Hypermap Specification and Certified Linked Implementation using Orbits

Jean-François Dufourd *

ICUBE, Université de Strasbourg et CNRS,
Pôle API, Boulevard S. Brant, BP 10413, 67412 Illkirch, France - jfd@unistra.fr

Abstract. We propose a revised constructive specification and a certified hierarchized linked implementation of combinatorial hypermaps using a general notion of orbit. Combinatorial hypermaps help to prove theorems in algebraic topology and to develop algorithms in computational geometry. Orbits unify the presentation at conceptual and concrete levels and reduce the proof effort. All the development is formalized and verified in the Coq proof assistant. The implementation is easily proved observationally equivalent to the specification and translated in C language. Our method is transferable to a great class of algebraic specifications implemented into complex data structures with hierarchized linear, circular or symmetric linked lists, and pointer arrays.

1 Introduction

We propose a revised constructive specification and a certified hierarchized linked implementation of 2-dimensional combinatorial hypermaps using a general notion of orbit. *Combinatorial hypermaps* [8] are used to algebraically describe meshed topologies at any dimension. They have been formalized to interactively prove great mathematical results, e.g. the famous Four-Colour theorem [20] or the discrete Jordan curve theorem [10], with the help of a proof assistant. They are also at the root of the certification of functional algorithms in computational geometry, e.g. Delaunay triangulation [14]. Once implemented, hypermaps (or derivatives) are a basic data structure in geometric libraries, e.g. Topofil [2] or CGAL [27]. *Orbits*, whose formal study is presented in [12], allow us to deal with trajectories, at conceptual and concrete levels. A precise correspondence between hypermaps and their linked implementation remained a challenge which we have decided to address. The novalties of this work are:
- An entire development formalized and verified in the *Coq proof assistant* [1]. Nothing is added to its higher-order calculus, except the *axiom of extensionality* and another one for *address generation* of new allocated memory cells. The first says that two functions are equal if they are equal at any point, and the second that an address generated for a block allocation is necessarily fresh and non-null;
- An extensive use of *orbits*, which nicely unifies and simplifies the presentation of specifications and linked implementations and reduces the proof effort;

* This work was supported in part by the French ANR project GALAPAGOS.

- An intricate pointer implementation in a general simple memory model, and concrete operations described in a functional form which is easy to translate in a "true" programming language, e.g. in C;
- A proof of observational equivalence between specification and pointer implementation thank to morphisms.

The underlying method is transferable to a great class of complex data structures with hierarchized linear, circular or symmetric lists, and pointer arrays. That is greatly due to the unification provided by orbits. In Sect. 2, we recall the orbit formalization. In Sect. 3 and 4, we present and specify the hypermaps. In Sect. 5, we formalize the general memory model, then we describe in Sect. 6 the linked implementation. In Sect. 7, we prove the observational equivalence between specification and implementation. We discuss related work in Sect. 8 and conclude in Sect. 9. The whole formalization process is described but the proof scripts are out of the scope of this article. The Coq development, including ref. [12] and orbit library files, may be downloaded [11].

2 Orbits for functions in finite domains

General definitions and properties [12] In Coq, all objects are strongly typed, `Prop` is the type of propositions, and `Type` can be viewed as the "type of types" [1]. We work in a *context* composed of: `X:Type`, any type whose built-in equality `=` is equipped with `eqd X`, a *decision function*, `exc X:X`, an element chosen as *exception*, `f:X -> X`, any *total function* on `X`, and `D`, a *finite* (sub)domain of `X` never containing `exc X`. For technical reasons, `D` is described as a finite list of type `list X` with *no repetitive element*. We write `In z l` when `z` occurs in the list `l`, `nil` for the empty list, and `~` for `not`. For any `z:X` and `k >= 0`, we consider `zk := Iter f k z`, the `k`-*th iterate* of `z` by `f` (with `z0 := z`). Since `D` is finite, during the iteration process calculating `zk`, a time necessarily comes where `zk` goes outside `D` or encounters an iterate already met [12].

Definition 1. *(*Orbital sequence, length, orbit, limit, top*)*
(i) The orbital sequence *of* `z` *by* `f` *at the order* `k >= 0`*, denoted by* `orbs k z`*, is the list containing* `z(k-1), ..., z1, z0`*, written from first to last elements.*
(ii) The length *of* `z`*'s orbit by* `f` *w.r.t* `D`*, denoted by* `lorb z`*, is the least integer* `p` *such that* `~ In zp D` *or* `In zp (orbs p z)`*.*
(iii) The orbit *of* `z` *by* `f` *w.r.t.* `D` *is* `orbs (lorb z) z`*, in short* `orb z`*.*
(iv) The limit *of* `z` *by* `f` *w.r.t.* `D`*, is* `zp` *−* *or* `z(lorb z)` *−, in short* `lim z`*.*
(v) When `In z D`*, the* top *of* `z` *by* `f` *w.r.t.* `D` *is* `z(lorb z - 1)`*, in short* `top z`*.*

So, an orbit is a bounded list *without repetition*, possibly empty, which can be viewed as a finite set when it is more convenient. Necessarily the *shape* of `z`'s orbit (Fig. 1(Left)) is: *(i) empty* when `~ In z D`; *(ii)* a *line* when `~ In (lim z) D`, what is denoted by the predicate `inv_line z`; *(iii)* a *crosier* when `In (lim z) (orb z)` what is denoted by `inv_crosier z`; *(iv)* a *circuit* when `lim z = z`,

what is denoted by `inv_circ z`. An empty orbit is a line and a circuit as well, and a circuit is a crosier, since `lim z = z` entails `In (lim z) (orb z)`. The

