Spécification et certification d'implantation chaînée avec des orbites : les hypercartes combinatoires

Jean-François Dufourd

ICUBE, Université de Strasbourg - CNRS, France

Journée GDR-LTP'2013, Bordeaux, LABRI, 18 novembre 2013

Sommaire

Introduction

- *Objectif*
  - Spécifier des librairies de types et opérations
  - Certifier des implantations (séquentielles) avec chaînages
- *Moyens utilisés*
  - Types algébriques
  - CCI (avec extensionnalité) et preuves avec Coq
  - Simulation du langage C
  - Notion d'orbite
- *Moyens non utilisés*
  - Logique de Hoare
  - Logique de séparation
- *Etude de cas*
  - Hypercartes combinatoires [Cori 70, ..., Gonthier 08, ...]

  Applications : *Combinatoire* : énumération, coloration...
  Modélisation géométrique : subdivisions de l'espace

Orbites en Coq

Contexte

- `X: Type`, avec *égalité décidable* et *exception* `exc: X`
- `f: X->X`, *fonction totale*
- `D`, *sous-domaine fini* de `X` ne contenant pas `exc`

(représenté comme une liste finie sans duplicata)

Définitions

Soient `z:X` et les *itérés* `zk := Iter f k z`, pour tout `k >= 0`.

(i) `orbs k z` := si `k = 0` alors `nil` sinon `z(k-1)::orbs (k-1) z` : `k`-*séquence orbitale* de `z`

(ii) `lorb z` : *longueur* de l'orbite de `z` : plus petit entier `p` tel que `~In zp D` ou `In zp (orbs p z)`

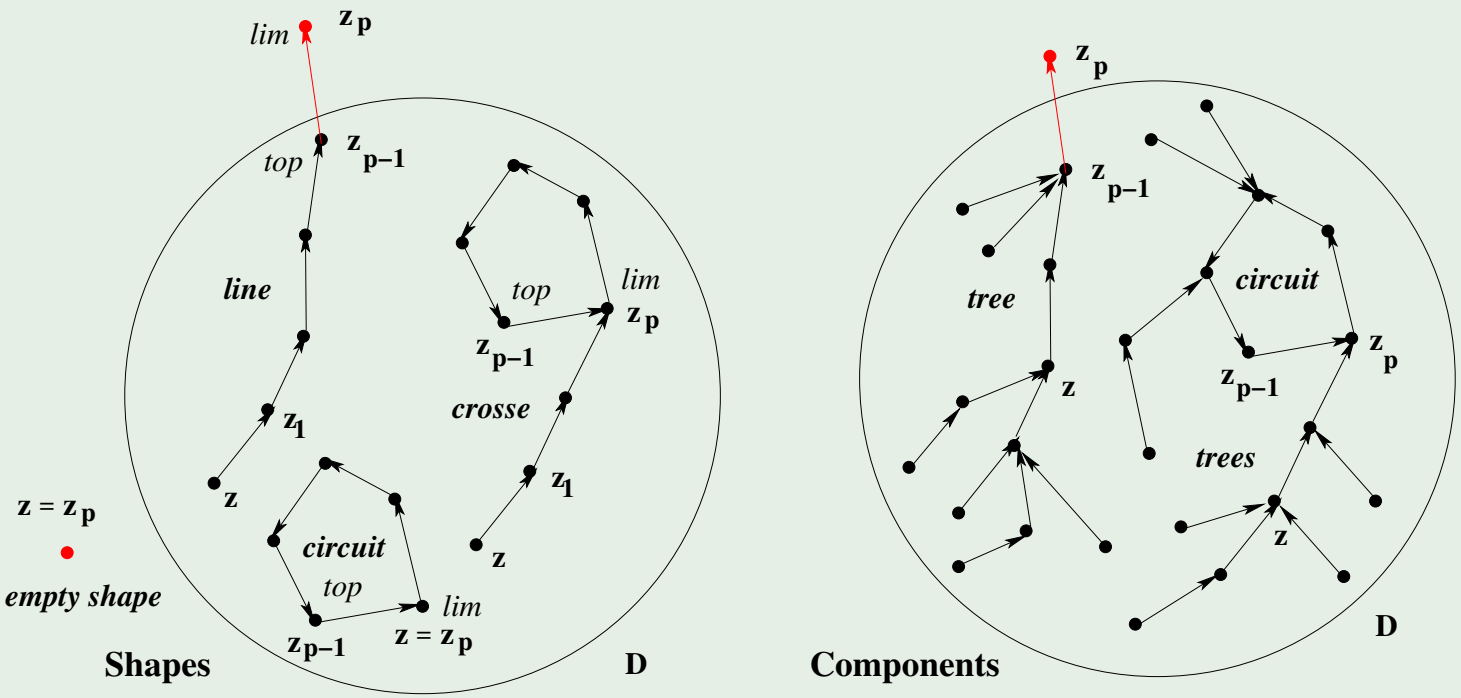
(iii) `orb z := orbs (lorb z) z` : *orbite* de `z`

(iv) `lim z := z(lorb z)` (ou `zp`) : *limite* de `z`

(v) `top z := z(lorb z-1)` : *sommet* de `z` quand `In z D`

Orbites

Exemple : Orbites (4 pos. de z) / Composantes (3 pos. de z)



