Hypermap Specification and Linked Implementation Certification using Orbits

Jean-François Dufourd

ICUBE Laboratory, University of Strasbourg - CNRS, France

FLOC - ITP'2014, Vienna

Introduction

- *Aim:*
  - Specify libraries and certify pointer implementations
- *Use of:*
  - Algebraic datatypes
  - Calculus of Inductive Constructions (CIC) and Coq system
  - Simulation of a fragment of C
  - Orbits [Dufourd 14, soon in TCS...]:
  central in specification and implementation
- *Non-use of:*
  - Floyd-Hoare logic (and Separation logic)
- *Case study, focused on orbits:*
  - Combinatorial hypermaps [Cori 70, ..., Gonthier 08, ...]
  Applications in *Combinatorics* and *Geometric modeling*

Dedicated to George Gonthier...

Outline

Orbits in Coq

Context

- A *type* `X`, with a *decidable equality* `eqd X:X->X->Prop`
- A *total function* `f:X->X`
- A *finite subdomain* `D:list X`

(represented as a finite list without duplication)

Definitions, Notations

For any `z:X`, consider the `f`-*iterates*:

`z0 := z, z1 := f z,..., zk := f z(k-1),...`

It is proved that there is a *smallest* `p` such that `zp` is not in `D` or `zp` is already met among `z0,..., z(p-1)`.

(i) `f`-*orbit* of `z`: `orb X f D z` :=

`[z0...z(p-1)]` if `z` is in `D`, `[ ]` otherwise

(ii) *length* of `z`'s `f`-orbit: `lorb X f D z := p`

(iii) `f`-*limit* of `z`: `lim X f D z := zp`

Orbits

Orbit shapes

The *orbit* of `z:X` can be:

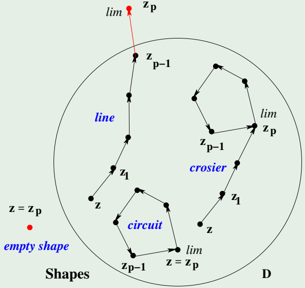
(i) *empty*: `~ In z D`

(ii) a *line*: `inv_line X f D z`

(iii) a (closed) *crosier*: `inv_crosier X f D z`

(iv) a *circuit*: `inv_circ X f D z`

Example: Orbits (4 positions of z)

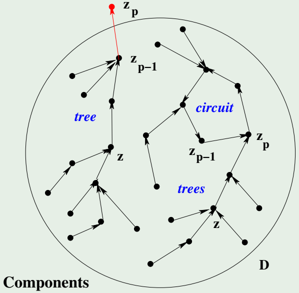


Connected components

Connected component shapes

The orbits of all `z` of `D` are separated or in collision. In fact, `(D,f)` is a *functional graph*, each *connected component* being:
(i) either a *tree*
(ii) or a *circuit on which trees are grafted*

Example: Components (2 positions of `z`*)*



Operations on orbits (I)

Definitions: Inverse, closure

- `f_1 X f D z`: *inverse* of `f` at `z`, when `z` has only one `f`-predecessor in `D`
- `Cl X f D`: *closure* of `f`, when all components are (linear) branches or circuits (`f` : *partial injection* in `D`)

Example: Inversion, closure

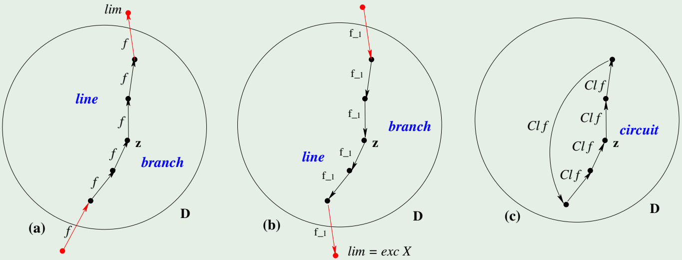


Figure : (a) Branch containing `z` / (b) Inversion / (c) Closure.

Operations on orbits (II)

Addition/deletion

Addition of a new element `a` in `D`, when `~ In a D`.
Deletion of an element `a` from `D`.

Example: Addition/deletion

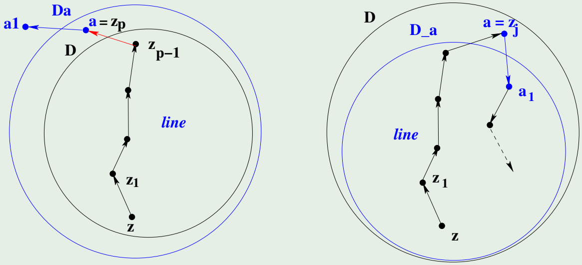


Figure : Addition (Left) / Deletion (Right).

