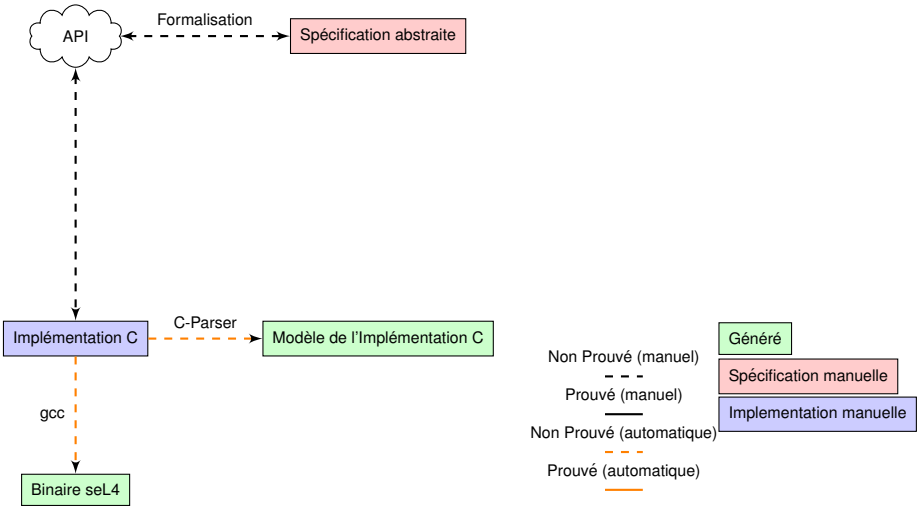
Exemple

```
typedef struct list *list_t;  
void list_insert_front(list_t *l, struct list *x)  
{  
    x->next = *l;  
    *l = x;  
}
```

La preuve de SeL4



La preuve de SeL4



Spécification abstraite

Spécification abstraite

Une représentation formelle de ce qu'on attends de l'API

Exemple

```
lemma list_insert_front_correct :  
  "new  $\neq$  NULL  $\Rightarrow$   
   { $\lambda$ s. listp xs l s  $\wedge$  is_valid_list_C s new}  
   list_insert_front l new  
  { $\lambda$ _. listp (new#xs) l}!"
```

Modèle de l'implémentation C

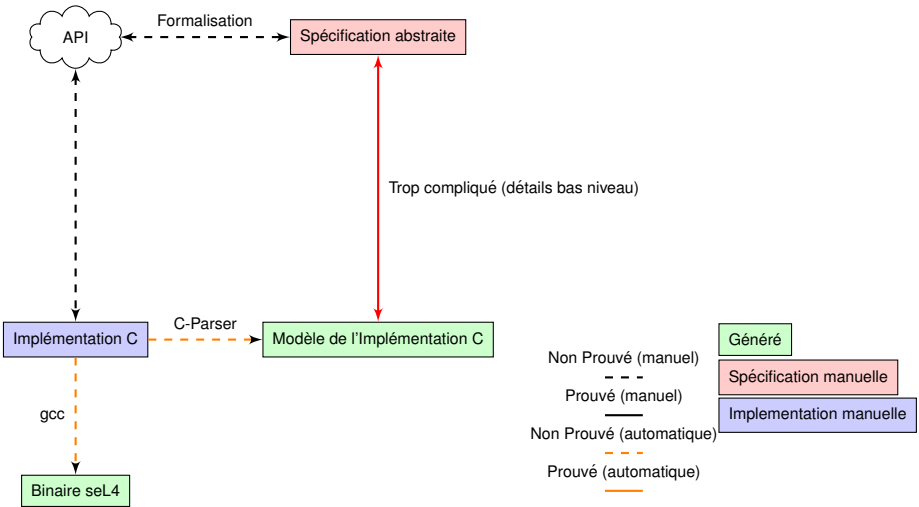
Modèle du C

Une représentation formelle du comportement de la fonction.