Orbites

Formes d'orbites

Une *orbite* est :
(i) *vide* : quand `~In z D`
(ii) une *ligne* : quand `~In (lim z) D`, noté `inv_line z`
(iii) une *crosse* : quand `In (lim z) (orb z)`, noté `inv_crosse z`
(iv) un *circuit* : quand `lim z = z`, noté `inv_circ z`

Formes de composantes connexes

Une *composante connexe* du *graphe fonctionnel* `(D,f)` est :
(i) soit un *arbre*
(ii) soit un *circuit sur lequel des arbres sont greffés*

Orbites

Inverse, clôture

- `f_1 z` : *inverse* de `z`, quand `z` a un seul `f`-prédécesseur dans `D` ou bien l'orbite de `z` est un circuit
- `Cl f` : *clôture* de `f`, quand toutes les composantes sont des branches (linéaires) ou des circuits (`f` : *injection partielle* dans `D`)

Exemple : inversion, clôture

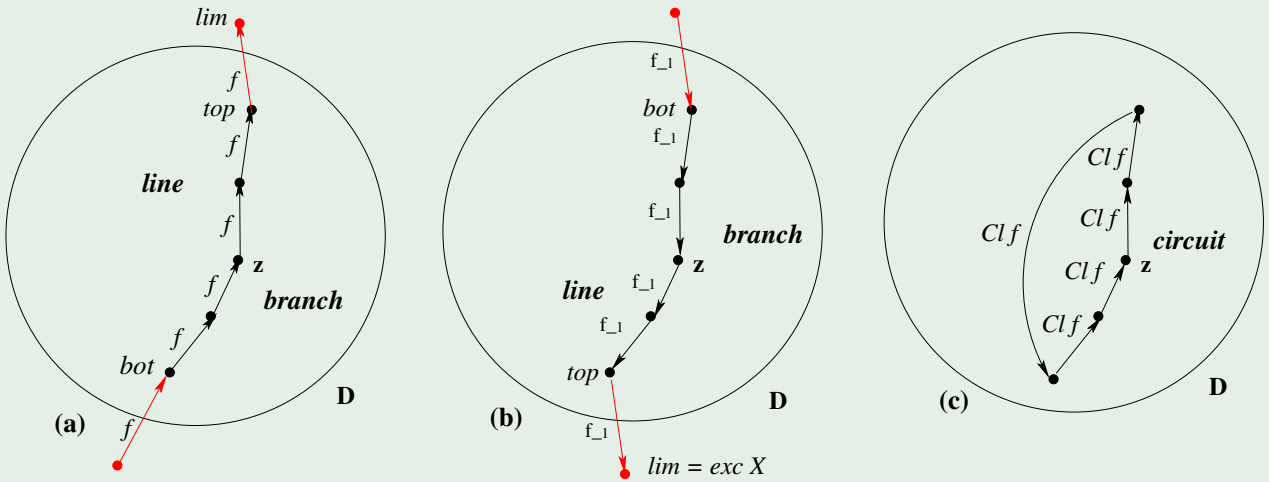


Figure: (a) Branche contenant `z` / (b) Inversion / (c) Clôture.

Orbites

Addition/suppression

Addition d'un nouvel élément `a` dans `D`, avec `~In a D`. Deux cas :
- `lim D f z <> a` : orbite de `z` préservée
- `lim D f z = a` : grands changements possibles...

Si `~In (f a) D`, la nouvelle orbite de `z` est l'ancienne complétée par `a` seulement (Fig)

Suppression d'un élément `a` de `D`. Deux cas :
- `~In a (orb D f z)` : orbite de `z` préservée
- `In a (orb D f z)` : orbite de `z` coupée en `a` (Fig)

Exemple : addition/suppression

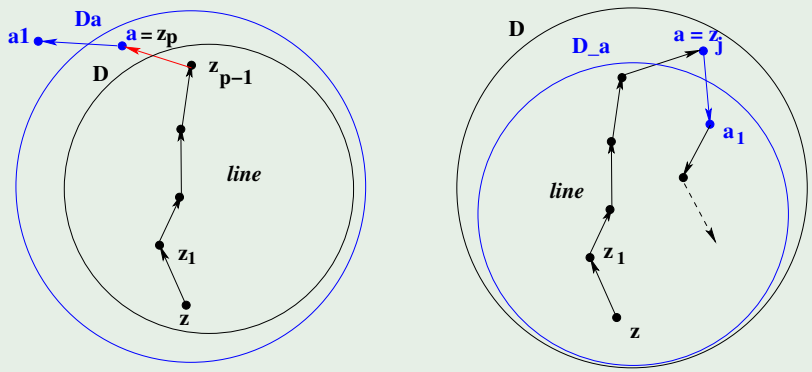


Figure: Addition (Gauche) / Suppression (Droite).

Orbites

Mutation

Une *mutation* modifie l' *image* par `f` d'un élément `u`, i.e. `f u`, en un élément, nommé `u1`, alors que tous les autres, et `D`, sont inchangés :

```
Definition Mu(f:X->X)(u u1:X)(z:X):X :=
  if eqd X u z then u1 else f z.
```

Si `~In u D`, rien d'important. Si `In u D`, deux cas pour `u1` and `u`:
- Cas A : `~In u (orb f D u1)` : l'orbite de `u1` ne change pas et la nouvelle orbite de `u` est celle de `u1` plus `u` lui-même (Fig)
- Cas B : `In u (orb f D u1)` : création d'un *nouveau circuit* (Fig)

Pour `z:X` en général, différents cas selon les positions respectives de `z`, `u1` et `u` (assez compliqué).