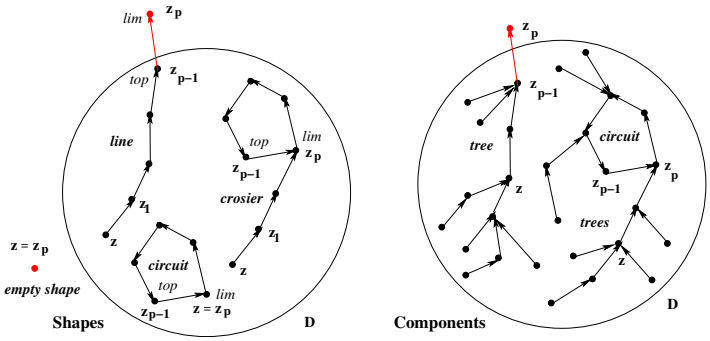


Fig. 1. Orbit Shapes (for 4 positions of z) / Components (for 2 positions of z).

existence of an *orbital path* from `z` to `t`, i.e. the fact that `t` is in `z`'s orbit, is written `expo z t`, where `expo` is a binary *reflexive transitive* relation, which is *symmetric* only if `z`'s shape is a circuit. A lot of lemmas express the variation of `lorb`, `orb` and `lim` along `z`'s orbit, depending on its shape [11]. In fact, `(D, f)` forms a *functional graph* whose *connected components* are *trees* or *circuits with grafted trees* in `D`. Fig. 1(Right) shows components for two positions of `z` in `D`.

When the orbit of `z` is a non-empty circuit or `z` has exactly one `f`-predecessor in `D`, the definition of an *inverse* for `f`, denoted by `f_1 z`, is immediate (see [11]). That is the case for any `z` having an `f`-predecessor in `D` when `f` is *partially injective* w.r.t. `D` in the following sense (Here, `->` denotes the implication):

```
forall z t, In z D -> In t D -> f z <> exc X -> f z = f t -> z = t
```

That is the usual injection characterization for `f`, but only with `f z <> exc X` to fully capture the features of linked lists in memories. In this case, the orbit shapes are only lines and circuits, and the connected components only (linear) *branches* and circuits. The branch of `z` is obtained by prolongation of `z`'s orbit with the `f`-ancestors of `z` in `D` (Fig. 2(a)). When `f` is a partial injection, its inverse `f_1` enjoys expected properties, e.g. `f_1 (f z) = f (f_1 z) = z` and orbit shape similarities with `f` [11]. Fig. 2(b) shows the *inversion* of a branch connected component (Fig. 2(a)) of `z`. The previous notions are given in a *context*, in fact a *Coq section*, where `X`, `f` and `D` are *variables*. Outside the section, each notion is parameterized by `X`, `f` and `D`. For instance, `lorb z` becomes `lorb X f D z` and is usable *for any type, function and domain*. This is necessary when `f` or `D` are changing as in what follows.

So, when `f` is *partially injective*, if we want to work only with circuits, we can *close* all the "open orbits", in fact the branches. This operation (Fig. 2(c)), which modifies `f` but not `D`, uses `top` (Sect. 2) and `bot` (Fig. 2(a,b)):

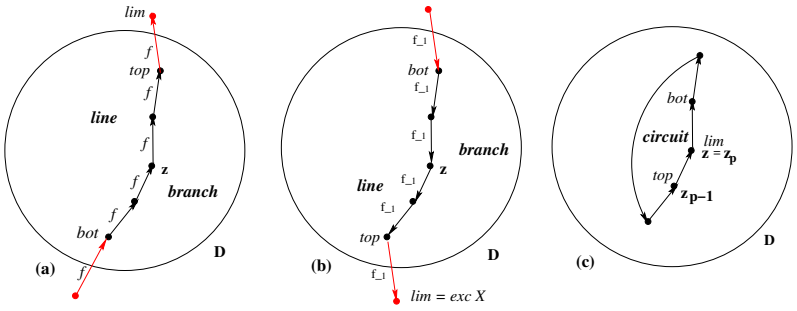


Fig. 2. (a) Branch containing `z` / (b) Inversion / (c) Closure.

```
Definition bot f D z := top (f_1 X f D) D z.
Definition Cl f D z := if In_dec X z D then if In_dec X (f z) D then f z else bot f D z else z.
```

In `Cl f D`, all the orbits are circuits and the reachability `expo` is symmetrically obtained from this for `f`. Now, we outline orbit updating operations.

• **Addition/deletion** An *addition* inserts a new element `a` in `D`, while the (total) function `f:X -> X` remains the same. We require that `~ In a D`, we pose `Da := a :: D` and `a1 := f a`. Regarding the orbit variation, for any `z:X`, two cases arise: *(i)* When `lim D f z <> a`, the orbit of `z` is entirely preserved; *(ii)* Otherwise, great changes can occur (See [12]). But, if we suppose that `~ In a1 D` (Fig. 3(Left)), which corresponds to most practical cases, `z`'s new orbit is the previous one completed by `a` only. The "inverse" operation is the *deletion* of an

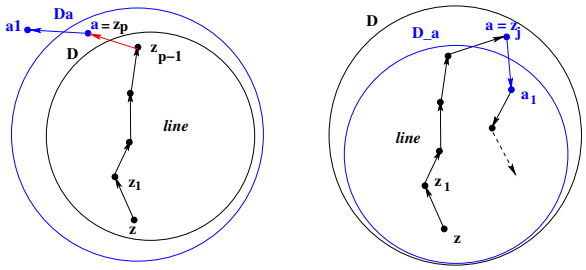


Fig. 3. Addition (Left)/Deletion (Right) effect on an orbit.

element `a` from `D`. Regarding the orbit variation for any `z:X`, two cases arise: *(i)* When `In a (orb D f z)`, `z`'s orbit is cut into a *line* with limit `a` (Fig. 3(Right), where `D_a:= remove D a`); *(ii)* Otherwise it is entirely preserved.