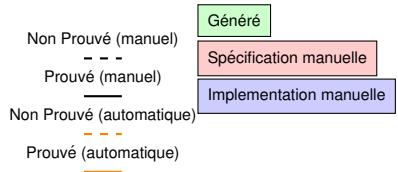
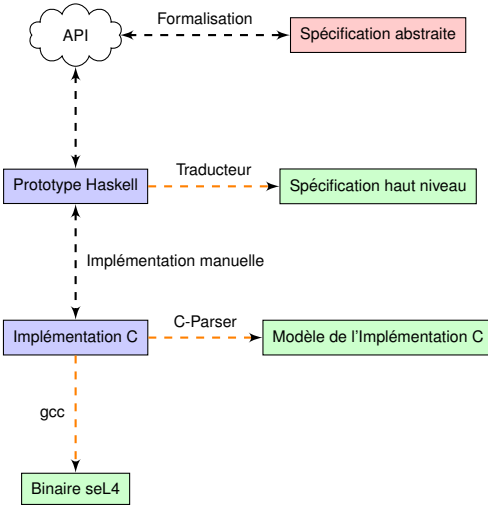
Exemple

```
"all_global_addresses.list_insert_front_body ≡  
TRY  
  Guard C_Guard {[c_guard 'new]}  
  (Guard C_Guard {[c_guard 'l]}  
    ('globals := t_hrs'_update (hrs_mem_update (  
      heap_update (Ptr &('new→['next_C'])) (h_val (  
        hrs_mem 't_hrs) 'l)))));;  
  Guard C_Guard {[c_guard 'l]} ('globals := t_hrs'_update (  
    hrs_mem_update (heap_update 'l 'new)))  
CATCH SKIP  
END"
```

La preuve de SeL4



La preuve de SeL4



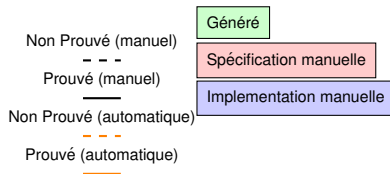
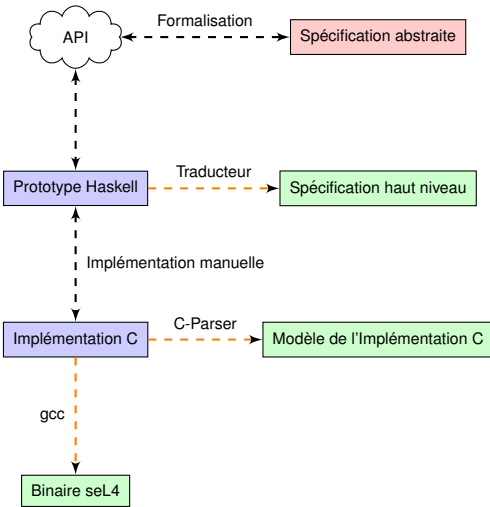
Prototype Haskell

Prototype Haskell

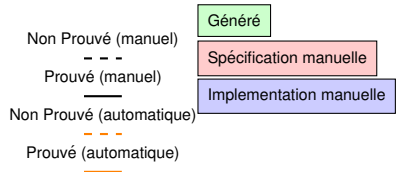
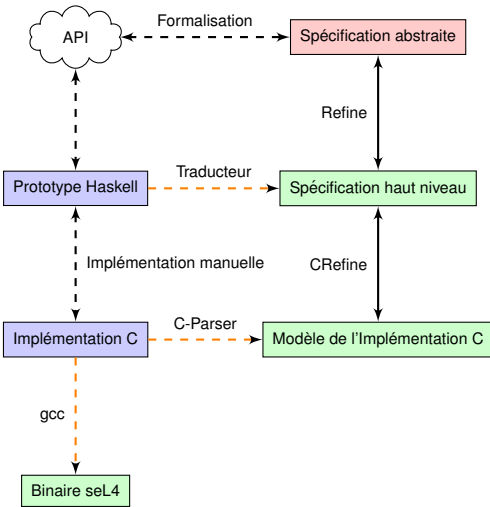
Exemple

```
insert_front :: PPtr ListT -> PPtr List -> Kernel ()  
insert_front l new = do  
    mlist <- getListPtr l  
    updateNext new mlist  
    updateListPtr mlist new
```