Exemple : mutation

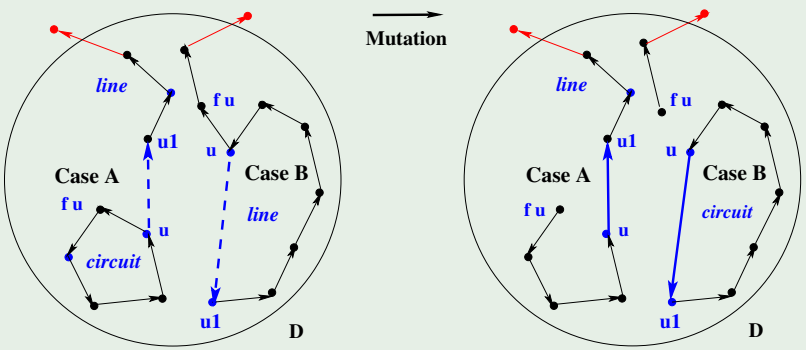


Figure: Mutation : Cas A et B.

Orbites

Transposition

Une *transposition* *échange* les images par `f` de deux éléments appartenant à des circuits, et laisse les autres intactes (on précise juste un des éléments, `u`, et sa nouvelle image, `u1`) :

```
Definition Tu(f:X->X)(D: list X)(u u1:X)(z:X):X :=
   if eqd X u z then u1
   else if eqd X (f_1 X f D u1) z then f u else f z.
```

Opération utilisable quand :

- `u` et `u1` dans le même circuit : éclatement d'orbite (*split*) (Fig)
- `u` et `u1` dans deux circuits distincts : fusion d'orbites (*merge*) (Fig)

Exemple : transposition

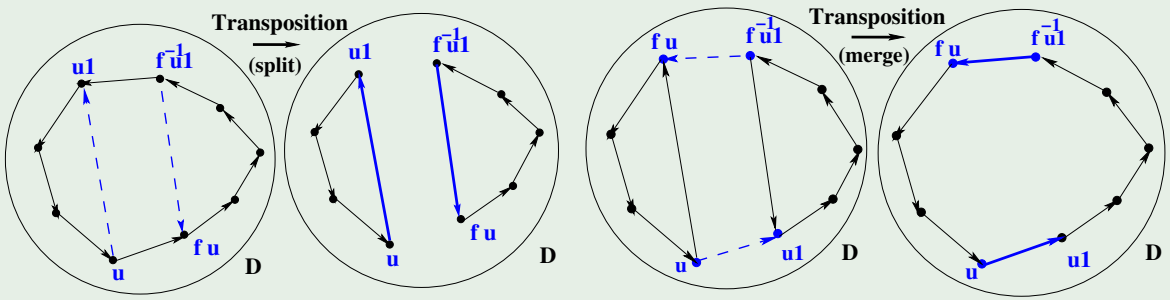


Figure: Split (Gauche) / merge (Droite).

Rappels mathématiques sur les hypercartes

Définition

Une *hypercarte combinatoire* (de dimension 2) est une structure algébrique, $M = (D, \alpha_0, \alpha_1)$, où D est un ensemble fini dont les éléments sont appelés des *brins*, et α_0, α_1 sont deux *permutations* sur D indexées par une *dimension*, 0 ou 1.

Exemple : hypercarte

D	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
α0	1	5	3	2	4	7	6	8	10	9	12	11	14	13	16	15
α1	2	3	1	7	6	5	8	4	16	11	10	13	12	15	14	9

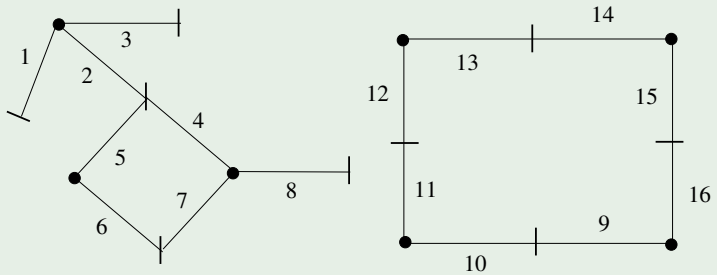


Figure: Hypercarte combinatoire plongée dans le plan.

Rappels mathématiques sur les hypercartes

Orbites d'hypercartes

- L'*arête* (resp. le *sommet*, la *face*) de z est son orbite (circulaire) par α_0 (resp. α_1, $\phi = \alpha_1^{-1} \circ \alpha_0^{-1}$) par rapport à D.
- Si t est dans la f-orbite circulaire de z, celle-ci et l'orbite de t peuvent être *identifiées modulo une permutation circulaire*, et ne compter que pour 1 lors d'un dénombrement (z et t sont dans la même composante connexe de (D, f)).

Classification

Les hypercartes sont *classées* en fonction de leurs nombres d'arêtes, sommets, faces et composantes connexes (de $(D, \{\alpha_0, \alpha_1\})$), avec les notions de *caractéristique d'Euler*, *genre* et *planarité*.

