Preuve formelle de micro-noyau

Adrien Stalain

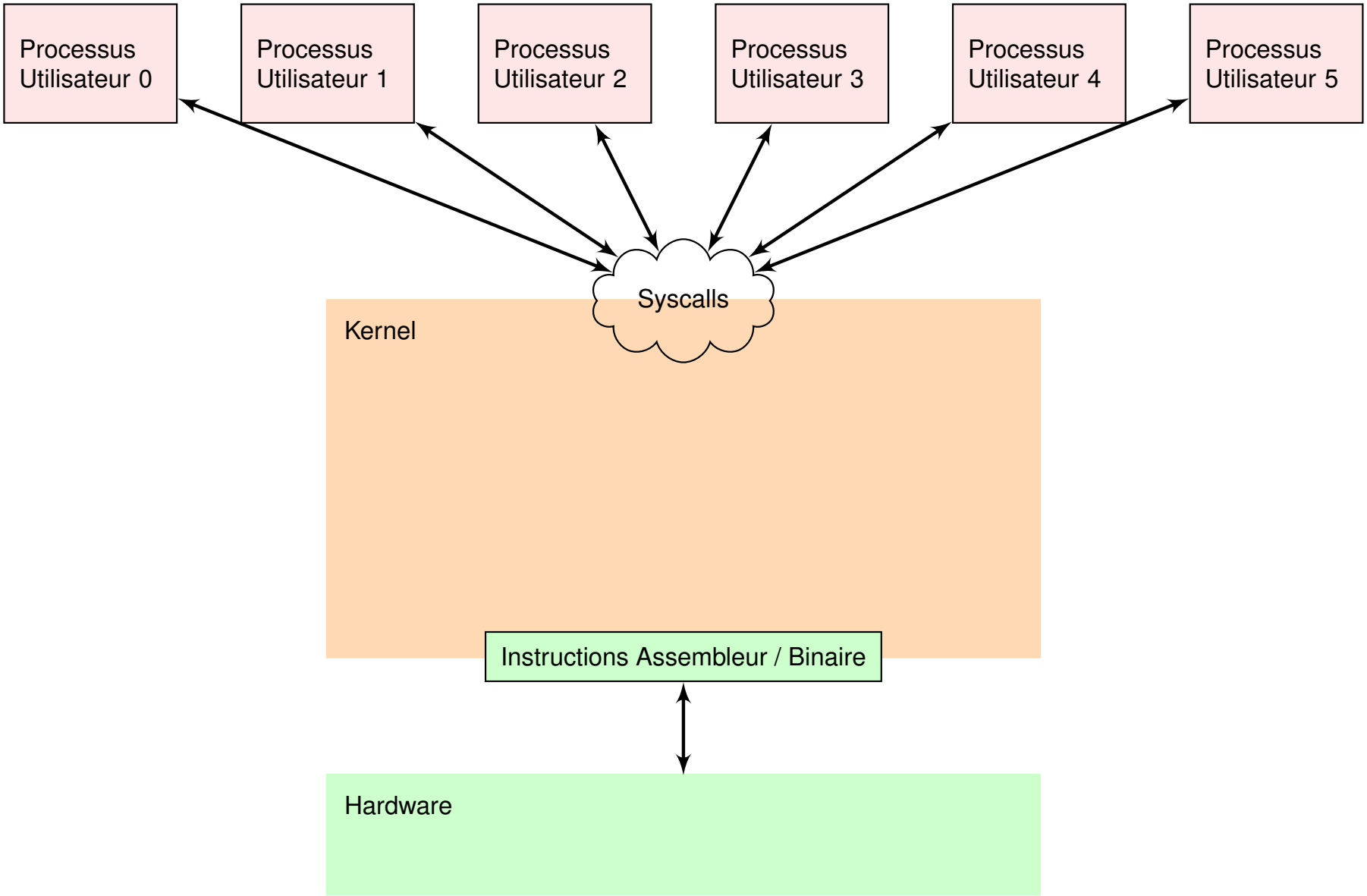
Orange Labs - Stagiaire

Tuteur - V. Sanchez Leighton

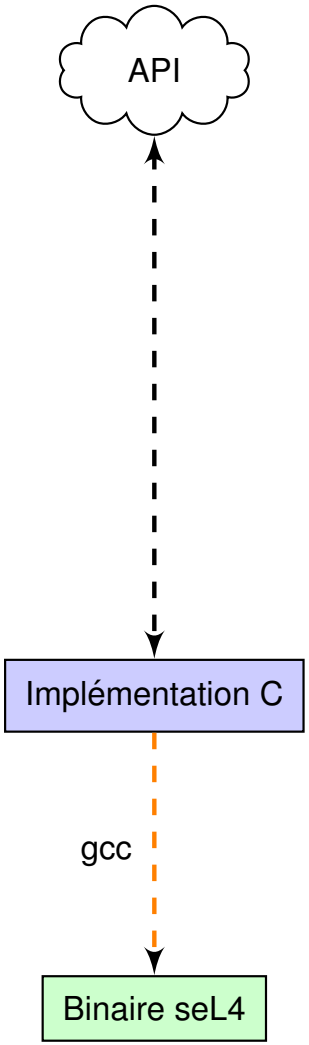
August 9, 2019

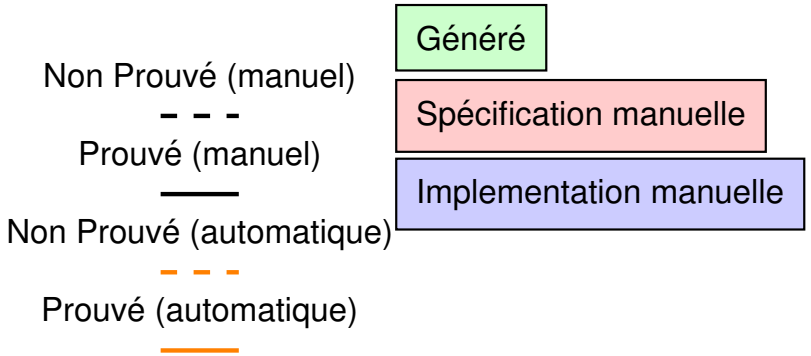
SeL4

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

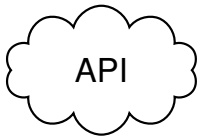

Processus Utilisateur 0
Processus Utilisateur 1
Processus Utilisateur 2
Processus Utilisateur 3
Processus Utilisateur 4
Processus Utilisateur 5
Syscalls
Kernel
Instructions Assembleur / Binaire
Hardware

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 chainée.