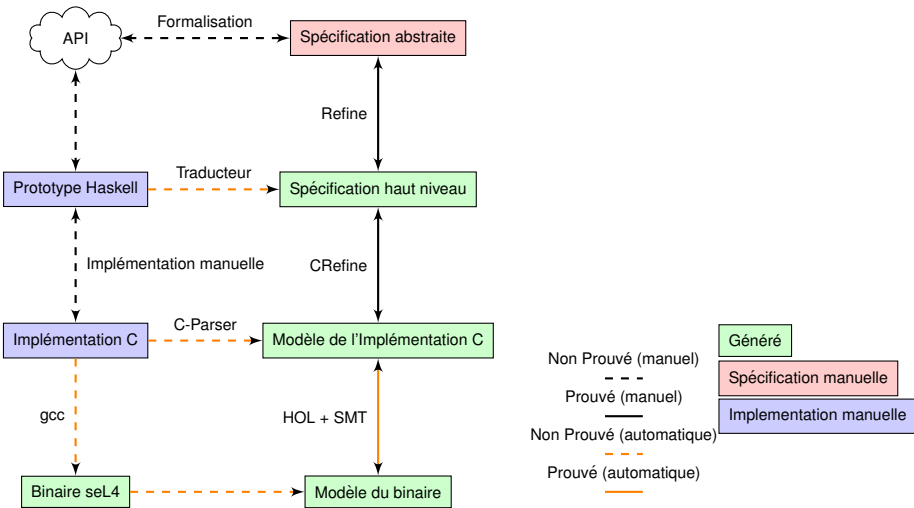
La preuve de SeL4



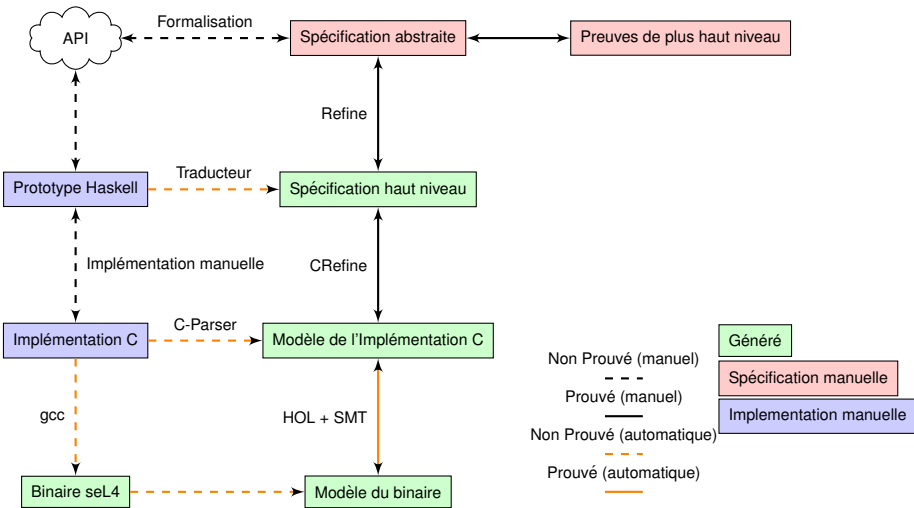
La preuve de SeL4



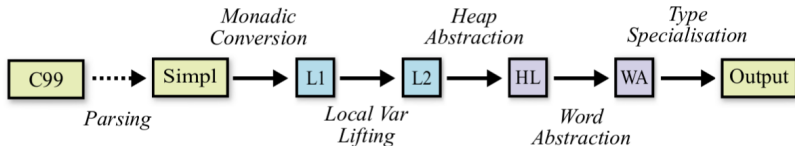
La preuve de SeL4



La preuve de SeL4



Autocorres



Don't Sweat the Small Stuff, Formal Verification of C Code Without Pain

– David Greenaway, Japheth Lim, June Andronick, Gerwin Klein

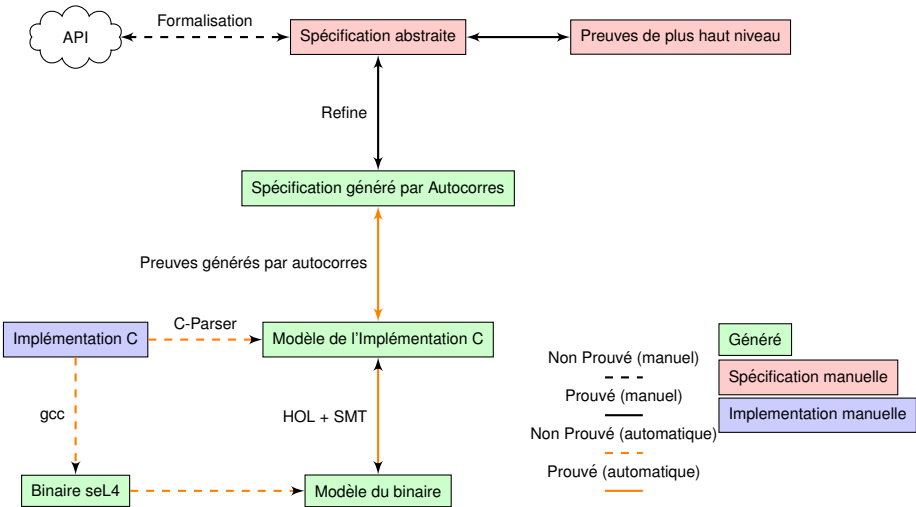
- Outil d'abstraction automatique C vers Isabelle
- Génère une preuve de correspondance
- Développé par le groupe qui développe SeL4

```
void list_insert_front
```

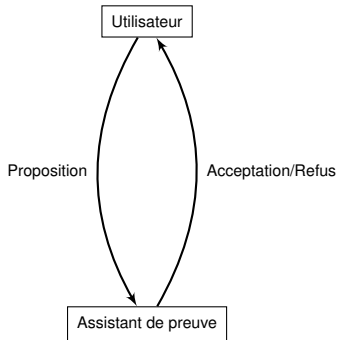
```
(list_t *l, struct list *new)
{
  new->next = *l;
  *l = new;
}
```

```
"all.list_insert_front' ?l ?new ≡
do guard(λs. is_valid_list_C s ?new);
guard(λs. is_valid_list_C'ptr s ?l);
modify(λs. s[?new->next := s[?l]]);
modify(λs. s[?l := ?new])
od"
```

La preuve de SeL4



Assistant de preuves



- Logiciel qui aide l'interaction entre le noyau du prouveur et l'utilisateur
- Permet de résoudre les preuves pas à pas
- Isabelle - Coq - Mizar

TRIPLET DE HOARE

```
"{ λs. (x mod 2) = 0 }  
  do  
