TODO

Adrien Stalain

April 11, 2019

```
theory Listp
  imports "Import_C"
begin
```

0.1 list

0.1.1 list definition

Le `list_C` est un type importé du C (struct list) qui représente une liste chainée, sa définition est:

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

```
type_synonym list_nodes = "list_C ptr list"
```

```
primrec list :: "list_nodes ⇒ list_C ptr ⇒ 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 p
                             ∧ p ≠ NULL
                             ∧ list xs s[p]→next s
                             )"
```

La propritétée `list x p s` est vraie quand `p` est une liste valide contenant les nodes `x` dans l'état global `s`

Une liste bien définie est soit vide, et dans ce cas là `p` est `NULL`, soit une liste commençant par `x` et avec comme reste `xs` et dans ce cas là `p = x`, p est valide dans l'état de heap `s` (`is_valid_list_C s p`) `p` n'est pas `NULL`, et `xs` est une liste valide qui commence par `s[p]→next`

```
definition a_list :: "list_C ptr ⇒ lifted_globals ⇒ bool" where
  "a_list p s ≡ (∃xs. list xs p s)"
```

La propriétée `a_list p s` est valide si `p` est une liste valide dans l'état `s`

0.1.2 list properties

```
lemma list_is_empty_r : "list a NULL s ⟹ [] = a"
```

Une liste valide qui commence par `NULL` ne peux que être vide

```
lemma list_is_cons_r : "p ≠ NULL ⟹ list x p s = (∃ys. (x = p#ys) ∧ (is_valid_list_C
s p)
                                                    ∧ (p ≠ NULL) ∧ (list ys s[p]→next
s))"
```

Une liste valide qui n'est pas `NULL`, contient au moins un élément

```
lemma list_not_2_same : "list (x#y#z) p s ⟹ x ≠ y"

lemma list_append_Ex: "list (xs @ ys) p s ⟹ (∃q. list ys q s)"

lemma list_unique:  "⟦ list xs p s ; list ys p s ⟧ ⟹ xs = ys"

lemma list_distinct : "list x p s ⟹ distinct x"

lemma list_head_not_in_cons : "list (x#xs) x s ⟹ x ∉ set xs"

lemma the_list_unique : "list xs p s ⟹ (THE ys. list ys p s) = xs"

lemma list_next_in_list : "⟦ list xs p s ; a ∈ set xs ; s[a]→next ≠ NULL ⟧⟹(s[a]→next)
∈ set xs"

lemma list_has_end_not_null : "list (xs @ [x]) p s ⟹ p ≠ NULL"

lemma list_no_next_is_last : "⟦ list (xs @ [x]) p s ; w ∈ set (xs @ [x]) ; s[w]→next
= NULL ⟧⟹ w = x"

lemma list_last_next_is_null : "list (xs @ [x]) p s ⟹ s[x]→next = NULL"

lemma list_content_is_valid : "⟦ list xs p s ; w ∈ set xs ⟧ ⟹ is_valid_list_C
s w ∧ w ≠ NULL
                                                                  ∧ (∃ys. list ys s[w]→next
s)"

lemma first_element_in_list : "⟦ list xs p s ; p ≠ NULL ⟧ ⟹ p ∈ set xs"
```

0.2 listp

0.2.1 listp definition

```
definition listp :: "list_nodes ⇒ list_C ptr ptr ⇒ lifted_globals ⇒ bool" where
  "listp n pt s ≡ (ptr_coerce pt ∉ set n ∧ is_valid_list_C'ptr s pt ∧ list n s[pt]
s)"
```

La propriétée `listp x p s` indique que `p` est une liste valide contenant les nodes `x` dans l'état global `s`

0.2.2 listp properties

```
lemma listp_unique: "⟦ listp xs p s ; listp ys p s ⟧ ⟹ xs = ys"
```

state update

lemma list_st_update_ignore [simp] : "q ∉ set xs ⟹ list xs p (s[q→next := ω]) = list xs p s"

lemma list_st_add [simp] : "⟦ is_valid_list_C s x ; x ≠ NULL; x ∉ set xs ⟧ ⟹ list (x#xs) x s[x→next := p] = list xs p s"

lemma list_st_upd_any_base_ptr [simp] : "ptr_coerce (l :: list_C ptr ptr) ∉ set xs ⟹ list xs p s[l := ω] = list xs p s"

0.3 hoare helpers

0.3.1 nondet monad

lemma grab_asm_NF : "(G ⟹ ⦃P⦄ f ⦃Q⦄!) ⟹ ⦃λs. G ∧ P s⦄ f ⦃Q⦄!"

lemma hoare_conjINF:
"⟦ ⦃P⦄ f ⦃Q⦄; ⦃P⦄ f ⦃R⦄! ⟧ ⟹ ⦃P⦄ f ⦃λr s. Q r s ∧ R r s⦄!"

lemma hoare_conjINFR:
"⟦ ⦃P⦄ f ⦃Q⦄!; ⦃P⦄ f ⦃R⦄ ⟧ ⟹ ⦃P⦄ f ⦃λr s. Q r s ∧ R r s⦄!"

lemma hoare_transf_predNF: "⟦ (G' ⟹ ⦃ P ⦄ f ⦃ Q ⦄!) ; (G ⟹ G') ⟧ ⟹ ⦃ λs. G ∧ P s ⦄ f ⦃ Q ⦄!"