Hypercartes (constructives) en Coq

Brins, dimensions, cartes libres (free maps)

```
Definition dart := nat.
Definition nild := 0.
...
Inductive dim:Type :=
    zero : dim | one : dim.
Inductive fmap:Type :=
    V : fmap
  | I : fmap->dart->fmap
  | L : fmap->dim->dart->dart->fmap.
```

Exemple : hypercarte

```
m1 := I ( I (I (I ( I (I V 1) 2) 3) 4) 5) 6.
m2 := L (L m1 zero 4 2) zero 2 5).
m3 := L (L (L m2 one 1 2) one 2 3) one 6 5.
```

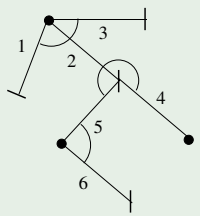


Figure: Codage partiel de l'hypercarte en exemple (orbites ouvertes).

Hypercartes en Coq

Observateurs

```
(* existence of a dart in a fmap: *)
  Fixpoint exd(m:fmap)(z:dart){struct m}: Prop :=
   match m with
      V => False
    | I m0 x => x = z \/ exd m0 z
    | L m0 _ _ _  => exd m0 z
   end.

(* k-successor: *)
  Fixpoint pA(m:fmap)(k:dim)(z:dart){struct m}: Prop :=  ...

(* k-successor's closure : "alpha" of the math. def. *)
  Fixpoint A(m:fmap)(k:dim)(z:dart){struct m}: Prop :=  ...
```

Preconditions, invariant, type des hypercartes

```
 Definition prec_I (m:fmap)(x:dart) := x <> nild /\ ~ exd m x .

 Definition prec_L (m:fmap)(k:dim)(x y:dart) :=
   exd m x /\ exd m y /\ ~succ m k x /\ ~prev m k y /\ A m k x <> y.

 Fixpoint inv_hmap(m:fmap):Prop:= match m with
     V => True
   | I m0 x => inv_hmap m0 /\ prec_I m0 x
   | L m0 k0 x y => inv_hmap m0 /\ prec_L m0 k0 x y end.
 Definition hmap:Type:= {m:fmap | inv_hmap m}.
```

Hypercartes en Coq

Propriétés "orbitales" des hypercartes (I)

Idée : étudier `pA` et `A` via les propriétés de leurs *orbites*.

```
Theorem inv_line_pA:  forall m k z, inv_hmap m ->
  inv_line dart (pA m k) (m2s m) z.
 (* Les orbites par (pA m k) restent "ouvertes" *)
Theorem A_eq_Cl: forall m k, inv_hmap m ->
  A m k = Cl dart (pA m k) (m2s m).
Theorem inv_circ_A: forall m k z, inv_hmap m ->
  inv_circ dart (A m k) (m2s m) z.
 (* Les orbites par (A m k) sont "fermees" *)
Lemma A_1_eq_f_1: forall m k z, inv_hmap m -> exd m z ->
  A_1 m k z = f_1 dart (A m k) (m2s m) z.
```

Propriétés des hypercartes (II)

- Définitions incrémentales des *nombres d'arêtes, sommets, faces, composantes connexes*.
- Définitions de *caractéristique d'Euler*, *genre* et *planarité*.
- Preuve inductive du *théorème du Genre*.
- Critère constructif de *planarité*.
- Preuve du *théorème de Jordan discret*.

Hypercartes en Coq

Opérations (conservatives) de haut niveau

- `B m k x` : *suppression du* `k`*-lien* depuis `x`
- `D m x` : *suppression du brin* (isolé) `x`
- `Shift m k x` : *glissement du trou* de la `k`-orbite de `x` juste après `x`
- `Split m k x y` : *éclatement* de la `k`-orbite de `x` en deux parties
- `Merge m k x y` : *fusion* des `k`-orbites de `x` et `y`

Exemple : éclatement/fusion pour 1-orbites

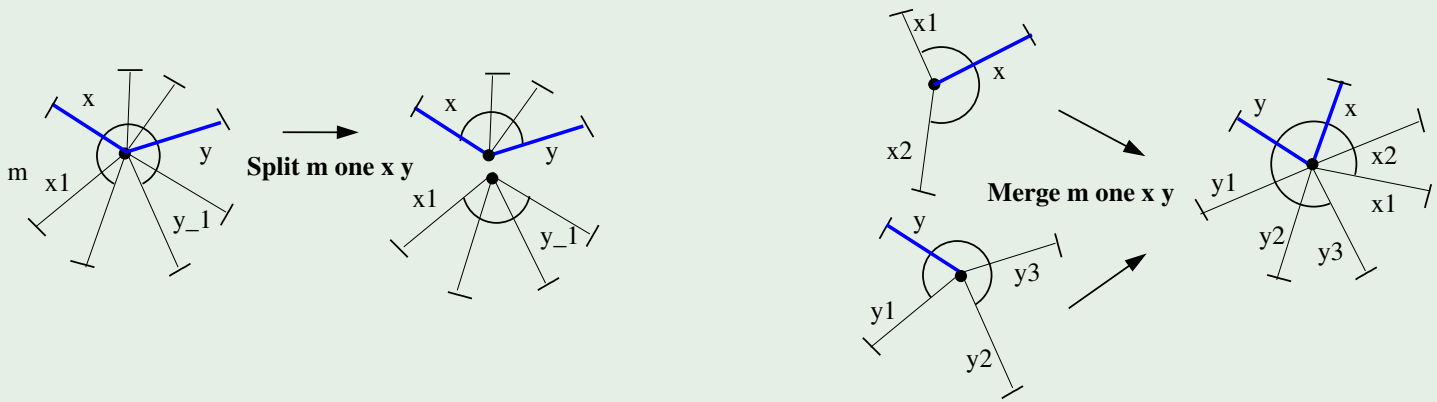


Figure: Split (Gauche) and Merge (Droite) à dimension 1.