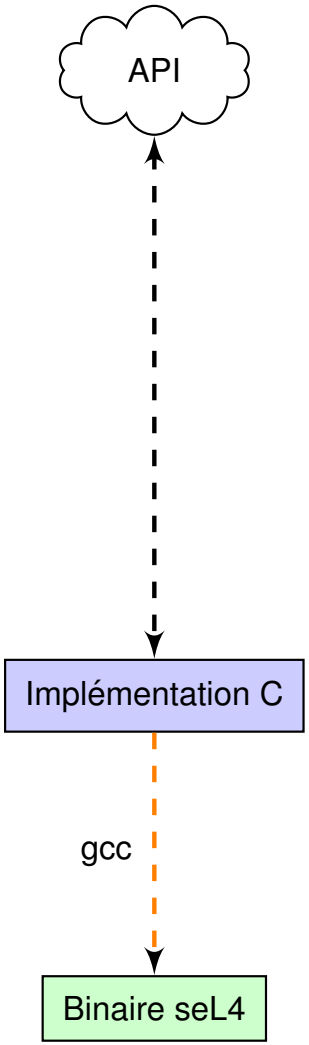
Implémentation C

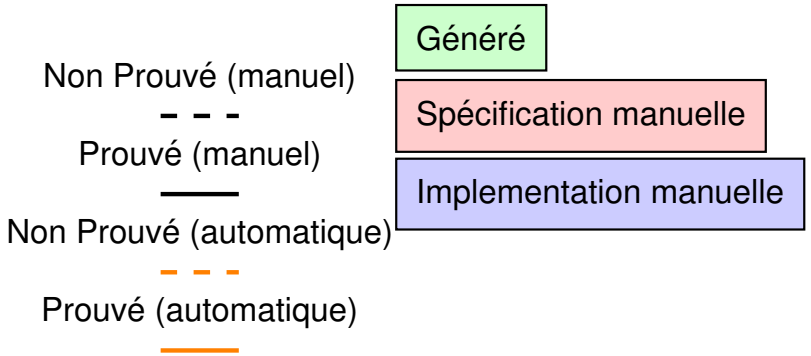
Le Code source C

Exemple

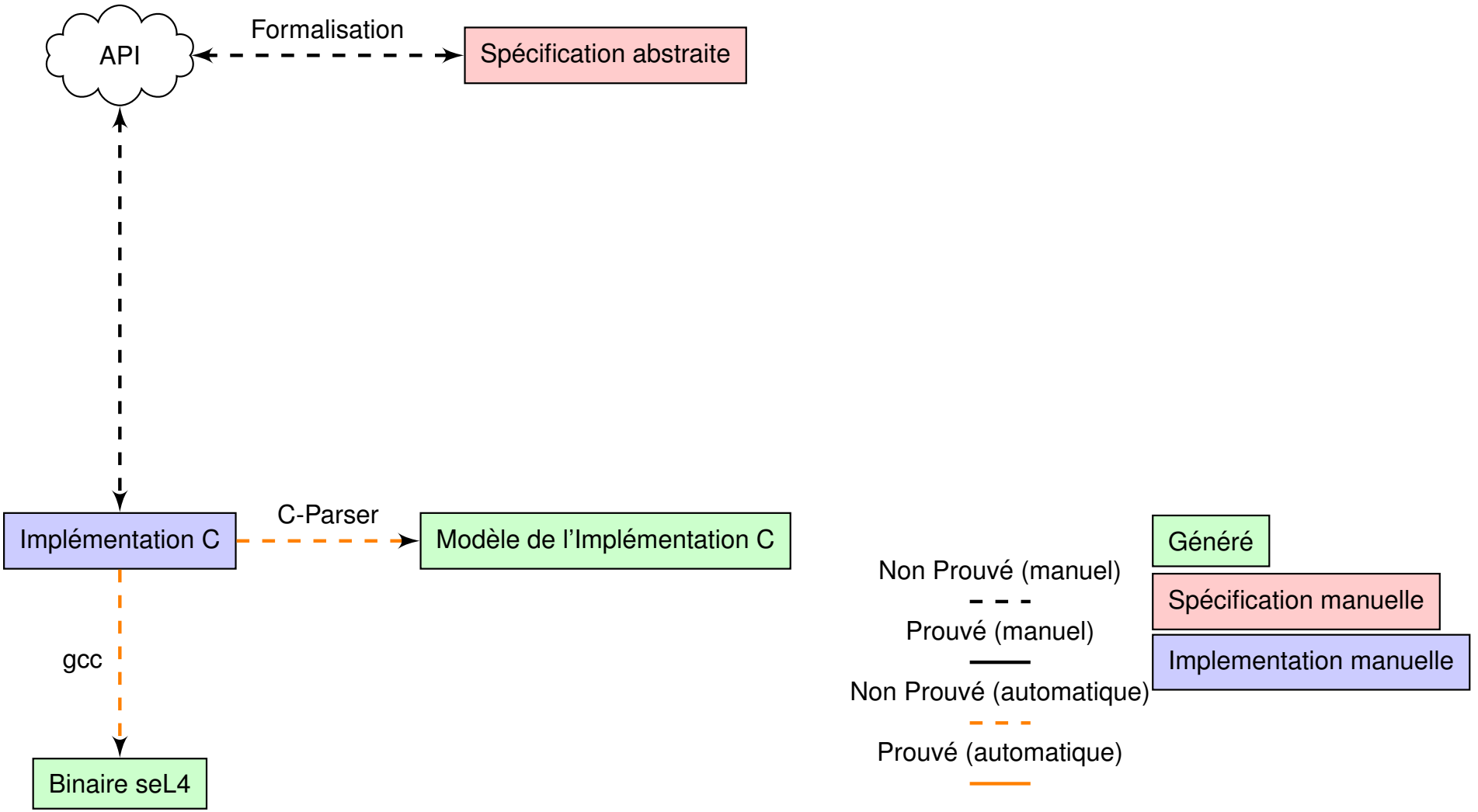
```
typedef struct list *list_t;
void list_insert_front(list_t *l, struct list *x)
{
  x->next = *l;
  *l = x;
}
```