Operations on orbits (III)

Mutation

A *mutation* modifies the `f`-*image* `f u` of an element `u` into an element `u1` while all the other images do not change:

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

Example: Mutation

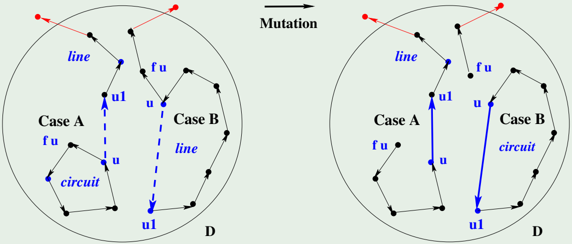


Figure : Mutation: Cases A and B.

Operations on orbits (IV)

Transposition

A *transposition* *exchanges* the `f`-images of two elements in circuits and do not change the others (only one element `u` and its new image `u1` are precised):

```
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.
```

When:
- `u` and `u1` are in the same circuit: *split* into two circuits
- `u` et `u1` are in two circuits: *merge* into one circuit

Example: Transposition

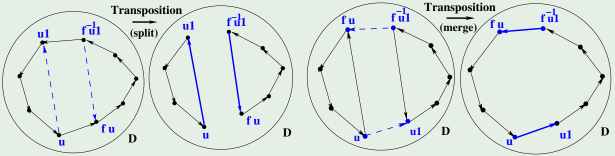


Figure : Split (Left) / Merge (Right).

Hypermaps in mathematics

Definition

A *combinatorial (2-dimensional) hypermap* is an algebraic structure, $M = (D, \alpha_0, \alpha_1)$, where:
- D is a finite set, the elements are called *darts*,
- and α_0, α_1 are two *permutations* on D indexed by a *dimension*, 0 or 1.

Example: hypermap

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

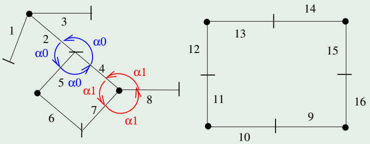


Figure : Hypermap embedded in the plane
(dart embedding: a Jordan arc beginning by a bullet, ending by a small strike).

Mathematical recalls

Orbits/components of hypermaps

The *edge* (resp. *vertex*, *face*, *component*) of z is its connected component in the graph (D, α_0) (resp. (D, α_1), $(D, \alpha_1^{-1} \circ \alpha_0^{-1})$, $(D, \{\alpha_0 \cup \alpha_1\})$)
(roughly: edge = small strike, vertex = bullet...).

Classification

The hypermaps are *classified* according to their numbers of edges, vertices, faces and components, thanks to the notions of *Euler characteristic*, *genus* and *planarity*.

(Constructive) Hypermaps in Coq (I)

Darts, dimensions, hypermaps

```
(* Type of darts and exception: *)
  Definition dart := nat.
  Definition nild := 0.
  ...
(* Type of dimensions: *)
  Inductive dim:Type :=
      zero : dim | one : dim.

(* Type of hypermaps: *)
  Inductive hmap:Type :=
      V : hmap                           (* Void (empty) hmap *)
    | I : hmap->dart->hmap               (* Insertion of a dart *)
    | L : hmap->dim->dart->dart->hmap.   (* k-Linking from a dart
                                            to another *)
```

Example: Hypermap

```
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.
```

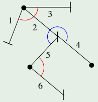


Figure : Partial coding of the example hypermap
(The 0- and 1-orbits stay "open" in the specification).

Hypermaps in Coq (II)

Free map observers (inductively defined)

- *Existence* of a dart `z` in the hmap `m`: `exd m z`
- `k`-*successor* of `z` in `m`: `pA m k z`
returns `nild` when there is no `k`-link from `z`
- `k`-*successor* of `z` *in the closure* of `pA m k`: `A m k z`
Note: two successor notions which are useful in the specification

Example: Zoom on an edge of hypermap

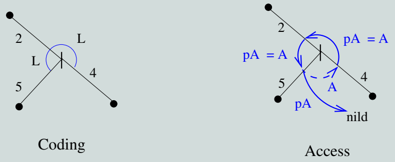


Figure : An edge: open for pA, closed for A.

Preconditions, invariant of hypermaps

- *Preconditions* on `I` and `L` impose that edges and vertices remain open for `pA`
- An *invariant* of hypermaps `inv_hmap m` derives.