Hypercartes en Coq

Propriétés des opérations

```
Lemma A_Split_eq_Tu: forall m k x y,
 inv_hmap m -> prec_Split m k x y ->
     A (Split m k x y) k = Tu dart (A m k) (m2s m) x y.
Lemma A_Merge_eq_Tu: forall m k x y,
 inv_hmap m -> prec_Merge m k x y ->
     A (Merge m k x y) k = Tu dart (A m k) (m2s m) x y.
```

D'où propriétés héritées des orbites générales.

Modèle général de mémoire

Hypothèses

- La mémoire est *non bornée* et les allocations sont toujours réussies
- Elle est *partitionnée* selon les *types* de données

Dans chaque partie :

- Les *adresses* potentielles sont les entiers naturels
- La *validité* d'une adresse peut être testée
- Une *adresse exception* `null` (= `0`), est toujours invalide
- Une *adresse fraîche* (invalide et non-`null`) peut toujours être engendrée par une fonction appelée `adgen`.

Idée : mimer l'allocation fournie par la macro C :

```
#define alloc(T) ((T *) malloc(sizeof(T)))
```

Modèle général de mémoire

Formalisation en Coq

```
Definition Addr := nat.
Definition null := 0.
...

Variables (T : Type) (undef:T).

Inductive Mem: Type:=
  initm : Mem
| insm : Mem -> Addr -> T -> Mem.

Fixpoint valid(M:Mem)(z:Addr):Prop := ...

Definition prec_insm M a := ~valid M a /\ a <> null.
Fixpoint inv_Mem(M:Mem): Prop :=
  match M with
    initm => True
  | insm M0 a t => inv_Mem M0 /\ prec_insm M a
 end.

Parameter adgen: Mem -> nat.
Axiom adgen_axiom: forall M,
  let a := adgen M in ~valid M a /\ a <> null.
(* Single axiom, apart from extensionality *)
```

Modèle général de mémoire

Opérations conservatives

- `alloc M` : *allocation*, renvoie `M` mis à jour et une adresse fraîche :

```
Definition alloc(M:Mem):(Mem * Addr)%type :=
  let a := adgen M in (insm M a undef, a).
```

Spécifiées inductivement :

- `load M z` : *chargement*
- `mut M z t` : *mutation*
- `free M z` : *libération*

Représentation des hypercartes

Cellules pour les brins

```
Record cell:Type:=
  mkcell { s : dim -> Addr;   (* k-successors *)
           p : dim -> Addr;   (* k-predecessors *)
           next : Addr        (* successor in the main list *)
         }.

Definition Memc := Mem cell.
```

Exemple : une cellule

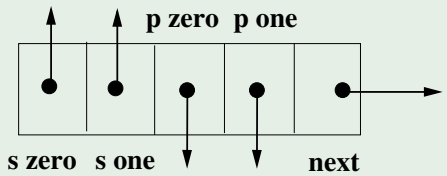


Figure: Une cellule de représentation d'hypercarte.

Représentation des hypercartes

Listes chaînées emboîtées

- une *liste chaînée principale simple* de brins
- pour chaque brin, 4 *listes chaînées circulaires* pour les `k`-liens.

```
Definition Rhmap := (Memc * Addr)%type.
```

Opérations (noms précédés de "R")

```
Definition Rnext M z := next (loadc M z).
Definition Rorb Rm := let (M, h) := Rm in
  orb Addr (Rnext M) (domc M) h.
Definition Rlim Rm := let (M, h) := Rm in
  lim Addr (Rnext M) (domc M) h.
Definition Rexd Rm z := In z (Rorb Rm).
Definition RA M k z := s (loadc M z) k.
Definition RA_1 M k z := p (loadc M z) k.
```

Représentation des hypercartes

Invariant de représentation

```
Definition inv_Rhmap1 (Rm:Rhmap) := let (M, h) := Rm in
   inv_Memc M /\ (h = null \/ In h (domc M)) /\
     lim Addr (Rnext M) (domc M) h = null.

Definition inv_Rhmap2 (Rm:Rhmap) := let (M, h) := Rm in
   forall k z, Rexd Rm z ->
      inv_circ Addr (RA M k) (Rorb Rm) z /\
        RA_1 M k z = f_1 Addr (RA M k) (Rorb Rm) z.

Definition inv_Rhmap (Rm:Rhmap) := inv_Rhmap1 Rm /\ inv_Rhmap2 Rm.
```

Conséquences

```
Lemma RA_RA_1: forall Rm k z, inv_Rhmap Rm ->
 let (M, h) := Rm in In z (Rorb Rm) ->
   RA M k (RA_1 M k z) = z.
Lemma inv_circ_RA_1: forall Rm k z, inv_Rhmap Rm ->
 let (M, h) := Rm in In z (Rorb Rm) ->
   inv_circ Addr (RA_1 M k) (Rorb Rm) z.
```

- Les listes secondaires sont organisées en *listes circulaires doublement chaînées*, chacune correspondant à une arête ou à un sommet pouvant être parcouru en sens direct ou retrograde.
- Pour un `k` et un sens fixé, elles déterminent une *partition* de la liste principale.