0.3.2 option monad

lemma ovalidNF_is_validNF : "ovalidNF P f P' ⟹ ⦃ P ⦄ gets_the f ⦃ P'⦄!"

lemma ovalid_is_valid : "ovalid P f P' ⟹ ⦃ P ⦄ gets_the f ⦃ P'⦄"

lemma ovalid_herit_NF : "⟦ ovalid P f Q ; ovalidNF P f Q' ⟧ ⟹ ovalidNF P f Q"

lemma ovalidNF_comb3p [wp_comb]:
"⟦ ovalidNF P f Q; ovalidNF P f Q' ⟧ ⟹ ovalidNF (λs. P s) f (λr s. Q r s ∧ Q' r s)"

lemma ovalidNF_comb3pr [wp_comb]:
"⟦ ovalidNF P f Q; ovalid P f Q' ⟧ ⟹ ovalidNF (λs. P s) f (λr s. Q r s ∧ Q' r s)"

lemma ovalid_grab_asm2: "(P' $\Longrightarrow$ ovalid (λs. P s $\wedge$ R s) f Q) $\Longrightarrow$ ovalid (λs. P s $\wedge$ P' $\wedge$ R s) f Q"

lemma ovalid_drop_post: "$[\![$ R ; ovalid P f Q $]\!]$ $\Longrightarrow$ ovalid P f (λr s. R $\wedge$ Q r s)"

lemma ovalidNF_drop_post: "$[\![$ R ; ovalidNF P f Q $]\!]$ $\Longrightarrow$ ovalidNF P f (λr s. R $\wedge$ Q r s)"

0.4 isabelle lists helpers

lemma list_non_empty_has_init : "x $\neq$ [] $\Longrightarrow$ $\exists$w. w @ [last x] = x"

0.5 state helpers

lemma next_upd_diff : "x $\neq$w $\Longrightarrow$ s[x$\rightarrow$next := a][w$\rightarrow$next := b] $\equiv$ s[w$\rightarrow$next := b][x$\rightarrow$next := a]"

lemma listptrptr_upd_not_mod : "ptr_coerce (k :: list_C ptr ptr) $\neq$ (x :: list_C ptr)
$\Longrightarrow$ s[k := ω][x] = s[x]"

0.6 program proof

0.6.1 list_empty

correct

lemma list_empty_correct : "$\{\!|$ λ s. is_valid_list_C'ptr s l $|\!\}$ all.list_empty' l $\{\!|$ λ_. listp [] l $|\!\}$!"

pure

lemma list_empty_pure : " ($\forall$s. P s $\longrightarrow$ P s[l := NULL]) $\Longrightarrow$ $\{\!|$ P $|\!\}$ all.list_empty' l $\{\!|$ λ_. P $|\!\}$"

0.6.2 list_insert_front

correct

lemma list_insert_front_correct : "$[\![$ x $\notin$ set xs ; x $\neq$ NULL ; x $\neq$ 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 ⦄!"
```

pure

```
lemma  list_insert_front_pure : "(∀s. P s ⟶  P s[x→next := s[l]][l := x])
                                                        ⟹ ⦃ P ⦄ all.list_insert_front'
l x ⦃λ_. P ⦄"
```

0.6.3 list_singleton

correct

```
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 r. listp [x] l ⦄!"
```

pure

```
lemma list_singleton_pure : "(∀s. P s ⟶ P s[l := x][x→next := NULL]) ⟹ ⦃
P ⦄ all.list_singleton' l x ⦃ λ_. P ⦄"
```

0.6.4 list_insert_after

correct

```
lemma list_insert_inside : "⟦ n ∉ set(x1 @ w # x2) ; n ≠ NULL ; is_valid_list_C
s n
                            ; list (x1 @ [w] @ x2) p s⟧
                            ⟹ list (x1 @ w # n # x2) p s[w→next := n][n→next
:= s[w]→next]"

lemma list_insert_after_correct : "⟦ x ≠ w ; x ∉ set xa ; x ∉ set xb ; x ≠ NULL
⟧
                                          ⟹ ⦃ λs. list (xa @ [w] @ xb) p s ∧ is_valid_list_C
s x ⦄
                                             all.list_insert_after' w x
                                             ⦃ λr s. list (xa @ [w,x] @ xb) p s
⦄!"

lemma list_insert_after_correct_p : "⟦ ptr_coerce p ≠ x ;  x ∉ set (xa @ [w] @
xb) ; x ≠ NULL ⟧ ⟹ ⦃ λs. listp (xa @ [w] @ xb) p s ∧ is_valid_list_C s x ⦄ all.list_insert_a
w x ⦃ λr s. listp (xa @ [w,x] @ xb) p s ⦄!"
```

pure