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 ≠ NULL ⟹
    {|λs. listp xs l s ∧ is_valid_list_C s new|}
        list_insert_front l new
    {|λ_. listp (new#xs) l|}!"
```

Modèle de l'implémentation C

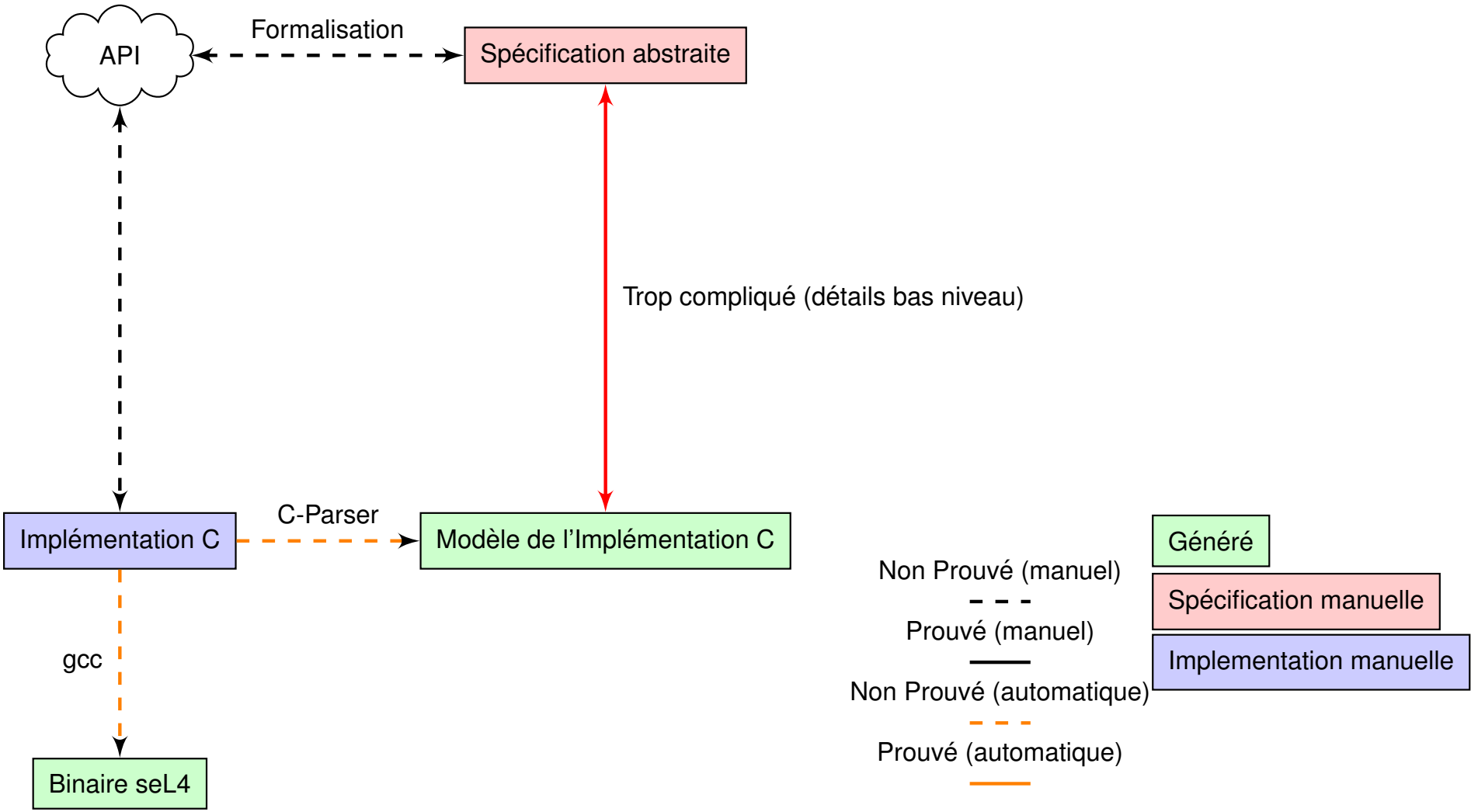
Modèle du C

Une représentation formelle du comportement de la fonction.

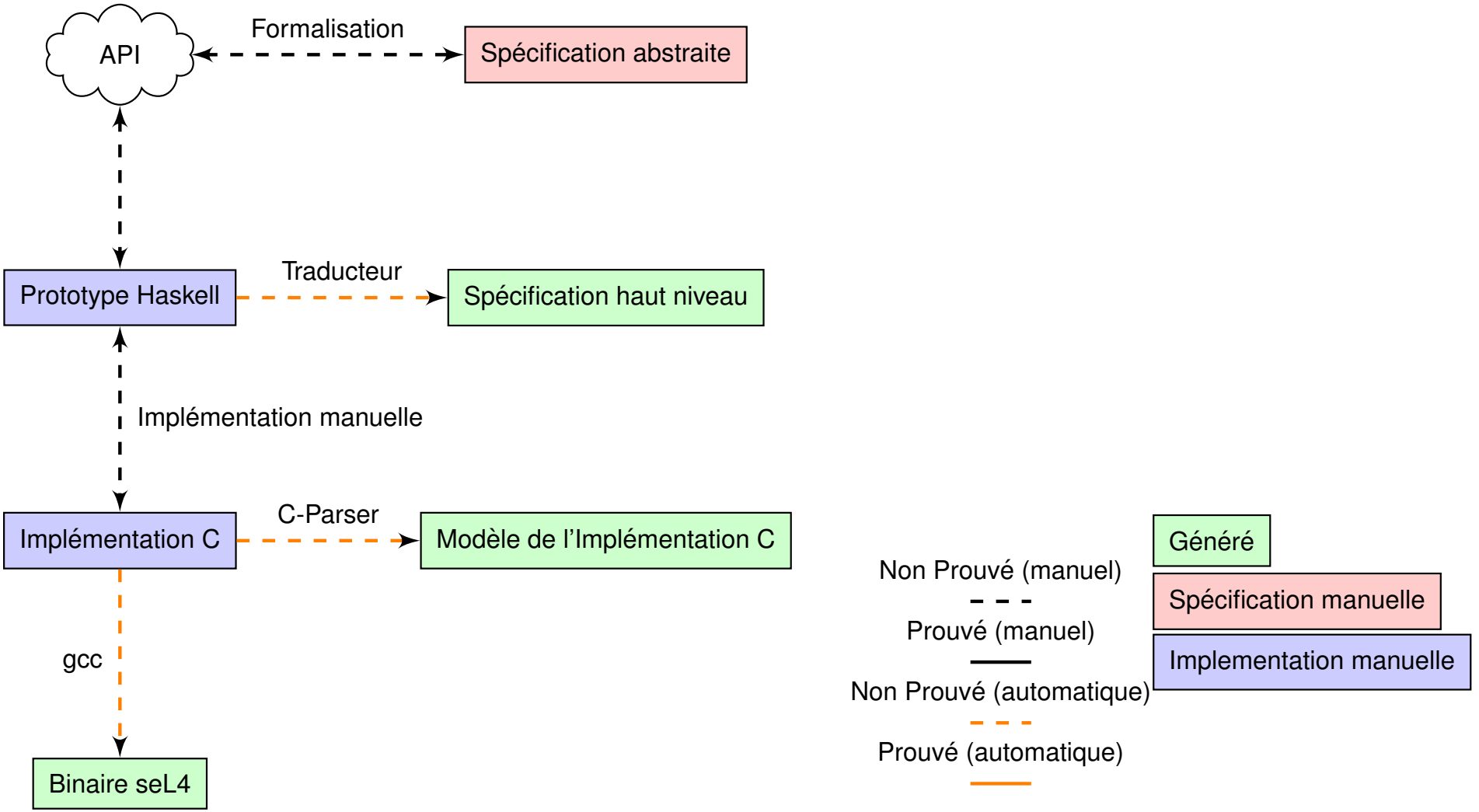
Exemple