Opérations pour l'utilisateur (I) : hypercarte vide

Idée : fournir un jeu complet d'opérations de la spécification *non risquées* (cachant la manipulation des pointeurs).

Hypercarte vide : RV

```
Definition RV(M:Memc): Rhmap := (M, null).
```

Propriétés

```
Lemma Rexd_RV: forall M z, inv_Memc M ->
  ~Rexd (RV M) z.
```

Opérations pour l'utilisateur (II) : insertion

Insertion d'un nouveau brin isolé : RI

```
Definition RI(Rm:Rhmap):Rhmap :=
  let (M, h) := Rm in
  let (M1, x) := allocc M in
  let M2 := mutc M1 x (modnext (ficell x) h) in (M2, x).
```

Exemple : insertion d'un brin isolé

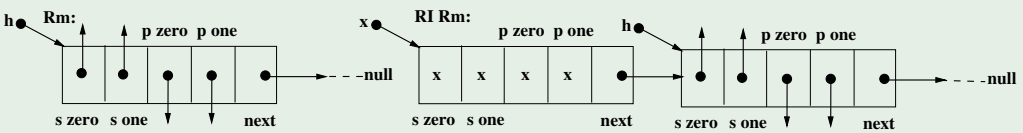


Figure: RI: insertion d'un brin dans `Rm` (Gauche) donnant `RI Rm` (Droite).

Propriétés

```
Lemma Rorb_RI: Rorb (RI Rm) = Rorb Rm ++ (x :: nil).
Lemma Rexd_RI: forall z,
  Rexd (RI Rm) z <-> Rexd Rm z \/ z = adgenc M.
Lemma RA_RI: forall k z,
  RA (fst (RI Rm)) k z = if eq_nat_dec x z then x else RA M k z.
Lemma RA_1_RI: forall k z,
  RA_1 (fst (RI Rm)) k z = if eq_nat_dec x z then x else RA_1 M k z.
```

Preuves héritées des orbites générales :

```
Lemma Rnext_M2_Mu: Rnext M2 = Mu Addr (Rnext M) z_1 (Rnext M z).
Lemma RA_M2_Mu: forall k, RA M2 k = Mu Addr (RA M k) x x.
```

Opérations pour l'utilisateur (III) : transposition

Transposition de 2 brins : RL

```
Definition RL(Rm:Rhmap)(k:dim)(x y:Addr): Rhmap :=
  let (M, h) := Rm in
  let xk := RA M k x in let y_k := RA_1 M k y in
  let M3 := mutc M x (mods (loadc M x) k y) in
  let M4 := mutc M3 y (modp (loadc M3 y) k x) in
  let M5 := mutc M4 y_k (mods (loadc M4 y_k) k xk) in
  let M6 := mutc M5 xk (modp (loadc M5 xk) k y_k) in (M6, h).
Definition prec_RL Rm k x y:= In x (Rorb Rm) /\ In y (Rorb Rm).
```

Exemple : transposition de 2 brins

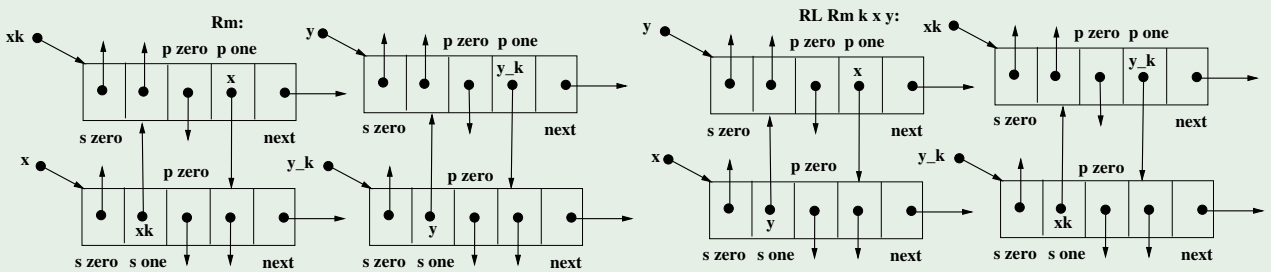


Figure: RL: Transposition de 2 brins de `Rm` (Gauche) donnant `Rm k x y` (Droite).

Opérations pour l'utilisateur (III) : transposition

Propriétés

```
Lemma Rexd_RL: forall Rm k x y z, inv_Rhmap Rm -> prec_RL Rm k x y ->
 (Rexd (RL Rm k x y) z <-> Rexd Rm z).
Lemma RA_RL_z: forall Rm k x y z, inv_Rhmap Rm -> prec_RL Rm k x y ->
  RA (fst (RL Rm k x y)) k z =
    if eqd Addr x z then y
    else if eqd Addr (RA_1 (fst Rm) k y) z then RA (fst Rm) k x
         else RA (fst Rm) k z.
Lemma RA_1_RL_z: ... similar to RA_RL_z ...
```

Preuves héritées des orbites générales par :

```
Lemma RA_M6_eq_Tu: RA M6 k = Tu Addr (RA M k) (Rorb (M,h)) x y.
```