• **Mutation** A *mutation* modifies the *image* by `f` of an element `u`, i.e. `f u`,

into an element, named `u1`, while all the other elements, and `D`, are unchanged. The new function, named `Mu f u u1`, is defined by:

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

If `u` is not in `D` nothing important occurs. If u is in `D`, two cases arise to determine the new orbits of `u1` and `u` (Fig. 4): *(i)* When `~ In u (orb f D u1)`, the orbit of `u1` does not change and the new orbit of `u` is this of `u1` plus `u` itself (Fig. 4(Case A)); *(ii)* Otherwise, the mutation closes a *new circuit* which, say, starts from `u1`, goes to `u` by the old path, then goes back to `u1` in one step (Fig. 4(Case B)). Then, the new orbit of any `z:X` can be obtained similarly. Different cases arise depending on the respective positions of `z`, `u1` and `u` (See [12]).

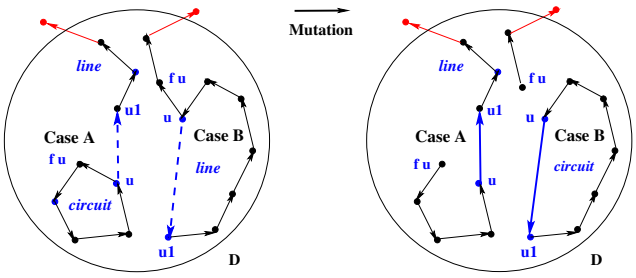


Fig. 4. Mutation: Cases A and B.

• **Transposition** A *transposition* consists in *exchanging* the images by `f` of two elements which must belong to circuits. In fact, just one element, `u`, and its *new successor*, `u1`, are provided in the transposition, whose definition is:

```
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.
```

The definition also says that the new successor of `f_1 X f D u1`, i.e. `u1`'s predecessor, is `f u`, i.e. the old `u`'s successor, and that nothing is changed for the other elements (Fig. 5). In the case where `u1 = f u`, the transposition has no effect. This operation is usable if `u` and `u1` are in the same circuit or not (Fig. 5, Left and Right). Intuitively, in the first case, the unique circuit is *split* into two circuits, and, in the second case, the two circuits are *merged* into a unique one. That is the intuition, but the formal proof of these facts is far from being easy. Of course, for any `z`, it is proved that `z`'s orbit (and connected components) are not modified by `Tu` if it contains neither `u` nor `f_1 X f D u1`.

3 Combinatorial hypermaps

Combinatorial hypermaps [8] describe space subdivision topologies at any dimension. We focus to the dimension 2, and surface subdivisions, like in [20, 10].

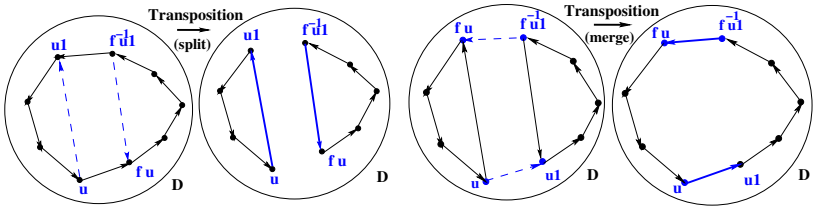


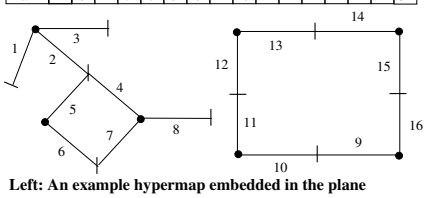
Fig. 5. Split (Left) and merge (Right).

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

Fig. 6(Left) shows a hypermap with $D = \{1, ..., 16\}$ and α_0, α_1 given by a table. It is *embedded* in the plane, its darts being represented by half-Jordan arcs (labeled by the darts) which are oriented from a bullet to a small transverse stroke. It z is a dart, α_0 z shares z's stroke, α_1 z shares z's bullet. A hypermap describes surface subdivision topologies, the cells of which can be defined by orbits, where $X = dart$, the type of the darts, and $D \subset dart$ in Def. 1.

Definition 3. *(*Hypermap orbits*)*
Let $M = (D, \alpha_0, \alpha_1)$ be a hypermap and $z \in D$ any dart in M. The edge *(resp.* vertex*,* face*) of z is the orbit of z by α_0 (resp. α_1, $\phi = \alpha_1^{-1} \circ \alpha_0^{-1}$) w.r.t. D.*

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



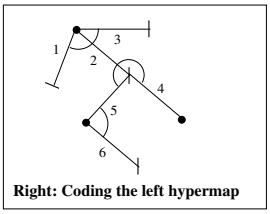


Fig. 6. 2-combinatorial hypermap and partial coding in Coq.