```
"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

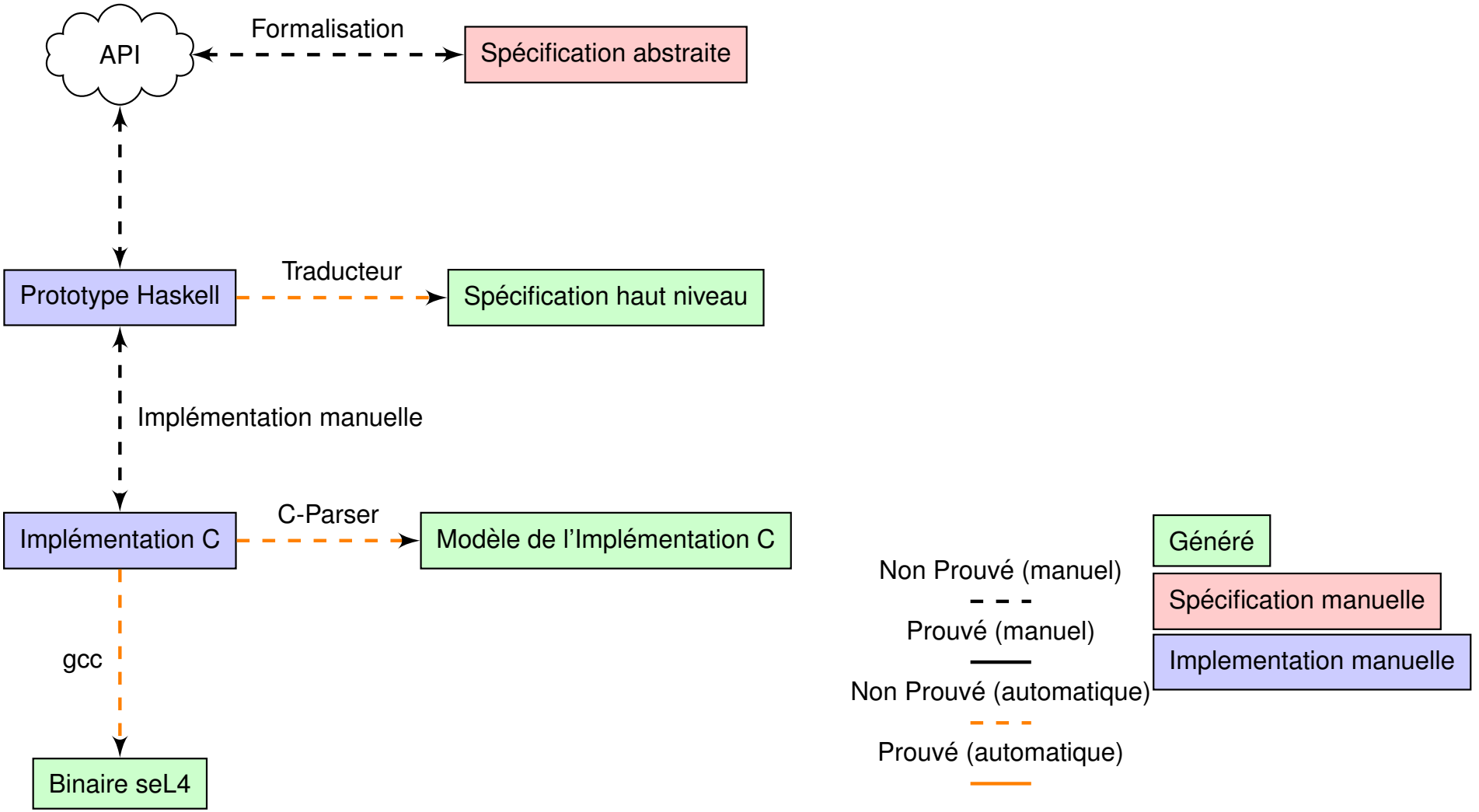


Prototype Haskell

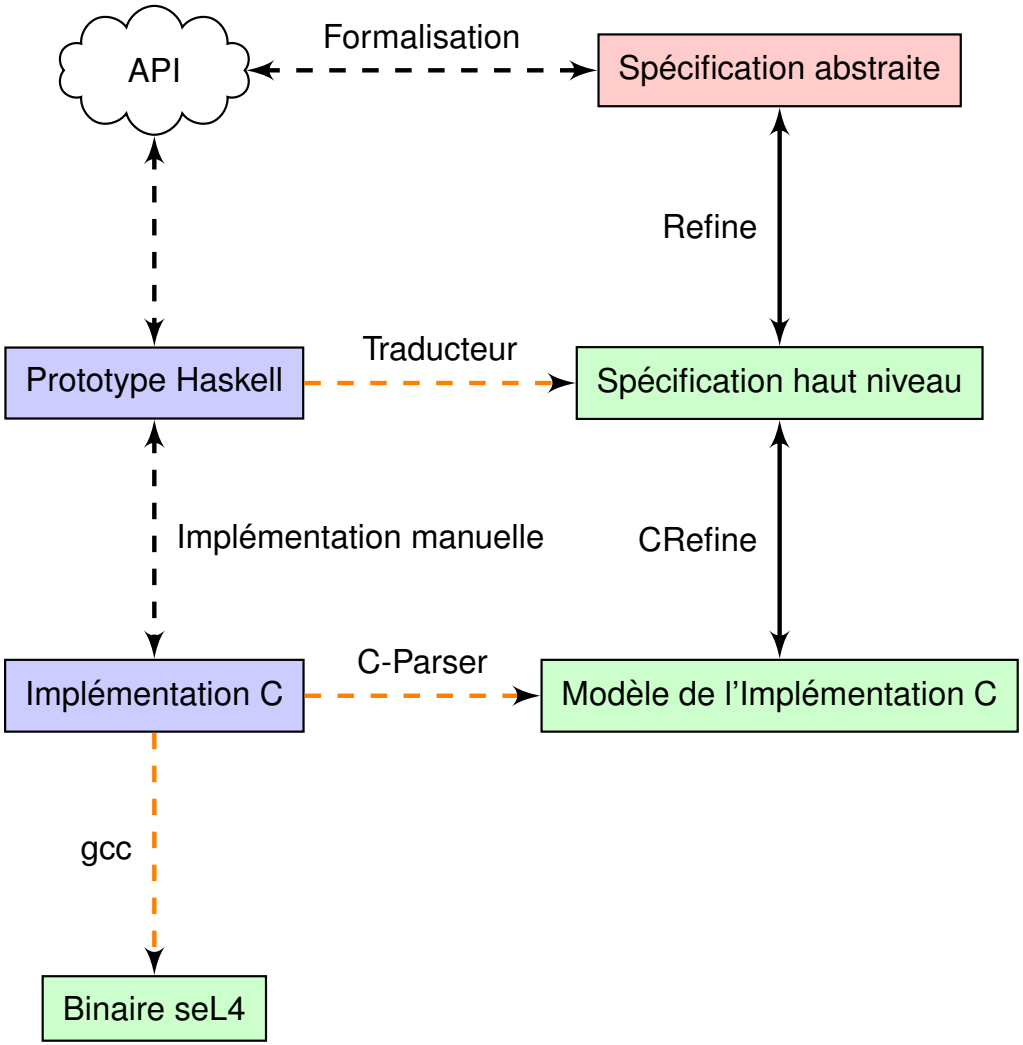
Exemple

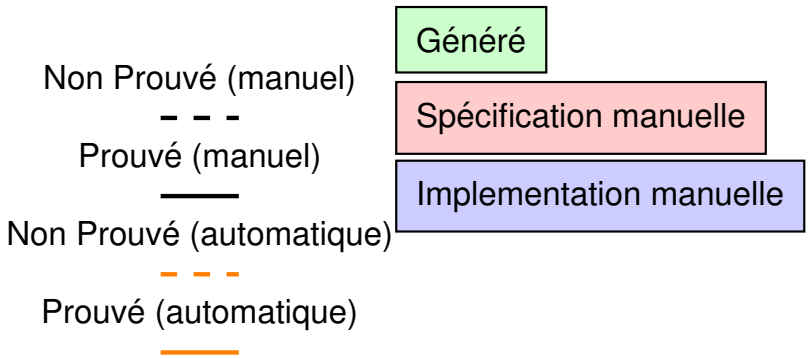
```
insert_front::PPtr ListT -> PPtr List -> Kernel ()
insert_front l new = do
                     mlist <- getListPtr l
                     updateNext new mlist
                     updateListPtr mlist new
```