Opérations pour l'utilisateur (IV) : suppression

Suppression d'un brin isolé : RD

```
Definition RD(Rm:Rhmap)(x:Addr)(H: inv_Rhmap1 Rm): Rhmap :=
 let (M,h) := Rm in
  if eqd Addr h null then Rm
  else if eqd Addr x null then Rm
       else if eqd Addr h x
            then let h1 := Rnext M h in
                 let M1 := freec M h in (M1, h1)
            else let x_1 := Rnext_1 Rm H x in
              if eqd Addr x_1 null then Rm
              else let M2 := mutc M x_1 (modnext (loadc M x_1) (Rnext M x)) in
                   let M3 := freec M2 x in (M3,h).
Definition prec_RD Rm x :=
  forall k, Rexd Rm x -> RA (fst Rm) k x = x /\ RA_1 (fst Rm) k x = x.
```

Exemple : suppression d'un brin isolé

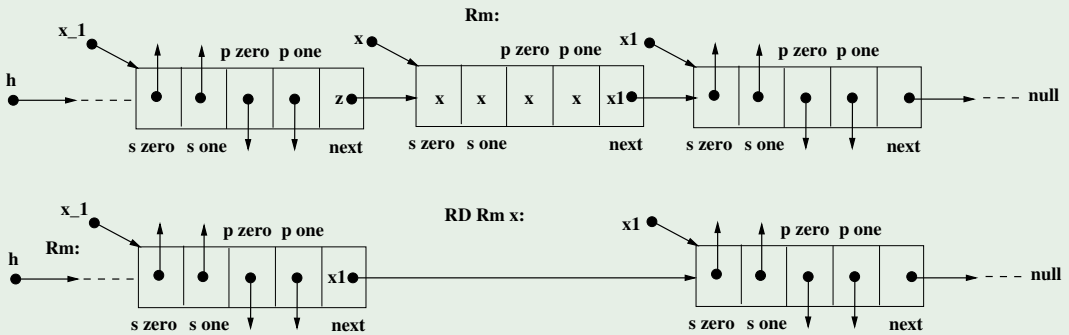


Figure: RD: Suppression d'un brin de `Rm` (Haut) donnant `RD Rm x` (Bas).

Opérations pour l'utilisateur (IV) : suppression

Propriétés

Comportement correct... Preuves héritées des orbites générales par :

```
Lemma Rnext_M2_Mu: Rnext M2 = Mu Addr (Rnext M) x_1 (Rnext M x).
Lemma Rnext_M3_Mu: Rnext M3 = Mu Addr (Rnext M2) x null.
```

Equivalence observationnelle spécification/implantation

Fonction (morphisme) d'abstraction

```
(* CRm : CRhmap Rm est un predicat inductif exprimant que Rm est
         construite exclusivement en utilisant RV, RI, RL et RD : *)
Fixpoint Abs (Rm: Rhmap) (CRm : CRhmap Rm) {struct CRm}: fmap :=
  match CRm with
    CRV M H0 => V
  | CRI Rm0 H0 => let m0 := Abs Rm0 H0 in I m0 (adgenc (fst Rm0))
  | CRL Rm0 k x y H0 H1 => let m0 := Abs Rm0 H0 in
      if expo_dec Addr (RA (fst Rm0) k) (Rorb Rm0) x y
      then if eqd Addr (A m0 k x) y then m0 else Split m0 k x y
      else Merge m0 k x y
  | CRD Rm0 x Inv H0 H1 => let m0 := Abs Rm0 H0 in D m0 x.
  end.
```

Propriétés

```
  Lemma inv_hmap_Abs: forall Rm CRm, inv_Rhmap Rm ->
   inv_hmap (Abs Rm CRm).
  Lemma exd_Abs: forall Rm CRm z, inv_Rhmap Rm ->
   (exd (Abs Rm CRm) z <-> Rexd Rm z).
  Lemma A_Abs: forall Rm CRm k z, inv_Rhmap Rm ->
   exd (Abs Rm CRm) z -> A (Abs Rm CRm) k z = RA (fst Rm) k z.
  Lemma A_1_Abs: forall Rm CRm k z, inv_Rhmap Rm ->
   exd (Abs Rm CRm) z -> A_1 (Abs Rm CRm) k z = RA_1 (fst Rm) k z.
```

Equivalence observationnelle spécification/implantation

Fonction (morphisme) de représentation

```
Fixpoint Rep (M:Memc)(m:fmap): (Rhmap * Prop)%type :=
 match m with
    V => (RV M, True)
  | I m0 x t p =>
      let (Rm0, P0) := Rep M m0 in
      (RI Rm0 t p, P0 /\ x = adgenc (fst Rm0))
  | L m0 k x y =>
      let (Rm0, P0) := Rep M m0 in (RL Rm0 k x y, P0)
end.
(* Dans (Rm,Pm) := Rep M m, Rm est une representation correcte ssi Pm :
   les brins sont les adresses generees par adgen successifs depuis M *)
```