    return (x >>1)  
  od  
  { λr s. r*2 = x }"
```

```
/* ASSERT : (x % 2) == 0 */  
return x >> 1;  
/* ASSERT : (ret_value * 2) == x */
```

Expression vraie quand pour tout état s dans lequel
précondition est valide, après l'exécution de
instructions dans cet état, alors **postcondition** sera
valide dans le nouvel état.

Exemple: liste chaînée

```
typedef void *data_ptr;

struct list
{
    struct list *next;
    data_ptr data;
};

typedef struct list *list_t;
```

Autocorres avec la syntaxe Lifted heap

C	Isabelle
struct list a;	$a :: \text{list_C}$
struct list *a;	$a :: \text{list_C ptr}$
a.next	$a \rightarrow \text{next} :: \text{list_C ptr}$
	$s :: \text{lifted_globals}$
*a	$s[a] :: \text{list_C}$
a->next	$s[a] \rightarrow \text{next} :: \text{list_C ptr}$
*a = x	$s[a := x] :: \text{lifted_globals}$

Preuve de list_empty

```
void list_empty(list_t *l)
{
    *l = NULL;
}
```


Spécification informelle

On veut: Pour l un pointeur (valide) sur pointeur sur node, après l'appel de `list_empty l`, l est un pointeur sur liste valide vide.

```
void list_empty(list_t *l)
{
    *l = NULL;
}
```