```
lemma list_insert_after_pure : "(∀s ω. P s ⟶ P s[x→next := ω][w→next := x])
⟹ ⦃ P ⦄ all.list_insert_after' w x ⦃ λr. P ⦄"
```

specialisations

```
lemma list_insert_after_last : "⟦ (∃a. a @ w # [] = xs) ; x ∉ set xs ; x ≠ NULL
; (ptr_coerce p) ≠ x ⟧ ⟹ ⦃ λs. listp xs p s ∧ is_valid_list_C s x ⦄ all.list_insert_after'
w x ⦃λ r. listp (xs @ [x]) p ⦄!"
```

```
lemma list_insert_after_the_last : "⟦ x ∉ set xs ; x ≠ NULL ; (ptr_coerce p) ≠
x ; xs ≠ [] ⟧ ⟹ ⦃λs. listp xs p s ∧ is_valid_list_C s x ⦄ all.list_insert_after'
(last xs) x ⦃ λ_. listp (xs @ [x]) p⦄!"
```

```
lemma list_insert_after_the_last_pre : "⟦ x ∉ set xs ; x ≠ NULL ; (ptr_coerce
p) ≠ x ; xs ≠ [] ; w = last xs ⟧ ⟹ ⦃λs. listp xs p s ∧ is_valid_list_C s x ⦄
all.list_insert_after' w x ⦃ λ_. listp (xs @ [x]) p⦄!"
```

0.6.5 list_find_last_node

list_find_last_node_inner_loop_content

```
definition list_find_last_node_inner_loop_content :: "list_C ptr ⇒ lifted_globals
⇒ list_C ptr option" where
  "list_find_last_node_inner_loop_content p ≡ DO oguard (λs. is_valid_list_C s p);
        p <- ogets (λs. s[p]→next);
        oguard (λs. is_valid_list_C s p);
        oreturn p
     OD"
```

```
lemma list_find_last_node_inner_loop_content_correct : " ⟦ a ∈ set (xs @ [x]) ;
a ≠ NULL ⟧ ⟹ ovalidNF (λs. list (xs @ [x]) pb s ∧ a_list a s ∧ s[a]→next ≠ NULL)
         (list_find_last_node_inner_loop_content a) (λa b. a ∈ set (xs @ [x]) ∧
a ≠ NULL ∧ list (xs @ [x]) pb b ∧ a_list a b)"
```

```
lemma list_find_last_node_inner_loop_content_pure : "ovalid P (list_find_last_node_inner_loop_content
a) (λ_. P)"
```

list_find_last_node_inner_loop

```
definition list_find_last_node_inner_loop :: "list_C ptr ⇒ lifted_globals ⇒ list_C
ptr option" where
 "list_find_last_node_inner_loop p ≡ owhile (λp s. s[p]→next ≠ NULL) (λp. DO oguard
(λs. is_valid_list_C s p);
```

```
        p <- ogets (λs. s[p]→next);
        oguard (λs. is_valid_list_C s p);
        oreturn p
      OD) p"
```

lemma list_find_last_node_inner_loop_content_mesure : " ⟦ a ∈ set (xs @ [x]) ; a ≠ NULL ⟧ ⟹ ovalid
            (λs. list (xs @ [x]) pb s ∧ a_list a s ∧ s[a]→next ≠ NULL ∧ length (THE xs. list xs a s) = m)
            (list_find_last_node_inner_loop_content a)
            (λr s. length (THE xs. list xs r s) < m)"

lemma list_find_last_node_inner_loop_correct : "ovalidNF (list (xs @ [x]) pb) (list_find_last_
pb) (λr _. r = x)"

lemma list_find_last_node_inner_loop_pure : "ovalid P (list_find_last_node_inner_loop p) (λr. P)"

correct

lemma list_find_last_node_correct : "ovalidNF (listp (xs @ [x]) p) (all.list_find_last_node'
p) (λr _. r = x) "

lemma list_find_last_node_correct2 : "xs = ys @ [w] ⟹ ovalidNF (listp xs p) (all.list_find_last_node' p) (λr _. r = w) "

pure

lemma list_find_last_node_pure : "ovalid P (all.list_find_last_node' p) (λr. P)"

specialisations

lemma list_find_last_node_correct_ND : "xs ≠ [] ⟹ ⦃ listp xs l ⦄ gets_the (all.list_find_la
l) ⦃λxa s. xa = last xs⦄!"

lemma list_find_last_node_pure_ND : "⦃ P ⦄ gets_the (all.list_find_last_node' l) ⦃ λ_. P ⦄"

0.6.6 list_insert_back

correct

lemma list_insert_back_correct : "(x ∉ set xs ∧ x ≠ NULL ∧ x ≠ (ptr_coerce l)) ⟹ ⦃λs. listp xs l s ∧ is_valid_list_C s x ⦄ all.list_insert_back' l x ⦃ λ _. listp (xs @ [x]) l ⦄!"
end