Since α_0, α_1 and ϕ are bijections on D, then, for any $z \in D$, the orbits of z by them w.r.t. D are *circuits*. It is the same for their inverses – α_0^{-1}, α_1^{-1} and ϕ^{-1} – which are also defined over all D. So, in this cyclic case, if t is in z's orbit by f, t's orbit and z's orbit may be identified into a *unique circuit*, defined *modulo* a cyclic permutation. So, the common orbit only counts for 1 in the number of

orbits by f. In fact, these cyclic orbits modulo permutation correspond exactly to the *connected components* generated by f in D. Cells and connected components (w.r.t. $\{\alpha_0, \alpha_1\}$) can be counted to classify hypermaps according to their *Euler characteristic*, *genus* and *planarity* [9, 12].

4 Hypermap Coq formalization

Preliminaries For simplicity, `dart` is the Coq library type `nat` of the natural numbers. The `eq_dart_dec` decision function of `dart` equality is `eq_nat_dec`, the decision function of `nat` equality, and the `dart` exception is `nild := 0`. Then, the *dimensions* are the two constants `zero` and `one` of the `dim` *enumerated* type, a simple case of *inductive* type.

Free maps As in [9, 10, 14], the 2-dimensional combinatorial hypermap specification begins with the inductive definition of a type, called `fmap`, of *free maps*, i.e. without any constraint:

```
Inductive fmap:Type :=
  V : fmap | I : fmap->dart->fmap | L : fmap->dim->dart->dart->fmap.
```

It has three *constructors*: `V`, for the *empty* − or *void* − free map; `I m x`, for the *insertion* in the free map `m` of an *isolated* dart `x`; and `L m k x y`, for the *linking* at dimension `k` of the dart `x` to the dart `y`: `y` becomes the `k`-successor of `x`, and `x` the `k`-predecessor of `y`. Any hypermap can be built by using these constructors, i.e. viewed as a *term* combining `V`, `I` and `L`. For instance, the six-darts part `m2` (Fig. 6(Right), with `k`-links by `L` being symbolized by small circle arcs) of the hypermap of Fig. 6(Left) is built by:

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

Observers Some *observers* (or *selectors*) can be easily defined on `fmap`. So, the *existence* in a free map `m` of a dart `z` is a predicate defined by structural induction on `fmap` by (`True` (resp. `False`) is the predicate always (resp. never) satisfied):

```
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.
```

The `k`-*successor*, `pA m k z`, of any dart `z` in `m` is similarly defined. It is `nild` if `m` is empty or contains no `k`-link from `z`. A similar definition is written for `pA_1 m k z`, the `k`-*predecessor*. To avoid returning `nild` in case of exception, `A` and `A_1`, *closures* of `pA` and `pA_1`, are defined in a mutual recursive way ([11]). They exactly simulate the expected behavior of the α_k permutations (Def. 2).

Hypermaps To build *(well-formed) hypermaps*, `I` and `L` must be used only when the following preconditions are satisfied. Then, the *invariant* `inv_hmap m`, inductively defined on `m`, completely characterizes the *hypermaps* [11]:

```
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 /\ pA m k x = nild /\ pA_1 m k y = nild /\ 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.
```

When `m` is a well-formed hypermap, the last constraint of `prec_L`, `A m k x <> y`, entails that, for any `z`, the orbit for `pA m k` and `pA_1 m k` are *never closed*, i.e. are never circuits, and always remain *lines* (Sect. 2). For instance, our `m2` example respects these preconditions and satisfies `inv_hmap m2`, so the circle arcs in Fig. 6(Right) do not form full circles. This choice is motivated by *normal form* considerations allowing inductive proofs of topological results [9, 10].

Hypermap properties When `inv_hmap m` is satisfied, it is proved that `A m k` and `A_1 m k` are *inverse bijective* operations. Then, considering the function `m2s m` which sends the free map `m` into its support "set" − or instead finite list, it is proved that `pA m k` and `A m k` are *partial injections* on `m2s m`. So, for any dart `z`, the orbit of `z` for `pA m k` w.r.t. `m2s m` always is a *line* (ended by `nild`). It is proved that `A m k` is really the *closure* of `pA m k` in the orbit meaning (Sect. 2), so the orbits for `A m k` and `A_1 m k` are circular:

```
Theorem inv_line_pA:  forall m k z, inv_hmap m -> inv_line dart (pA m k) (m2s m) z.
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.
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.
```

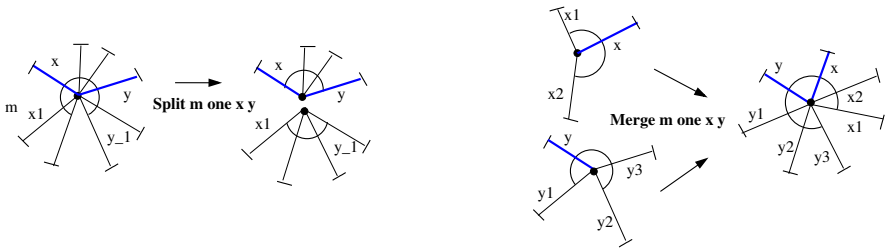


Fig. 7. Split (Left) and Merge (Right) at dimension `one`.

Then, orbit results on *connectivity* [11] for `A m k`, `A_1 m k` and their compositions apply, allowing recursive definitions of the numbers of edges, vertices, faces. The number of connected components, w.r.t. {`A m zero, A m one`}, is defined similarly. All this leads to an incremental definition of the *Euler characteristic*, to the *Genus theorem* and to *full constructive planarity criteria* [9, 11].