La preuve de SeL4

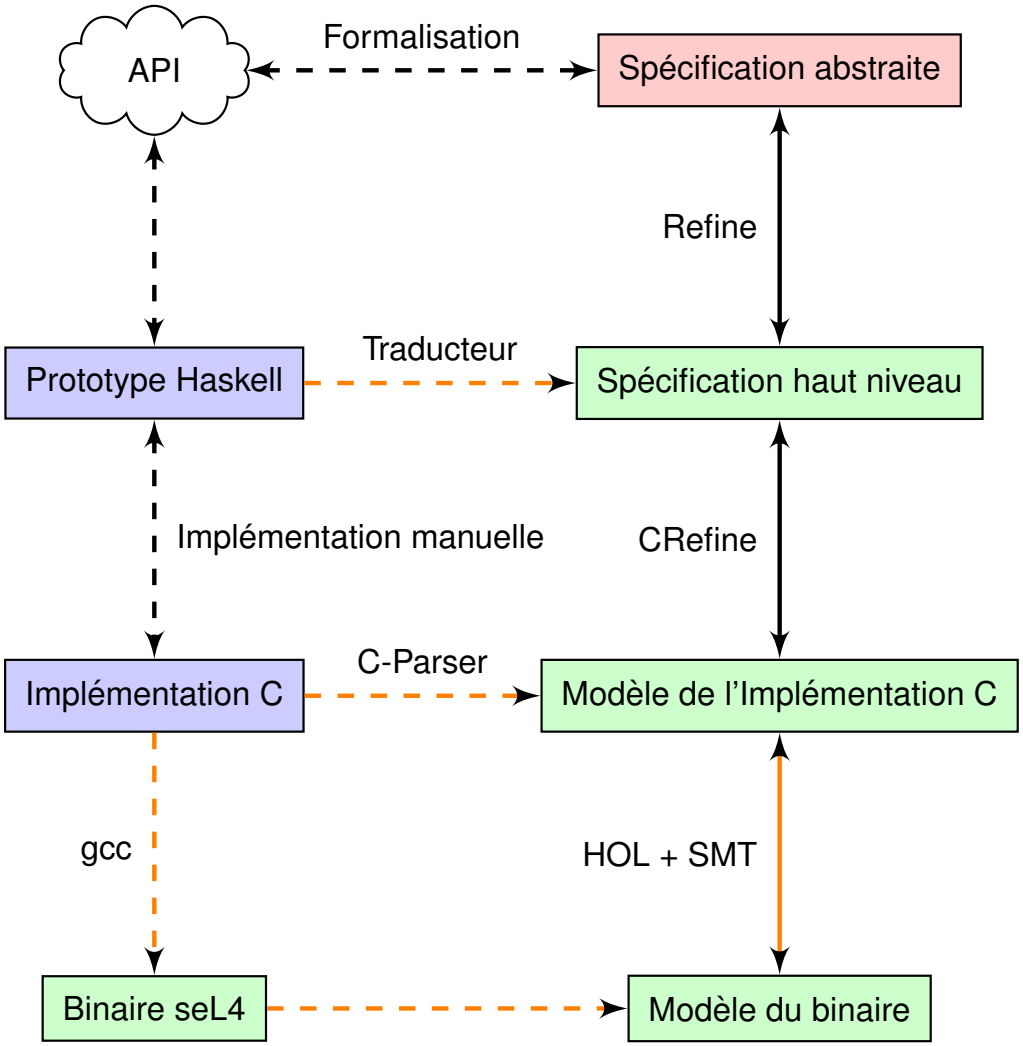


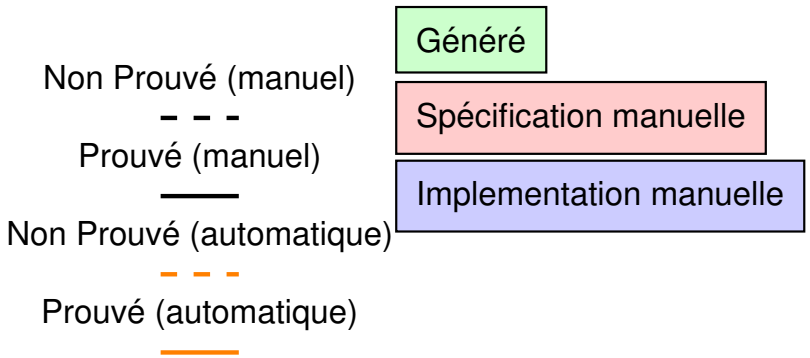
La preuve de SeL4



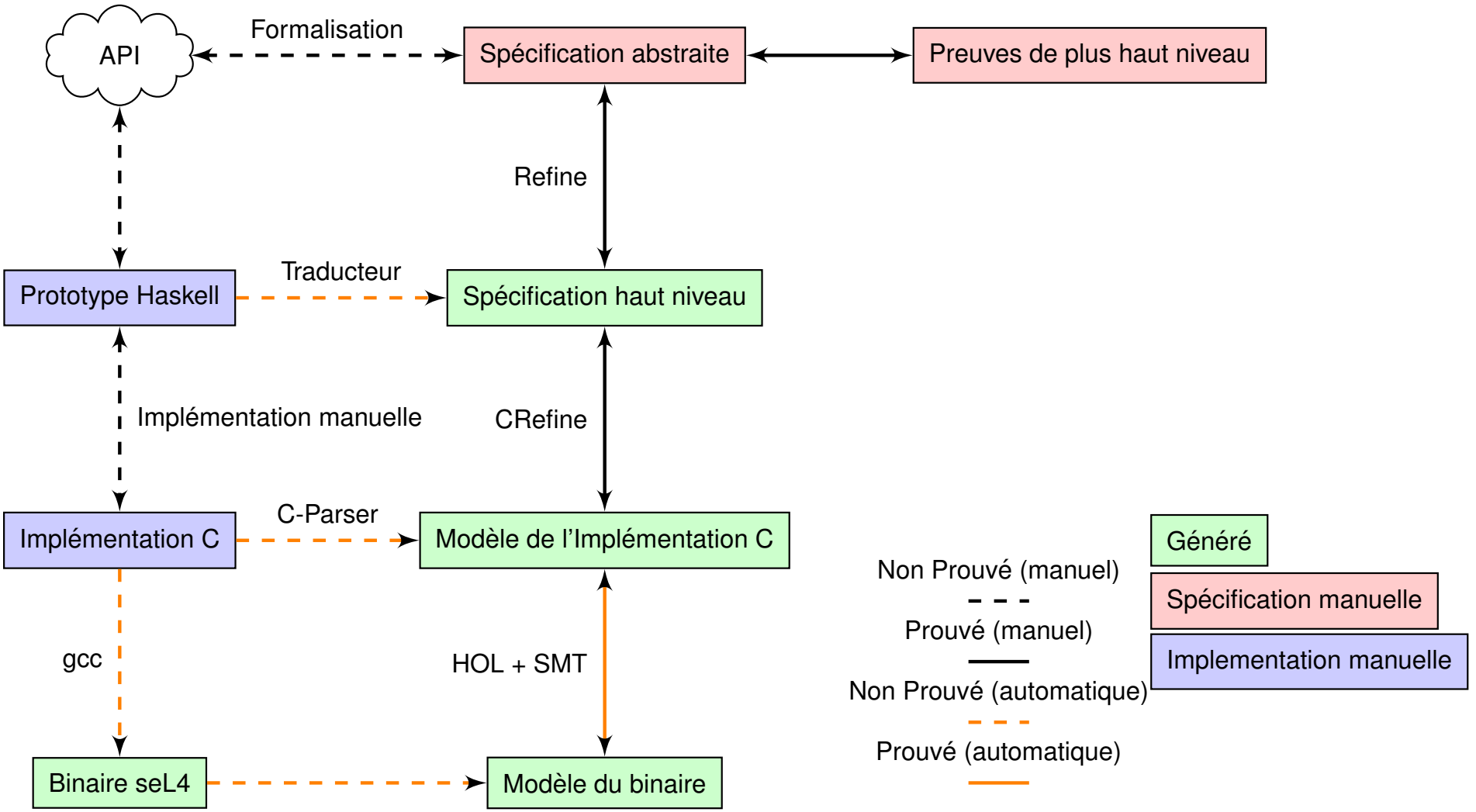


La preuve de SeL4

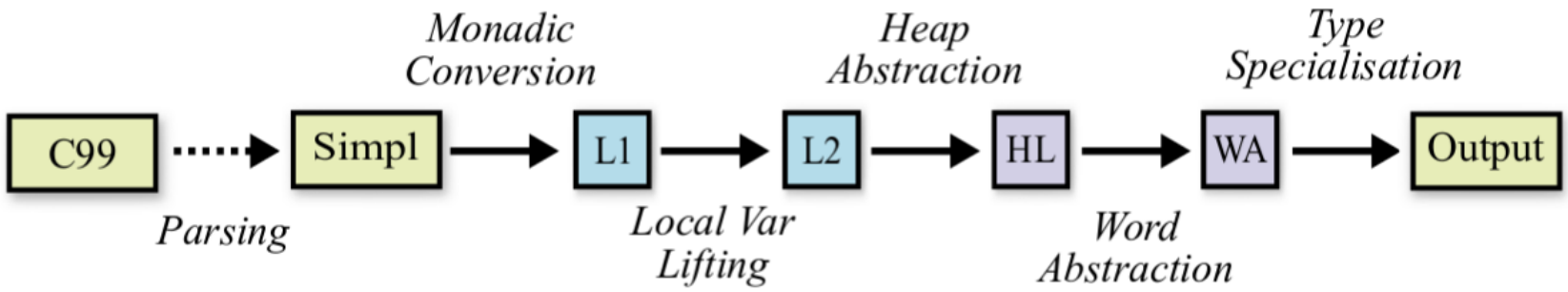




La preuve de SeL4



Autocorres



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

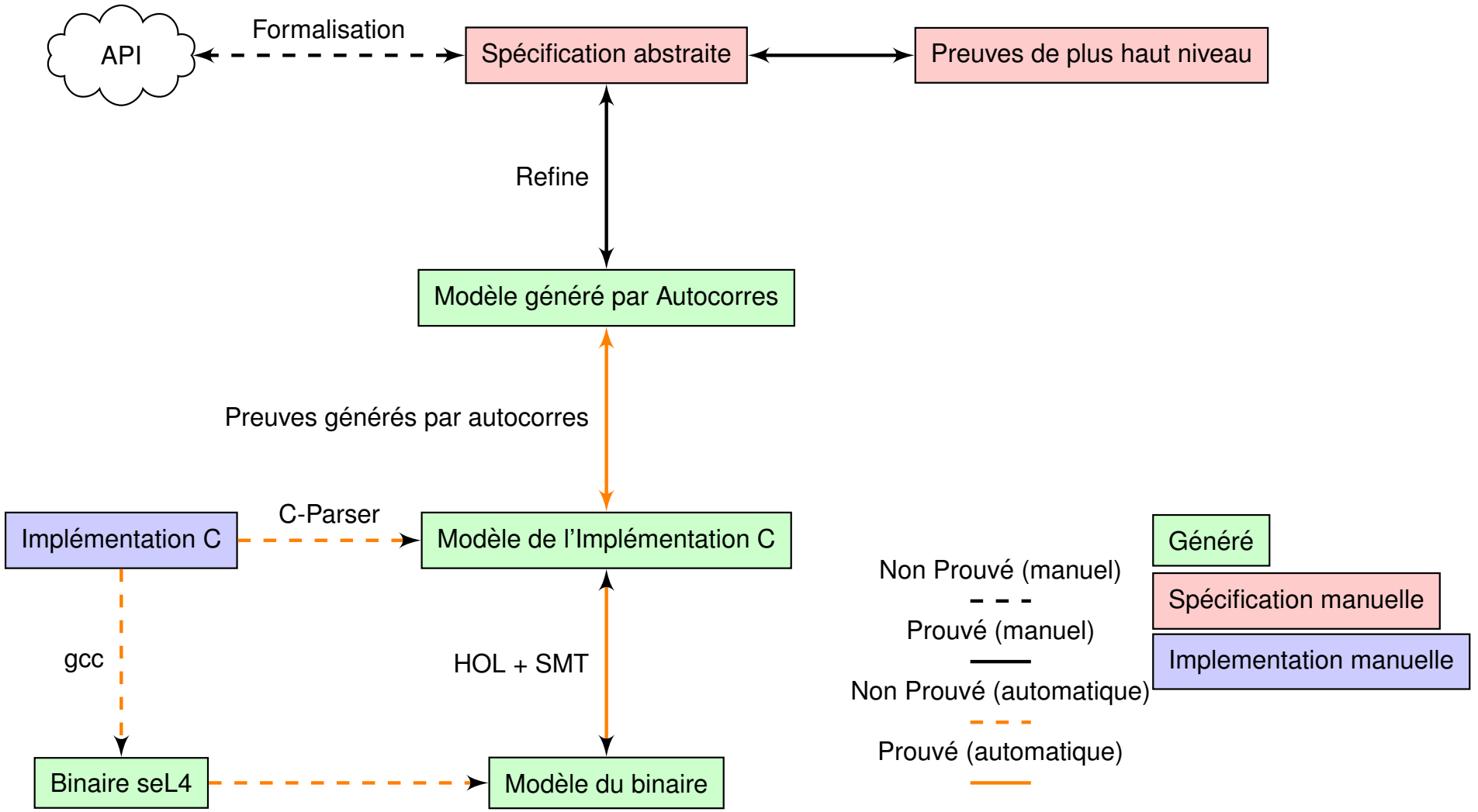
– David Greenaway, Japheth Lim, June Andronick, Gerwin Klein

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

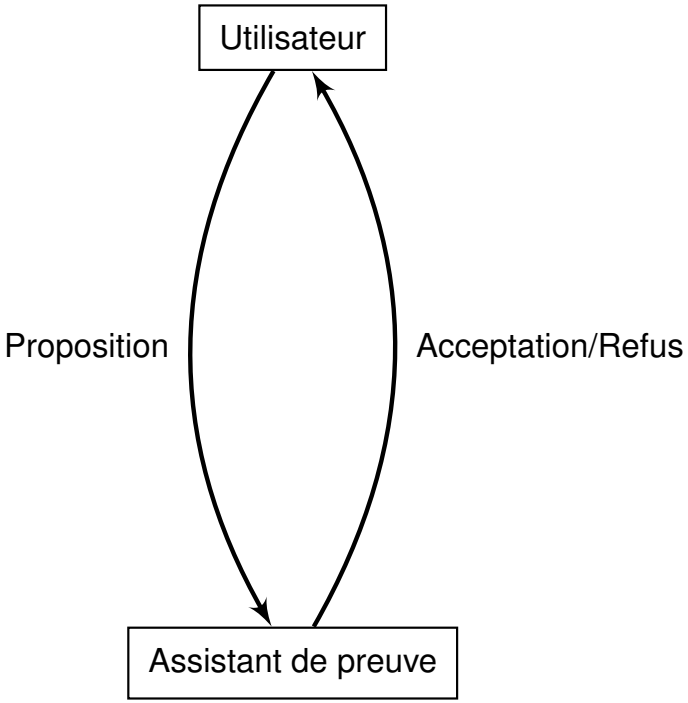
```
void list_insert_front
  (list_t *l, struct list *new)
{
  new->next = *l;
  *l = new;
}
```

```
"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'intéraction 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 / 2)