Hypermaps in Coq (III)

Orbital properties of hypermaps

Idea: studying `pA` et `A` by the properties of their *orbits*:
- The `pA`-orbits stay (open) *lines*
- The components w.r.t. `pA` are *branches*
- `A` is really the *closure* of `pA`
- The `A`-orbits are (closed) *circuits*
- The same for the inverses `pA_1` and `A_1`

Other properties of the hypermaps

All this leads to fundamental results on discrete topology:
- Incremental definitions of *numbers of edges, vertices, faces, components, Euler characteristic, genus* and *planarity*
- An inductive proof of the *Genus theorem*
- Constructive criteria of *planarity*
- A proof of the *discrete Jordan curve theorem*

Formalization of Memory (I)

Addresses

- The potential *addresses* are the natural numbers
- There is an *exception address* `null` (= `0`)

```
(* Address type: *)
  Definition Addr := nat.

(* Exception: *)
  Definition null := 0.
```

Memories/Validity

- A memory is *non-bounded* and the allocations always succed
- It is *partitioned* according to the *datatypes*

```
(* Contexte: *)
  Variables (T:Type) (undef:T).

(* Memory type: *)
  Inductive Mem:Type:=
    initm : Mem                  (* empty memory *)
  | insm : Mem->Addr->T->Mem.    (* insertion of (address, value) *)

(* Validity domain: *)
  Fixpoint dom(M:Mem)(z:Addr):list Addr := ...

(* From where a precondition on insm and an invariant inv_Mem... *)
```

Formalization of Memory (II)

Address generation

- A *fresh address* (invalid and non-`null`) can always be generated by a function we call `adgen`:

```
Parameter adgen: Mem->Addr.
```

Conservative memory operations

- *allocation*: `alloc M` returns `M` updated and a fresh address:

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

Inductively specified:

- *loading*: `load M z`
- *mutation*: `mut M z t`
- *releasing*: `free M z`

Hypermap Representation (I)

Cells for the darts, memories of dart cells

```
(* Dart cell type: *)
  Record cell:Type:=
    mkcell { s : dim->Addr;   (* "array" of 2 k-successors *)
             p : dim->Addr;   (* "array" of 2 k-predecessors *)
             next : Addr      (* successor *)
           }.

(* Type of cell memories: *)
  Definition Memc := Mem cell.
```

Example: A dart cell

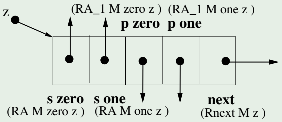


Figure : Cell of a hypermap representation in a memory M.

Hypermap Representation (II)

Hypermap representation

Any hypermap representation `Rm` is a pair composed of:
- a *cell memory* `M:Memc`
- a *pointer* (*head* of a main list) `h:Addr`

```
(* Type of hypermap representations: *)
  Definition Rhmap := (Memc * Addr)%type.
```

Observers (on Rhmap, names are prefixed by "R"; on Memc, names are suffixed by "c")

```
(* Observers of the main list: *)
  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.

(* Observers of hypermap: *)
  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.
  ...
(* Note that the representation of pA is useless...)
```

Hypermap Representation (III)

Invariant of representation

For any hypermap representation `Rm`, some features are required:
(1) A *main singly-linked list* of darts representations: a *line* with valid darts and a null *limit*:

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

(2) For each dart, 4 *circular singly-linked lists* for the `k`-links: each one is a *circuit* with darts in the main list, and `RA_1 M k` is always the *inverse* of `RA M k`:

```
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.
```

User update operations (I): Empty hypermap

Ideas:

- providing a set of specified operations which must be: *conservative* (w.r.t. invariants), *minimal, complete, ready-to-assemble, safe* (hiding pointer manipulations).
- simulating the C language

Empty hypermap: RV

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

Properties

Correct behavior w.r.t. the observer `Rexd`:

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

User update operations (II): Dart insertion

Insertion of a new isolated dart: 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).
```

Example: Insertion of an isolated dart

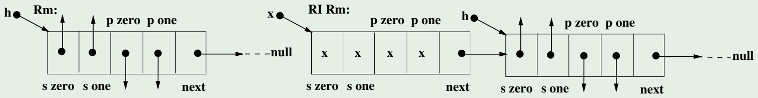


Figure : RI: insertion of a dart in `Rm` (Left) giving `RI Rm` (Right).

Properties

- *Correct observational behavior w.r.t.* `Rexd`, `RA`, `RA_1`, e.g.:
For any `k`, the new dart is a fixpoint w.r.t. `RA M2 k` and `RA_1 M2 k`
- *Proofs inherited from the general orbits: addition and mutation properties*

User update operations (III): Transposition

Transposition of two darts at dimension k: 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 of two darts x and y at dimension 1

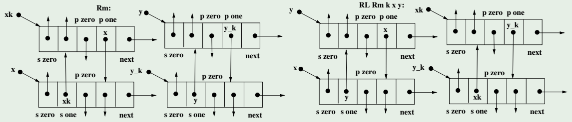


Figure : RL: Transposition in `Rm` (Left) giving `RL Rm one x y` (Right).

Properties

- *Correct observational behavior w.r.t.* `Rexd`, `RA`, `RA_1`, e.g.:
`y` is the new `k`-successor of `x`: `RL` *splits* or *merges* (circular) k-orbits w.r.t. `RA M k` and `RA_1 M k`
- *Proofs inherited from the general orbits: transposition properties*

User update operations (IV): Dart deletion

Deletion of an isolated dart: 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.
```

Example: Deletion of an isolated dart

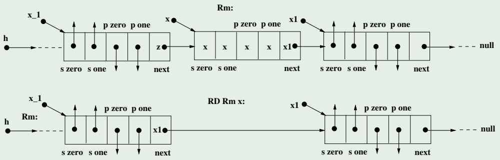


Figure : RD: Deletion of a dart of `Rm` (Top) giving `RD Rm x` (Bottom).

Properties

Correct behavior w.r.t. the observers `Rexd`, `RA`, `RA_1`, *with proofs inherited from the general orbits: mutation and deletion properties*

Observational Equivalence (I)

Abstraction function (morphism): Abs

`Abs` sends any hypermap representation which is *built by using* `RV`, `RI`, `RL`, `RD` onto an abstract hypermap which is *built by using* `V`, `I`, `L`

Properties

`Abs` *correctly carries* the observations by `Rexd`, `RA`, `RA_1` onto the observations by `exd`, `A`, `A_1`.

Observational Equivalence (II)

Representation function (morphism): Rep

`Rep` sends any abstract hypermap which is *built by using* `V, I, L` with *darts generated by successive uses of* `adgen` onto a hypermap representation which is *built by using* `RV, RI, RL`

Properties

`Rep` *correctly carries* the observations by `exd, A, A_1` onto the observations by `Rexd, RA, RA_1`.

Program in 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.

C Program

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;
}
```

C Program

Listing (II)

```
(* CRm:CRhmap Rm is an inductive predicate stating that Rm is
      constructed exclusively by using RV, RI, RL and RD: *)

Fixpoint Abs(Rm: Rhmap)(CRm:CRhmap Rm) {struct CRm}: fmap :=
  match CRm with ...

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;
}
```

C Program

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]; }
```

C Program

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;
}
```

C Program

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;
}
```

Related Work

Topics

- *Static proofs of programs*
  Floyd, Hoare, Reynolds, O'Hearn...
- *Inductive types and algebraic specifications*
  Bornat, Mehta-Nipkow, Marché, Conway-Barrett, Berdine et al..., Guttag, Goguen, Wirsing...
- *Models of memory and of programming*
  Leroy-Blazy, Chlipala...
- *Separation and collision*
  Burstall, Bornat, Reynolds, O'Hearn, Enea et al...
- *Specification and implementation of hypermaps*
  Cori, Gonthier, Dufourd, Bertrand, Bertot...
- *Dedicated proof systems*
  Malecha-Morrisett, Chlipala et al., Filliâtre...

Conclusion

Summary

- Interest for *libraries* with complex data structures
- Point of view of an *algebraic specifier*
- Exclusive use of a *higher-order logic (CIC)*: no Hoare logic, no Separation logic
- Intensive use of a *generic orbit library*: to handle arrays, singly- or doubly-linked lists, linear or circular, possibly nested
- *Coq development* for this study: 9,000 lines
  (with the memory model, but without the orbits,
  60 definitions, 630 lemmas and theorems)

Conclusion

Future work

- Generalize *orbits* to *multiple functions*: to deal with trees, forests and general graphs
- Connect *orbits* with *Separation logic*: orbits help to state and solve collision and separation problems
- Connect *orbits* with *proof platforms*: Why3, Frama-C, Ynot, Bedrock...
- *Compile* the "imperative Coq fragment" to C
- Develop other case studies with *complex data and algorithms*, particularly in computational geometry (Example: Delaunay / Voronoi diagrams in 3D)

Thank you for your attention!