High-level operations Operations to *break* a `k`-link, *delete* an (isolated) dart, and *shift* the hole of a `k`-orbit, are specified inductively. They preserve the hypermap invariant. Finally, `Split m k x y` *splits* an orbit for `A m k` into two parts and `Merge m k x y` *merges* two distinct orbits for `A m k` (Fig. 7 for `k:= one`). For both, the goal is to insert a new `k`-link from `x` to `y`, while restoring the consistency of the resulting hypermap. The two operations correspond exactly with an orbit *transposition* (Sect. 2):

```
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.
```

5 General memory model

Goal We want to derive imperative programs using "high-level" C type facilities, i.e. `typedef`. So, in our memory C manager simulation, we avoid "low-level" features − bits, bytes, words, registers, blocks, offsets, etc − as in projects about compilers or distributed systems [21, 6]. In C, an allocation is realized by `malloc(n)` where `n` is the byte-size of the booked area. A C *macro*, we call `MALLOC(T)`, hides the size and allows us to allocate an area for any object of type `T` and to return a pointer (of type `(T *)`) on this object:

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

Our (simple) *memory model* simulates it, with natural numbers as potential *addresses* and the possibility to always obtain a *fresh* address to store any object.

Coq formalization In Coq, the address type `Addr` is `nat`, with the exception `null` equal to `0`. In a Coq section, we first declare *variables*: `T:Type`, for any type, and `undef`, exception of type `T`. Then we define a *memory generic type* `Mem T`, for data of type `T`, inductively with two constructors: `initm`, for the empty memory; and `insm M a c`, for the insertion in the memory `M` at the address `a` of an object `c:T`. It is easy to recursively define `valid M a`, the predicate which expresses that `a` is a *valid* address (i.e. pointing to a stored object) in `M`, and `dom M`, the *validity domain* of `M`, a finite address set − or instead, list. A precondition derives for `insm` and an invariant `inv_Mem M` for each memory `M`. Finally, the parameter function `adgen` returns a fresh address from a memory, since it satisfies the *axiom* `adgen_axiom`, which is the second and last one (after *extensionality*) of our full development. This mechanism looks like the *axiom of choice* introduction which does not affect Coq's consistency:

```
Variables (T:Type) (undef:T).
Inductive Mem: Type:= initm : Mem | insm : Mem -> Addr -> T -> Mem.
Fixpoint valid(M:Mem)(z:Addr): Prop :=
   match M with initm => False | insm M0 a c => a = z \/ valid M0 z end.
Definition prec_insm M a := ~valid M a /\ a <> null.
Parameter adgen: Mem -> Addr.
Axiom adgen_axiom: forall(M:Mem), let a := adgen M in ~valid M a /\ a <> null.
```

Memory operations The *allocation* of a block for an object of type `T` (recall that `T` is an implicit parameter) is defined by (`%type` forces Coq to consider `*` as the Cartesian type product):

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

It returns a pair composed of a new memory and a fresh address for an allocated block containing `undef`. It formalizes the behavior of the above `MALLOC`. The other operations, which are "standard", are proved *conservative* w.r.t. to `inv_Mem`: `load M z` returns the data at the address `z` if it is valid, `undef` otherwise; `free M z` releases, if necessary, the address `z` and its corresponding block in `M`; and `mut M z t` changes in `M` the value at `z` into `t`.

6 Hypermap linked implementation

Linked cell memory As usual in geometric libraries, we orient the implementation toward linked cells. First, at each *dart* is associated a structure of type `cell`, defined thanks to a Coq `Record` scheme. So, a dart is itself represented by an address. Then, the type `Mem`, which was parameterized by `T`, is instantiated into `Memc` to contain `cell` data. In the following, all the generic memory operations are instantiated like `Memc`, with names suffixed by "`c`":

```
Record cell:Type:= mkcell { s : dim -> Addr; p : dim -> Addr; next : Addr }.
Definition Memc := Mem cell.
```

So, `s` and `p` are viewed as *functions*, in fact *arrays* indexed by `dim`, to represent *pointers* to the successors and predecessors by `A`, and `next` is a *pointer*, intended for a *next* cell, used for hypermap full traversals (e.g., see Fig. 8 or 9).

Main linked list A 2-dimensional hypermap is represented by a pair `(M, h)`, where `M` is such a memory and `h` the *head pointer* of a *singly-linked linear list* of cells representing exactly the darts of the hypermap. The corresponding type is called `Rhmap`. Then, a lot of functions are defined for this representation. Their names start with "`R`", for "Representation", and their meaning is immediate:

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

So, `Rnext M z` gives the address of `z`'s *successor* in the list, `Rorb Rm` and `Rlim Rm` are `z`'s *orbit* and *limit* for `Rnext M` w.r.t. `domc M` when `Rm = (M, h)`. Operations `Rexd`, `RA` and `RA_1` are intended to be the representations of `exd`, `A` and `A_1` of the hypermap specification. However, this will have to be proved.

Invariants and consequences To manage well-defined pointers and lists, our hypermap representation must be restricted. We have grouped constraints in a *representation invariant*, called `inv_Rhmap`, which is the conjunction of `inv_Rhmap1` and `inv_Rhmap2` dealing with orbits:

```
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.
```

For a hypermap representation `Rm = (M,h)`: *(i)* `inv_Rhmap1 Rm` prescribes that `M` is a well-formed cell memory, `h` is `null` or in `M`, and the limit of `h` by `Rnext M` w.r.t. the domain of `M` is `null`. Therefore, the corresponding orbit, called `Rorb Rm`, is a *line*; *(ii)* `inv_Rhmap2 Rm` stipulates that, for all dimension `k`, and address `z` in `Rorb Rm`, the orbit of `z` by `RA M k` w.r.t. `Rorb Rm` is a *circuit*, and `RA_1 M k z` is the *inverse image* of `z` for `RA M k`.