  od
 {| λr s. r * 2 = x |}"
```

```
/* Pre: (x % 2) == 0 */
/* Post: (ret_value * 2) == x */
int f (int x)
{
  return x / 2;
}
```

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 chainée

```
typedef void *data_ptr;

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

typedef struct list *list_t;
```

Hoare: Weakest Precondition

```
// Pre: l = [x,xs...]
struct list *list_pop(list_t *l)
{
  struct list *first = *l;
  *l = first->next;
  first->next = NULL;
  return first;
}
// Post: l = [xs...] , return_value = [x]
```

Hoare: Weakest Precondition

```
// Pre: l = [x,xs...]
struct list *list_pop(list_t *l)
{
  struct list *first = *l;
  *l = first->next;
  first->next = NULL;
  // l = [xs...], first = [x]
  return first;
}
// Post: l = [xs...] , return_value = [x]
```

Hoare: Weakest Precondition

```
// Pre: l = [x,xs...]
struct list *list_pop(list_t *l)
{
  struct list *first = *l;
  *l = first->next;
  // l = [xs...], first = [x,?...],
  first->next = NULL;
  // l = [xs...], first = [x]
  return first;
}
// Post: l = [xs...] , return_value = [x]
```

Hoare: Weakest Precondition

```
// Pre: l = [x,xs...]
struct list *list_pop(list_t *l)
{
  struct list *first = *l;
  // l = ???, first = [x,?...], first = [?,xs...],
  *l = first->next;
  // l = [xs...], first = [x,?...],
  first->next = NULL;
  // l = [xs...], first = [x]
  return first;
}
// Post: l = [xs...] , return_value = [x]
```