Spécification informelle

On veut: Pour l un pointeur (valide) sur pointeur sur node, après l'appel de `list_empty l`, l est un pointeur sur liste valide vide.

```
void list_empty(list_t *l)
{
    *l = NULL;
}
```

Une liste valide est une liste dont:

Spécification informelle

On veut: Pour l un pointeur (valide) sur pointeur sur node, après l'appel de `list_empty l`, l est un pointeur sur liste valide vide.

```
void list_empty(list_t *l)
{
    *l = NULL;
}
```

Une liste valide est une liste dont:

- Toutes les nodes sont des zones mémoire valides.

Spécification informelle

On veut: Pour l un pointeur (valide) sur pointeur sur node, après l'appel de `list_empty l`, l est un pointeur sur liste valide vide.

```
void list_empty(list_t *l)
{
    *l = NULL;
}
```

Une liste valide est une liste dont:

- Toutes les nodes sont des zones mémoire valides.
- Soit un pointeur NULL, dans ce cas là, la liste est vide.
- Soit un pointeur sur une node, dans ce cas là, la liste est composée de cette node et de la liste dans le champ `next` de la node.

Définition formelle du concept de liste chaînée

On définit la fonction `list` une fonction pour tester l'équivalence entre une liste Isabelle et une liste C:

```
type_synonym node = "list_C ptr"  
primrec list :: "node list  $\Rightarrow$  node  $\Rightarrow$  lifted_globals  $\Rightarrow$  bool"  
  where
```

Définition formelle du concept de liste chaînée

On définit la fonction `list` une fonction pour tester l'équivalence entre une liste Isabelle et une liste C:

```
type_synonym node = "list_C ptr"  
primrec list :: "node list  $\Rightarrow$  node  $\Rightarrow$  lifted_globals  $\Rightarrow$  bool"  
  where  
list_is_empty: "list [] p s = (p = NULL) "
```

Définition formelle du concept de liste chaînée

On définit la fonction `list` une fonction pour tester l'équivalence entre une liste Isabelle et une liste C:

```

type_synonym node = "list_C ptr"
primrec list :: "node list  $\Rightarrow$  node  $\Rightarrow$  lifted_globals  $\Rightarrow$  bool"
  where
list_is_empty:  "list [] p s = (p = NULL)" |
list_is_cons:  "list (x#xs) p s = ( p = x
                                    $\wedge$  is_valid_list_C s x
                                    $\wedge$  x  $\neq$  NULL
                                    $\wedge$  list xs s[x] $\rightarrow$ next s
                                   )"

```

On définit `listp x pt s` comme une expression qui est vraie quand le pointeur sur liste `struct list **pt` contient exactement les nodes de `x` dans l'état global `s`.

```
definition listp :: "node list  $\Rightarrow$  node ptr  $\Rightarrow$  lifted_globals  $\Rightarrow$   
  bool" where  
  "listp x pt s  $\equiv$  (list x s[pt] s  
     $\wedge$  ptr_coerce pt  $\notin$  set x  
     $\wedge$  is_valid_list_C'ptr s pt) "
```



```
lemma list_empty_correct :  
  "{ λs. is_valid_list_C'ptr s l }  
    all.list_empty' l  
  { λ_. listp [] l }!"
```

On prouve que:

voir (proof/Listp.thy)

```
lemma list_empty_correct :  
  "{ λs. is_valid_list_C'ptr s l }  
    all.list_empty' l  
  { λ_. listp [] l }!"
```

On prouve que:

- Si `l` est un `struct list ** valide`.

voir (proof/Listp.thy)

```
lemma list_empty_correct :  
  "{ λs. is_valid_list_C'ptr s l }  
    all.list_empty' l  
  { λ_. listp [] l }!"
```

On prouve que:

- Si `l` est un `struct list ** valide`.
- Alors, après l'exécution de `list_empty l`.

voir (proof/Listp.thy)

```
lemma list_empty_correct :  
  "{ λs. is_valid_list_C'ptr s l }  
    all.list_empty' l  
  { λ_. listp [] l }!"
```