So, for any address `z` in `Rorb Rm`, i.e. of a cell in the main list, we immediately have that `RA` and `RA_1` are *inverse operations* and that the orbit of `z` by `RA_1 M k` w.r.t. `Rorb Rm` is also a *circuit*. Consequently, for `k` = `zero` or `one`, the fields `(s k)` and `(p k)` are inverse pointers which determine *doubly-linked circular lists*, each corresponding to a hypermap edge or vertex, which can be traversed in forward and backward directions. Moreover, these lists are *never empty*, and, for each `k` and direction, they determine a *partition* of the main simply-linked list.

User operations Now, a *complete kernel* of user concrete operations may be defined, exactly as in geometric libraries, e.g. Topofil [2], preserving `inv_Rhmap` and hiding dangerous pointer manipulations. We "program" it by using Coq functional forms whose translation in C is immediate.

• Firstly, `RV` returns an *empty* hypermap representation in any well-formed cell memory `M`. So, no address `z` points in the representation:

```
Definition RV(M:Memc): Rhmap := (M, null).
Lemma Rexd_RV: forall M z, inv_Memc M -> ~ Rexd (RV M) z.
```

• Secondly, `RI Rm` *inserts* at the head of `Rm = (M, h)` a new cell whose generated address is `x`, initializes it with convenient pointers (`ficell x` initializes `s`'s and `p`'s pointers with `x`), and returns the new memory, `M2`, and head pointer, `x`. Fig. 8 illustrates `RI` when `Rm` is not empty:

```
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).
```

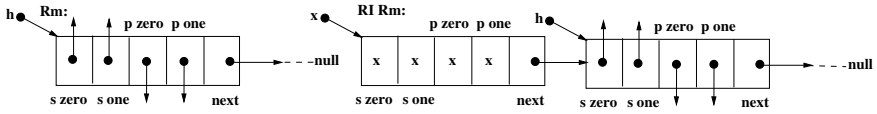


Fig. 8. Operation RI: Inserting a new dart in `Rm` (Left) giving `RI Rm` (Right).

Numerous properties are proved by following the *state changes* simulated by the variables assignments. Of course, `RI` does what it should do: adding a new cell pointed by `x`, and creating *new fixpoints* at `x` for `RA` and `RA_1` at any dimension. The corresponding orbits of `x` are *loops*, while the other orbits are unchanged (`eq_Addr_dec` is the address comparison):

```
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_Addr_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_Addr_dec x z then x else RA_1 M k z.
```

Proofs use in a crucial way properties of orbit operations (Sect. 2): `Rnext`, `RA` and `RA_1` can be viewed through additions and mutations in orbits, e.g.:

```
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.
```

• Thirdly, `RL m k x y` performs a *transposition* involving the orbits of `x` and `y` at dimension `k`. It replaces, for function `RA Rm k`, the successor of `x` by `y`, and the successor of `RA_1 m k y` by `RA m k x`, and also modifies the predecessors consistently. This operation, which is the composition of four memory mutations, is illustrated in Fig. 9 for `k:= one`. It is proved that the expected missions of `RL` are satisfied, for `Rexd`, `RA m k` and `RA_1 m k`. So, the orbits for `RA m k` and `RA_1 m k` remains *circuits*, and `RL` realizes either a *merge* or a *split*. The proofs extensively use relations established with the orbit operations, particularly with the generic orbit `Tu` transposition (Sect. 2), e.g., `M6` being the final memory state:

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

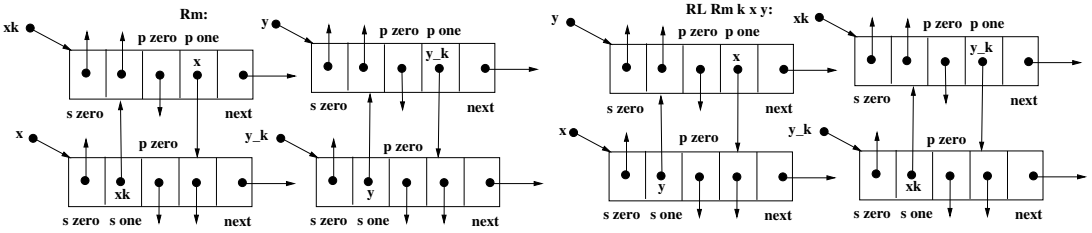


Fig. 9. Operation RL: Transposing two darts of `Rm` (Left) giving `Rm k x y` (Right).

• Fourthly, `RD Rm z H` achieves a *deletion* of the pointer `x` (which must be a fixpoint for `RA` and `RA_1`) and a *deallocation* of the corresponding cell, if any. The additional formal parameter `H` is a proof of `inv_Rhmap1 Rm`, which is used to guarantee the *termination* of `x`'s searching in `Rm`. Indeed, in the general case, one has to find `x_1`, the *predecessor* address of `x` in the linked list, thanks to a *sequential search*. It is proved that, after `RD`, `x` is always invalid, and `Rnext`, `RA` and `RA_1` are updated conveniently. Once again, the proofs [11] rely on properties of orbit deletion and mutation (Sect. 2).