Propriétés

```
Lemma inv_Rhmap_Rep: forall M m, inv_Memc M -> inv_hmap m ->
  let (Rm, Pm) := Rep M m in Pm -> inv_Rhmap Rm.
Lemma Rexd_Rep: forall M m z, inv_Memc M -> inv_hmap m ->
  let (Rm, Pm) := Rep M m in Pm -> (Rexd Rm z <-> exd m z).
Lemma RA_Rep: forall M m k, inv_Memc M -> inv_hmap m ->
  let (Rm, Pm) := Rep M m in Pm -> forall z, Rexd Rm z ->
  RA (fst Rm) k z = A m k z /\ RA_1 (fst Rm) k z = A_1 m k z.
```

Travaux connexes et discussion

Thèmes

- *Preuves statiques de programmes*
- *Spécifications algébriques*
- *Modèles de mémoire et de programmation*
- *Séparation et collision*
- *Spécification et implantation d'hypercartes*
- *Systèmes de preuve dédiés*

Conclusion

Bilan

- Intérêt pour des *librairies de types et opérations* avec des structures de données complexes
- Point de vue de *spécification algébrique*
- Utilisation exclusive d'une *logique d'ordre supérieur : CCI* : ni logique de Hoare, ni logique de séparation [Reynolds, O'Hearn...]
- Utilisation d'une *bibliothèque générique sur les orbites* : traitement de listes linéaires ou circulaires, simplement ou doublement chaînées, éventuellement avec emboîtement
- *Développement Coq* pour cette étude
  (avec le modèle de mémoire, mais sans les orbites) :
  9 000 lignes (60 définitions, 630 lemmes et théorèmes)

Conclusion

Perspectives

- *Généralisation des résultats sur les orbites* : fonctions multiples, pour traiter arbres, forêts et graphes
- Relations avec la *logique de séparation* : les orbites aident à poser et résoudre des problèmes de collision et séparation
- Relations avec les *plateformes de preuves (semi-)automatiques* : Why3, Frama-C...
- *Compilation* du "fragment impératif" Coq vers C
- Etudes de cas sur des *données et problèmes complexes*, notamment en algorithmique et modélisation géométriques (Exemple : diagrammes de Delaunay / Voronoi en 3D)

Annexe : programme C

Comments

- This appendix contains a C operational program for the concrete types, data structures and functions which correspond to the hypermap linked Coq representation.
- It is obtained by a direct translation where the memory is a global implicit object, memory variables are removed, addresses are pointers on cells, and `Rm` is identified to `h`.
- A run on a test game needs a simple wrapping in ad hoc types and functions to refer darts in play, e.g. by integers, and to traverse the data structures.

Programme C

Listing (I)

```
/* Programming in C the hypermap Coq representation */

#define MALLOC(t) ((t *) malloc(sizeof(t)))
#define null NULL

typedef enum {zero, one} dim;

typedef struct scell {
   struct scell * s[2];
   struct scell * p[2];
   struct scell * next;
} cell, * Addr, * Rhmap;

cell mkcell (Addr s[], Addr p[], Addr n) {
  cell c; int k;
  for(k=0;k<2;k++){c.s[k] = s[k]; c.p[k] = p[k];}
  c.next := n;
  return c;
}
```

Programme C

Listing (II)

```
cell mods(cell c, dim k, Addr m) { c.s[k] = m; return c; }

cell modp(cell c, dim k, Addr m) { c.p[k] = m; return c; }

cell modnext(cell c, Addr m) { c.next = m; return c; }

cell ficell(Addr x) {
  cell c; int k;
  for(k=0;k<2;k++){c.s[k] = c.p[k] = x;}
  c.next = null;
  return c;
}

cell initcell() {
  cell c; int k;
  for(k=0;k<2;k++){c.s[k] = c.p[k] = null;}
  c.next = null;
  return c;
}
```

Programme C

Listing (III)

```
cell load(Addr z) { return *z; }

void mut(Addr z, cell c) { *z = c;}

Addr alloc() {
   Addr x = MALLOC(cell);
   *x = initcell();
   return x;
}

/* free (z:Addr) BUILT-IN */

Addr Rnext (Addr z) { return z->next; }

Addr RA (dim k, Addr z) { return z->s[k]; }

Addr RA_1 (dim k, Addr z) { return z->p[k]; }
```

Programme C

Listing (IV)

```
Rhmap RV() { return null;}

Rhmap RI(Rhmap Rm) {
  Addr x = alloc();
  mut(x, (modnext(ficell(x), Rm)));
  return x;
 }

Rhmap RL(Rhmap Rm, dim k, Addr x, Addr y) {
  Addr xk = RA(k, x);
  Addr y_k = RA_1(k, y);
  mut(x, (mods(load(x), k, y)));
  mut(y, (modp(load(y), k, x)));
  mut(y_k, (mods(load(y_k), k, xk)));
  mut(xk, (modp(load (xk), k, y_k)));
  return Rm;
}
```

Programme C

Listing (V)

```
Addr Rnext_1(Rhmap Rm, Addr x) {
  if(Rnext(Rm) == x) return Rm;
  return Rnext_1(Rnext(Rm), x);
}

Rhmap RD(Rhmap Rm, Addr x) {
  Addr h1, x_1;
  if (Rm == null || x == null) return Rm;
  if (Rm == x)
    {
      h1 = Rnext(Rm);
      free (Rm);
      return h1;
    }
  x_1 = Rnext_1(Rm,x);
  if (x_1 == null) return Rm;
  mut(x_1, (modnext (load(x_1), Rnext(x))));
  free(x);
  return Rm;
}
```