Hoare: Weakest Precondition

```
// Pre: l = [x,xs...]
struct list *list_pop(list_t *l)
{
  struct list *first = *l;
  // l = ???, first = [x,xs...]
  // l = ???, first = [x,?...], first = [?,xs...],
  *l = first->next;
  // l = [xs...], first = [x,?...],
  first->next = NULL;
  // l = [xs...], first = [x]
  return first;
}
// Post: l = [xs...] , return_value = [x]
```

Hoare: Weakest Precondition

```
// Pre: l = [x,xs...]
struct list *list_pop(list_t *l)
{
  // l = [x,xs...]
  struct list *first = *l;
  // l = ???, first = [x,xs...]
  // l = ???, first = [x,?...], first = [?,xs...]
  *l = first->next;
  // l = [xs...], first = [x,?...]
  first->next = NULL;
  // l = [xs...], first = [x]
  return first;
}
// Post: l = [xs...] , return_value = [x]
```

Autocorres avec la syntaxe Lifted heap

C	Isabelle
`struct list a;`	a :: *list_C*
`struct list *a;`	a :: *list_C ptr*
`a.next`	$a \rightarrow next$ :: *list_C ptr*
	s :: *lifted_globals*
`*a`	$s[a]$ :: *list_C*
`a->next`	$s[a] \rightarrow next$ :: *list_Cptr*
`*a = x`	$s[a := x]$ :: *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 chainée

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 ⇒ node ⇒ lifted_globals ⇒ bool"
    where
```

Définition formelle du concept de liste chainée

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 ⇒ node ⇒ lifted_globals ⇒ bool"
    where
list_is_empty:  "list [] p s = (p = NULL)"
```

Définition formelle du concept de liste chainée

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 ⇒ node ⇒ lifted_globals ⇒ bool"
    where
list_is_empty:  "list [] p s = (p = NULL)" |
list_is_cons:  "list (x#xs) p s = ( p = x
                                  ∧ is_valid_list_C s x
                                  ∧ x ≠ NULL
                                  ∧ list xs s[x]→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 ⇒ node ptr ⇒ lifted_globals ⇒
    bool" where
  "listp x pt s ≡ (list x s[pt] s
                 ∧ ptr_coerce pt ∉ set x
                 ∧ 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'execution 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'execution 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'execution 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 ≡
  do guard (λs. is_valid_list_C'ptr s ?l);
     modify (λs. s[?l := NULL])
  od"
```

Démonstration: preuve de `list_empty`

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

list_empty_correct ✓

list_empty_pure ✓

```c
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 elément `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:

voir (`proof/Listp.thy`)

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` $\neq$ `(struct list *)l`.

voir (`proof/Listp.thy`)

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 ≠ (struct list *)l`.
- Si `l` et `x` sont des pointeurs valides sur leur types respectifs.

voir (`proof/Listp.thy`)

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 ≠ (struct list *)l`.
- Si `l` et `x` sont des pointeurs valides sur leur types respectifs.
- Alors, après l'execution de `list_singleton l x`.

voir (`proof/Listp.thy`)

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 ≠ (struct list *)l`.
- Si `l` et `x` sont des pointeurs valides sur leur types respectifs.
- Alors, après l'execution de `list_singleton l x`.
- On aura `l` un pointeur sur liste valide, contenant uniquement x

voir (`proof/Listp.thy`)

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 ≠ (struct list *)l`.
- Si `l` et `x` sont des pointeurs valides sur leur types respectifs.
- Alors, après l'execution 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.

voir (`proof/Listp.thy`)

Démonstration: preuve de `list_singleton`

Problème: on arrive pas a prouver que:

1. $\forall$ `s. [...]` $\Longrightarrow$ `s[x]`$\rightarrow$`next = NULL`

Problème: on arrive pas a prouver que:

1. $\forall$ `s. [...]` $\Longrightarrow$ `s[x]`$\rightarrow$`next = NULL`

C'est à dire:

- Que `x->next == NULL`.

Problème: on arrive pas a prouver que:

`1. ∀s. [...] ⟹ s[x]→next = NULL`

C'est à dire:

- Que `x->next == NULL`.

Pourquoi:

- Si la proposition est fausse, `x` ne serait pas nécéssairement le dernier élément puisque son `→next` est pas forcémment `NULL`.
- C'est donc un bug dans la fonction.

Problème: on arrive pas a prouver que:

```
1. ∀s. [...] ⟹ s[x]→next = NULL
```

C'est à dire:

- Que `x->next == NULL`.

Pourquoi:

- Si la proposition est fausse, `x` ne serait pas nécéssairement le dernier élément puisque son `→next` est pas forcémment `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 permetre 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:

voir (`proof/Listp.thy`)

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.

voir (`proof/Listp.thy`)

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.

voir (`proof/Listp.thy`)

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'execution de `list_insert_front l x`.

voir (`proof/Listp.thy`)

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'execution de `list_insert_front l x`.
- On aura `l` un pointeur sur liste valide, contenant x suivi des éléments de xs

voir (`proof/Listp.thy`)

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'execution 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 ne se produisent, et en un temps fini.

voir (`proof/Listp.thy`)

Démonstration: preuve de `list_insert_front`

Problème: on n'arrive pas a prouver que:

1. $\forall$s. [...] $\Longrightarrow$ ptr_coerce l $\neq$ x
2. $\forall$s. [...] $\Longrightarrow$ x $\notin$ set xs

Problème: on n'arrive pas a prouver que:

1. $\forall$`s. [...]` $\Longrightarrow$ `ptr_coerce l` $\neq$ `x`
2. $\forall$`s. [...]` $\Longrightarrow$ `x` $\notin$ `set xs`

C'est à dire:

- Que `(struct list*)l` est différent de `x`.
- Et que `x` n'est pas déja une node de la liste.

Problème: on n'arrive pas a prouver que:

1. $\forall$s. [...] $\Longrightarrow$ `ptr_coerce l` $\neq$ `x`
2. $\forall$s. [...] $\Longrightarrow$ `x` $\notin$ `set xs`

C'est à dire:

- Que `(struct list*)l` est différent de `x`.
- Et que `x` n'est pas déja une node de la liste.

Pourquoi:

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

Problème: on n'arrive pas a prouver que:

1. $\forall$s. [...] $\Longrightarrow$ `ptr_coerce l` $\neq$ `x`
2. $\forall$s. [...] $\Longrightarrow$ `x` $\notin$ `set xs`

C'est à dire:

- Que `(struct list*)l` est différent de `x`.
- Et que `x` n'est pas déja une node de la liste.

Pourquoi:

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

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 (posées) dans son contexte: La Théorie.
- Ce contexte peut être aveugle à certaines choses.
- Les projets prouvés ont une valeur que dans ce contexte.

Questions