7 Equivalence hypermap specification / implementation

Abstraction function We want to go from a hypermap representation, `Rm`, to its meaning − its *semantics* − in terms of hypermap specification by an *abstraction function*. From a user's viewpoint, the construction of `Rm` is realized exclusively from a well-formed memory `M` throughout `RV`, `RI`, `RL` and `RD` calls, satisfying the preconditions. That is expressed by a predicate, called `CRhmap Rm`, which is inductively defined (for technical reasons of recursion principle generation) in `Type` [11]. If `Rm` satisfies `CRhmap Rm`, then it is proved to be a consistent hypermap representation. Then, the *abstraction function*, called `Abs`, is recursively defined by a matching on a proof `CRm` of this predicate, `m0 := Abs Rm0 H0` being the result from the previous hypermap, `Rm0`, in the recursion, if any:

```
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.
```

So, we prove that `Abs` leads to a real abstract hypermap, with an exact correspondence between `exd`, `A` and `A_1` of `Abs Rm CRm` and `Rexd`, `RA` and `RA_1` of `Rm`.

Representation function Conversely, we go from a hypermap to its representation in a given memory `M`. However, the mapping is possible only if the darts correspond to addresses which would be generated by successive cell allocations from `M`. So, the *representation function* `Rep` returns from a hypermap a pair `(Rm, Pm)`, where `Rm:Rhmap` is a hypermap representation and `Pm:Prop` is a proposition saying if the preceding property is satisfied or not:

```
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.
```

Consequently, the representation into `Rm` succeeds *if and only if* `Pm` is satisfied. In this case, `Rm` satisfies `inv_Rhmap`, with an exact correspondence between `exd`, `A` and `A_1` of `m` and `Rexd`, `RA` and `RA_1` of `Rm`. These results make `Abs` and `Rep` *morphisms*. So, we have an *observational equivalence* specification-representation.

8 Related work and discussion

Static proofs of programs They are mostly rooted in Floyd-Hoare logic, which evaluates predicates over a variable *stack* and a memory *heap* throughout program running paths. To overcome difficulties of conjoint local and global

reasoning, Reynolds's *separation logic* [26] considers heap *regions* − combined by ∗, a *conjunction operator* −, to link predicates to regions. In fact, predicates often refer to underlying *abstract* data types. Then, the reasoning concerns at the same time low- and high-level concepts, which is quite difficult to control. These approaches are rather used to prove isolated already written programs.

Algebraic specification To specify and correctly implement entire *libraries*, we prefer starting with a formal specification, then deriving an implementation, and proving they correspond well. So, like [3, 24, 23, 17, 7], we find it better to specify with *inductive datatype* definitions in the spirit of *algebraic specifications* [28]. They must be constrained by *preconditions* inducing datatype *invariants*. Of course, definitions and proofs by *structural induction* must be supplemented by *Nœtherian* definitions and reasoning, to deal with termination problems.

Memory and programming models To implement, we focus on C, in which our geometric programs are mostly written [2]. Here, fine storage considerations [21, 6] are not necessary: A simple "type-based" memory model is good enough. Coq helps to simulate C memory management, expressions and commands at high level, following the "good rules" of structured (sequential) programming. Moreover, we try to converge to *tail-recursive definitions*, which are directly translatable into C loops. Of course, additional *variant* parameters to deal with termination, e.g. in `RD`, must be erased, as in the Coq *extraction* mechanism [1].

Separation and collision Burstall [4] defines *list systems* to verify linked list algorithms. Bornat [3] deals with address sequences like in our orbits. Mehta and Nipkow propose relational abstractions of linear or circular linked lists with some separation properties in Hoare logic embedded in Isabelle/HOL [24]. These questions are systematized in *separation logic* [26, 25] which is well suited to distributed or embedded systems [18] and to composite data structures [17]. Enea et al. propose a separation logic extension for program manipulating "overlaid" and nested linked lists [16]. They introduce, $*_w$, a field separating conjunction operator to compose structures sharing some objects. Their examples turn around nested *singly*-linked lists and list arrays. In fact, our orbit notion seems sufficient for all singly- or doubly-linked linear or cyclic data structures, at several levels, and predicates can be expressed by using Coq logic. Separation is expressed by *disjunction* of orbits, whereas collision (and aliasing) is described by their *coalescence*, orbits often playing the role of *heaps* in separation logic.

Hypermap specification To specify hypermaps [8] in Coq/SSReflect, Gonthier et al. [20] adopt an *observational* viewpoint with a definition similar to Def. 2 (Sect. 3). So, they quickly have many properties on permutations. Their final result, i.e. the proof of the Four-Colour theorem, is resounding [20]. Our approach is *constructive*, starting from a free inductive type, which is constrained into a hypermap type. It complicates the true beginning but favors structural induction, algorithms and verifications in discrete topology [9, 10, 14].

Dedicated proof systems The Ynot project is entirely based on Coq [15, 22]: Higher-order imperative programs (with pointers) are constructed and verified in a Coq specific environment based on *Hoare Type Theory*, separation logic, monads, tactics and Coq to OCaml extraction. Thanks to special annotations, proof obligations are largely automated. Our approach is more pragmatic and uses Coq, and orbits, without new logic features. The proofs are interactive, but specification and implementation are distinct, their links being morphisms, and the simulated imperative programs are close to C programs. Many verification platforms, e.g. Why3 [19] or Frama-C [5], are based on Hoare logic, and federate solvers, mainly SAT and SMT, often in first-order logic. The Bedrock system by Chlipala [6] is adapted to implementation, specification and verification of low-level programs. It uses Hoare logic, separation logic and Coq to reason about pointers, mostly automatically. However, it cannot tolerate any abstraction beyond that of assembly languages (words, registers, branching, etc).