On prouve que:

- Si `l` est un `struct list ** valide`.
- Alors, après l'exécution de `list_empty l`.
- On aura `l` un pointeur sur liste valide, contenant aucun élément.

voir (proof/Listp.thy)

```
lemma list_empty_correct :  
  "{ λs. is_valid_list_C'ptr s l }  
    all.list_empty' l  
  { λ_. listp [] l }!"
```

On prouve que:

- Si `l` est un `struct list ** valide`.
- Alors, après l'exécution de `list_empty l`.
- On aura `l` un pointeur sur liste valide, contenant aucun élément.
- Sans que d'erreur ne se produise, et en un temps fini.

voir (proof/Listp.thy)

TRADUCTION PAR AUTOCORRES

C

```
void list_empty(list_t *l)
{
    *l = NULL;
}
```

Isabelle

```
"all.list_empty' ?l  $\equiv$ 
  do guard ( $\lambda s. \text{is\_valid\_list\_C'ptr } s \text{ ?l}$ );
    modify ( $\lambda s. s[?l := \text{NULL}]$ )
od"
```

Démonstration: preuve de `list_empty`

```
void list_empty(list_t *l)
{
  *l = NULL;
}
```

list_empty_correct ✓
list_empty_pure ✓

```
void list_empty_alt1(list_t *l)
{
  *l = NULL;
  global_variable = (int)l;
}
```

list_empty_alt1_correct ✓
list_empty_alt1_pure ✗

Preuves trouvables dans `proof/Listp.thy`

Preuve de list_singleton

```
void list_singleton(list_t *l, struct list *x)
{
    *l = x;
}
```

Spécification informelle

On veut: Pour l un pointeur sur une liste valide, après l'appel de la fonction `list_singleton l x`, l pointera sur une liste avec comme seul élément x .

```
void list_singleton(list_t *l, struct list *x)
{
    *l = x;
}
```

Spécification formelle

```
lemma list_singleton_correct :  
  "[[ x ≠ NULL ; x ≠ ptr_coerce l ]] ⇒  
    {λs. is_valid_list_C s x ∧ is_valid_list_C'ptr s l}  
    all.list_singleton' l x  
    {λ_. listp [x] l }!"
```

On prouve que:

Spécification formelle

```

lemma list_singleton_correct :
  "[[ x ≠ NULL ; x ≠ ptr_coerce l ]] ⇒
  {λs. is_valid_list_C s x ∧ is_valid_list_C'ptr s l}
  all.list_singleton' l x
  {λ_. listp [x] l }!"

```

On prouve que:

- Pour tout x non-nul et $x \neq (\text{struct list } *)l$.

Spécification formelle

```
lemma list_singleton_correct :
  "[[ x ≠ NULL ; x ≠ ptr_coerce l ]] ⇒
  {λs. is_valid_list_C s x ∧ is_valid_list_C'ptr s l }
  all.list_singleton' l x
  {λ_. listp [x] l }!"
```

On prouve que:

- Pour tout x non-nul et $x \neq (\text{struct list } *)l$.
- Si l et x sont des pointeurs valides sur leur types respectifs.

Spécification formelle

```

lemma list_singleton_correct :
  "[[ x ≠ NULL ; x ≠ ptr_coerce l ]] ⇒
  {λs. is_valid_list_C s x ∧ is_valid_list_C'ptr s l}
  all.list_singleton' l x
  {λ_. listp [x] l }!"

```

On prouve que:

- Pour tout x non-nul et $x \neq (\text{struct list } *)l$.
- Si l et x sont des pointeurs valides sur leur types respectifs.
- Alors, après l'exécution de `list_singleton l x`.