9 Conclusion

We presented a study in Coq for the entire certified development of a hypermap library. We follow a data refinement by algebraic specifications and reuse at each stage orbit features [12]. To certify pointer implementations, we use only Coq on a simple memory model, instead of Hoare logic and separation logic. The Coq development for this hypermap application − including the memory model, contains about 9,000 lines (60 definitions, 630 lemmas and theorems) [11]. It imports a Coq generic orbit library [12] reusable to certify implementations with singly- or doubly-linked lists, alone, intermixed or nested.

In the future, we will generalize orbit results, in extension of [13], to deal with general trees or graphs. Then, we will try to establish an accurate relation with *separation logic*, where orbits could provide the basis of new predicates on complex structures. The *fragment of C* which we cover by Coq must be precised to characterize reachable programs and develop a translator to C. The use of automatic provers, often jointed in platforms is questionnable. It would be interesting to carry orbit notions in their specification languages, e.g. ACSL in Frama-C [5]. At high level, we will investigate 3-dimensional hypermaps to deal with 3D functional or imperative computational geometry programs, like the 3D Delaunay triangulation which remains a formidable challenge.

References

1. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004.
2. Y. Bertrand, J.-F. Dufourd, J. Françon, and P. Lienhardt. Algebraic Specification and Development in Geometric Modeling. In *4th Int. Joint Conf. CAAP/FASE: TAPSOFT, LNCS 668, Springer*, pages 75–89, 1993.
3. R. Bornat. Proving Pointer Programs in Hoare Logic. In *5th Conf. on Mathematics of Program Construction, MPC'00, LNCS 1837, Springer*, pages 102–126, 2000.

4. R.M. Burstall. Some Techniques for Proving Correctness of Programs which Alter Data Structures. *Machine Intelligence*, 7:23–50, 1972.
5. CEA-LIST and INRIA-Saclay-Proval Team. *Frama-C Project.* http://frama-c.com/about.html, 2013.
6. A. Chlipala. Mostly-Automated Verification of Low-level Programs in Computational Separation Logic. In *Int. ACM Conf. PLDI'11*, pages 234–245, 2011.
7. C. L. Conway and C. Barrett. Verifying Low-Level Implementations of High-Level Datatypes. In *CAV'10, LNCS 6174, Springer*, pages 306–320, 2010.
8. R. Cori. Un code pour les graphes planaires et ses applications. *Soc. Math. de France, Astérisque*, 27, 1970.
9. J.-F. Dufourd. Polyhedra genus theorem and Euler formula: A hypermap-formalized intuitionistic proof. *Theor. Comp. Science*, 403(2-3):133–159, 2008.
10. J.-F. Dufourd. An Intuitionistic Proof of a Discrete Form of the Jordan Curve Theorem Formalized in Coq with Combinatorial Hypermaps. *J. of Automated Reasoning*, 43(1):19–51, 2009.
11. J.-F. Dufourd. *Hmap Specification and Implementation - On-line Coq Development.* http://dpt-info.u-strasbg.fr/~jfd/Hmap.tar.gz, 2013.
12. J.-F. Dufourd. Formal Study of Functional Orbits in Finite Domains. *submitted to TCS*, 2013, 40 pages.
13. J.-F. Dufourd. Dérivation de l'Algorithme de Schorr-Waite par une Méthode Algébrique. In *JFLA'2012, INRIA, hal-00665909*, Carnac, Feb. 2012, 15 pages.
14. J.-F. Dufourd and Y. Bertot. Formal Study of Plane Delaunay Triangulation. In *ITP'10, LNCS 6172, Springer*, pages 211–226, 2010.
15. A. Chlipala et al. Effective Interactive Proofs for Higher-Order Imperative programs. In *ICFP'09*, pages 79–90, 2009.
16. C. Enea et al. Compositional Invariant Checking for Overlaid and Nested Linked Lists. In *ESOP'13, LNCS 7792, Springer*, pages 129–148, 2013.
17. J. Berdine et al. Shape Analysis for Composite Data Structures. In *Computer-Aided Verification, CAV'07, LNCS 4590, Springer*, 2007.
18. N. Marti et al. Formal Verification of the Heap Manager of an Operating System Using Separation Logic. In *ICFEM'06*, pages 400–419, 2006.
19. J.-C. Filliâtre. Verifying Two Lines of C with Why3: An exercise in program verification. In *VSTTE*, pages 83–97, 2012.
20. G. Gonthier. Formal Proof - the Four Color Theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.
21. X. Leroy and S. Blazy. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. of Autom. Reas.*, 41(1):1–31, 2008.
22. J. Gregory Malecha and Greg Morrisett. Mechanized Verification with sharing. In *ICTAC*, pages 245–259, 2010.
23. C. Marché. Towards Modular Algebraic Specifications for Pointer Programs: A Case Study. In *RCP'07, LNCS 4600, Springer*, pages 235–258, 2007.
24. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1-2):200–227, 2005.
25. P.W. O'Hearn, H. Yang, and J.C. Reynolds. Separation and information hiding. *ACM TOPLAS*, 31(3), 2009.
26. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS'02*, pages 55–74, 2002.
27. CGAL Team. *Computational Geometry Algorithms Library Project, Chapter 27: Combinatorial Maps.* http://www.cgal.org, 2013.
28. M. Wirsing. *Algebraic Specification - In Handbook of TCS*, volume B. Elsevier/MIT Press, 1990.