Spécification formelle

```
lemma list_singleton_correct :  
  "[[ x ≠ NULL ; x ≠ ptr_coerce l ]] ⇒  
    {λs. is_valid_list_C s x ∧ is_valid_list_C'ptr s l}  
    all.list_singleton' l x  
    {λ_. listp [x] l }!"
```

On prouve que:

- Pour tout x non-nul et $x \neq (\text{struct list } *)l$.
- Si l et x sont des pointeurs valides sur leur types respectifs.
- Alors, après l'exécution de `list_singleton l x`.
- On aura l un pointeur sur liste valide, contenant uniquement x

Spécification formelle

```

lemma list_singleton_correct :
  "[[ x ≠ NULL ; x ≠ ptr_coerce l ]] ⇒
  {λs. is_valid_list_C s x ∧ is_valid_list_C'ptr s l}
  all.list_singleton' l x
  {λ_. listp [x] l }!"

```

On prouve que:

- Pour tout x non-nul et $x \neq (\text{struct list } *)l$.
- Si l et x sont des pointeurs valides sur leur types respectifs.
- Alors, après l'exécution de `list_singleton l x`.
- On aura l un pointeur sur liste valide, contenant uniquement x
- Sans que d'erreurs se produise, et en un temps fini.

Démonstration: preuve de `list_singleton`

Problème: on arrive pas a prouver que:

1. $\forall s. [\dots] \Rightarrow s[x] \rightarrow \text{next} = \text{NULL}$

Problème: on arrive pas à prouver que:

1. $\forall s. [\dots] \Rightarrow s[x] \rightarrow \text{next} = \text{NULL}$

C'est à dire:

- Que $x \rightarrow \text{next} == \text{NULL}$.

Problème: on arrive pas a prouver que:

1. $\forall s. [...] \Rightarrow s[x] \rightarrow \text{next} = \text{NULL}$

C'est à dire:

- Que $x \rightarrow \text{next} == \text{NULL}$.

Pourquoi:

- Si la proposition est fausse, x ne serait pas nécessairement le dernier élément puisque son $\rightarrow \text{next}$ est pas forcément NULL .
- C'est donc un bug dans la fonction.

Problème: on arrive pas a prouver que:

1. $\forall s. [...] \Rightarrow s[x] \rightarrow \text{next} = \text{NULL}$

C'est à dire:

- Que $x \rightarrow \text{next} == \text{NULL}$.

Pourquoi:

- Si la proposition est fausse, x ne serait pas nécessairement le dernier élément puisque son $\rightarrow \text{next}$ est pas forcément NULL .
- C'est donc un bug dans la fonction.

Solution:

- Corriger le code source.

On a donc:

```
void list_singleton(list_t *l, struct list *x)
{
    *l = x;
    x->next = NULL;
}
```

La preuve peut donc permettre de corriger des bugs.

voir ([proof/Listp.thy](#))

Preuve de list_insert_front

```
void list_insert_front(list_t *l, struct list *x)
{
    x->next = *l;
    *l = x;
}
```

Spécification informelle

On veut: Pour l un pointeur sur une liste valide, après l'appel de la fonction `list_insert_front l x`, l pointera sur la même liste mais avec x en tête.

```
void list_insert_front(list_t *l, struct list *x)
{
    x->next = *l;
    *l = x;
}
```


Spécification formelle

```
lemma list_insert_front_correct :  
  "x ≠ NULL ⇒  
    {λs. listp xs l s ∧ is_valid_list_C s x }  
    all.list_insert_front' l x  
    {λr. listp (x#xs) l }!"
```

On prouve que:

Spécification formelle

```
lemma list_insert_front_correct :  
  "x ≠ NULL ⇒  
    {λs. listp xs l s ∧ is_valid_list_C s x }  
    all.list_insert_front' l x  
    {λr. listp (x#xs) l }!"
```

On prouve que:

- Pour tout x non-nul.

Spécification formelle

```
lemma list_insert_front_correct :  
  "x ≠ NULL ⇒  
    {λs. listp xs l s ∧ is_valid_list_C s x }  
    all.list_insert_front' l x  
    {λr. listp (x#xs) l }!"
```

On prouve que:

- Pour tout x non-nul.
- Si l est un pointeur sur liste valide contenant les éléments xs et x est un struct `list` valide.

Spécification formelle

```

lemma list_insert_front_correct :
  "x ≠ NULL ⇒
    {λs. listp xs l s ∧ is_valid_list_C s x }
    all.list_insert_front'l x
    {λr. listp (x#xs) l }!"

```

On prouve que:

- Pour tout x non-nul.
- Si l est un pointeur sur liste valide contenant les éléments xs et x est un struct list * valide.
- Alors, après l'exécution de `list_insert_front l x`.

Spécification formelle

```

lemma list_insert_front_correct :
  "x ≠ NULL ⇒
    {λs. listp xs l s ∧ is_valid_list_C s x }
    all.list_insert_front' l x
    {λr. listp (x#xs) l }!"

```

On prouve que:

- Pour tout x non-nul.
- Si l est un pointeur sur liste valide contenant les éléments xs et x est un struct `list` * valide.
- Alors, après l'exécution de `list_insert_front l x`.
- On aura l un pointeur sur liste valide, contenant x suivi des éléments de xs

Spécification formelle

```

lemma list_insert_front_correct :
  "x ≠ NULL ⇒
    {λs. listp xs l s ∧ is_valid_list_C s x }
    all.list_insert_front' l x
    {λr. listp (x#xs) l }!"

```

On prouve que:

- Pour tout x non-nul.
- Si l est un pointeur sur liste valide contenant les éléments xs et x est un struct list * valide.
- Alors, après l'exécution de `list_insert_front l x`.
- On aura l un pointeur sur liste valide, contenant x suivi des éléments de xs
- Sans que d'erreurs se produise, et en un temps fini.

Démonstration: preuve de `list_insert_front`

Problème: on arrive pas a prouver que:

1. $\forall s. [...] \Rightarrow \text{ptr_coerce } l \neq x$
2. $\forall s. [...] \Rightarrow x \notin \text{set } xs$

Problème: on arrive pas à prouver que:

1. $\forall s. [...] \Rightarrow \text{ptr_coerce } l \neq x$
2. $\forall s. [...] \Rightarrow x \notin \text{set } xs$

C'est à dire:

- Que $(\text{struct list*}) l$ est différent de x .
- Et que x n'est pas déjà une node de la liste.

Problème: on arrive pas à prouver que:

1. $\forall s. [...] \Rightarrow \text{ptr_coerce } l \neq x$
2. $\forall s. [...] \Rightarrow x \notin \text{set } xs$

C'est à dire:

- Que $(\text{struct list*})l$ est différent de x .
- Et que x n'est pas déjà une node de la liste.

Pourquoi:

- Si la proposition 1 est vraie, on corrompraît la liste.
- Si la proposition 2 est vraie, on créerait un cycle dans la liste chaînée.

Problème: on arrive pas à prouver que:

1. $\forall s. [...] \Rightarrow \text{ptr_coerce } l \neq x$
2. $\forall s. [...] \Rightarrow x \notin \text{set } xs$

C'est à dire:

- Que $(\text{struct list*}) l$ est différent de x .
- Et que x n'est pas déjà une node de la liste.

Pourquoi:

- Si la proposition 1 est vraie, on corrompraît la liste.
- Si la proposition 2 est vraie, on créerait un cycle dans la liste chaînée.

Solution:

- Ajouter ces deux propositions en tant que précondition.

On a donc:

```
lemma list_insert_front_correct :  
  "[[ x ∉ set xs ; x ≠ NULL ; x ≠ ptr_coerce l ]] ⇒  
    {λs. listp xs l s ∧ is_valid_list_C s x }  
    all.list_insert_front' l x  
    {λr. listp (x#xs) l }!"
```

La preuve peut donc permettre de corriger des erreurs de spécifications.

voir ([proof/Listp.thy](#))

Conclusion

- La preuve formule des garanties (mécaniques) sur des propriétés du programmes.
- Ces propriétés sont définies (posés) dans son contexte: La Théorie.
- Ce contexte peut être aveugle à certaines choses.
- Les projets prouvés ont une valeur que dans ce contexte.

Questions

Sources & Slides disponibles sur

<https://github.com/PolyB/isabelle-proof-exemple>

(Pas encore public)