Pointer Program Derivation using Coq: Graphs and Schorr-Waite Algorithm

Jean-François Dufourd *

University of Strasbourg - CNRS, ICUBE Laboratory,
Pôle API, Boulevard S. Brant, CS 10413, 67412 Illkirch, France - jfd@unistra.fr

Abstract. We present a specification, a derivation and total correctness proofs of operations for bi-functional graphs implemented with pointers, including the Schorr-Waite algorithm. This one marks such a graph with an economical depth-first strategy. Our approach is purely algebraic and functional, from a simple graph specification to the simulation of a tail-recursive imperative program, then to a true C pointer program by elementary classical transformations. We stay in the unique higher-order formalism of the Calculus of Inductive Constructions for specifications, programs and proofs. All the development is supported by Coq.

1 Introduction

The Schorr-Waite (in short SW) algorithm [31] traverses iteratively a bi-functional graph coded by pointers in depth-first order from an initial vertex and marks all the visited vertices. The problem is classical, but the solution of Schorr and Waite is inexpensive because it avoids any auxiliary storage by a clever use of temporarily unemployed graph pointers. Such an algorithm is useful in the cell marking step of a garbage collector, when the available memory is scarce. Many researchers used this algorithm as a *benchmark* to test manual (or little automated) methods of program transformation and verification [32, 33, 8, 20, 19, 17, 9, 4, 28, 34]. Since 2000, studies have addressed the proofs with automatic tools, for partial or total correctness [3, 1, 27, 22, 26, 15, 29]. Indeed, the proof of total correctness is considered as the first montain to climb in the verification of pointer programs [3].

Here, we report a new experiment of formal specification, derivation and total correctness proof of a bi-functional graph datatype with its concrete operations, including the SW algorithm whose study is particularly difficult. We highlight the following *novelties* of the derivation process:

• **Formalism.** We stay in the general higher-order *Calculus of Inductive Constructions* (in short CiC) formalism for specifications, programs and proofs, and use the *Coq proof assistant* as single software tool [2]. The only added axioms are *proof-irrelevance*, *extensionality*, and an axiom of *choice* for a fresh address during an allocation. We thus intentionally avoid *assertions method* and *Hoare logic* underlying most works in verification of programs.

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

• **Abstract level.** We first focus on the algebraic specification in Coq of abstract datatypes. We define by structural or Noetherian induction the results as abstract functions and prove required properties.
• **Concrete level.** We specify a memory algebraic type with pointers in which we implement the abstract specification. We seek *morphisms* carrying properties from abstract to concrete levels.
• **Programming.** Coq does an extraction in OCaml, while forgetting proof-parameters. Memory parameters are removed to go into an imperative program.
• **Orbits.** At the abstract and concrete levels, the *orbit* notion helps to manage the track of function iterations, e.g. to concisely write type invariants and to manage linkage traversals as in *shape analysis* [21]. Orbits were approached in [6, 30, 3, 27, 21] and deeply studied in [13]. They allow to deal with *separation* problems [30] without extending the CiC. To specify *combinatorial hypermaps* [18, 10], and to derive imperative pointer *geometric* datatypes and programs, *orbits* are particularly efficient [14]. So, our correctness proof of the SW algorithm can be considered as another case study for orbits.
• **SW algorithm.** Finally, this process provides a version of the SW algorithm acting on any marked bi-functional graph, for which there is a proof that the corresponding operation is *idempotent*, i.e. it has the same effect whatever applied once or several times.

Sect. 2 specifies marked bi-functional graphs and Sect. 3 a depth-first graph marking. Sect. 4 defines internal stacks and Sect. 5 constructs a depth-first marking with them. Sect. 6 specifies memories and Sect. 7 defines a graph-memory isomorphism. Sect. 8 carries the marking with internal stack from graph to memory. Sect. 9 extracts it into an OCaml function, which is transformed "by hand" into an iterative C-program. Sect. 10 proves that the initial specification fits well with reachability. Sect. 11 presents work related to the SW algorithm, and Sect. 12 concludes. The complete Coq development (with proofs) is available on-line [12], including [13]. A preliminary French version, with another specification, is in [11]. A basic knowledge of Coq makes the reading of this article easier.

2 Bi-functional graphs

Basic definitions. As in Coq [2], we write `nat` for the type of natural numbers. We assume that `undef` and `null`, not necessarily distinct, code particular natural values. In this work, a *(marked bi-functional) graph* `g = (E,mark,son0,son1)` is a finite subset `E` of *vertices* (or *nodes*) in `nat - {undef,null}`, so-called *support* of `g`, equipped with three functions: `mark` returning a number inside `{0,1,2}`, `son0` and `son1` returning natural numbers named *left* and *right* sons, which do not necessarily belong to `E`. An example is given in Fig. 1(Left), with `E = {1,...,8}`, marks in the vertex circles (filled with blank, light or dark grey depending on the mark value: 0, 1 or 2), `son0` and `son1` represented by arcs with labels 0 or 1.

In the specification, the functions are extended at the whole `nat`: outside `E`, `mark` returns 0, and `son0`, `son1` return `undef`. To avoid tedious elementary tests, we first inductively (here just enumeratively) define in Coq the mark type `nat2`

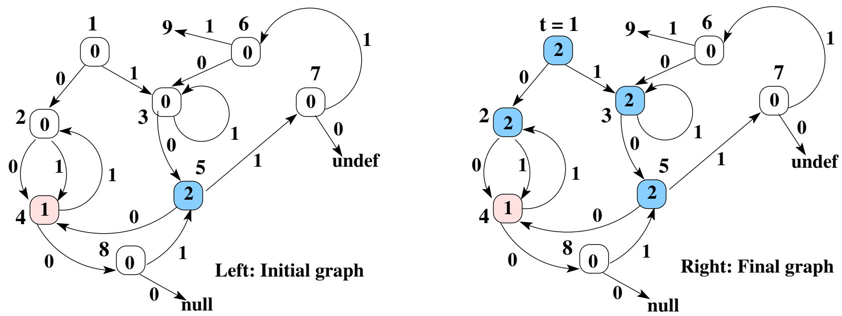

Fig. 1. Graph before (Left) and after (Right) depth-first marking from $t = 1$.

(`Type` is viewed as the "type of types"). Then, it is convenient to inductively define the type `graph` by two *constructors*: `vg`, returning the *empty* (or *void*) graph, and `iv g x m x0 x1`, *inserting* in the graph `g` a new vertex, `x`, with its mark, `m`, and its two sons, `x0` and `x1`:

```
Inductive nat2 : Type:= zero : nat2 | one : nat2 | two : nat2.
Inductive graph : Type:= vg : graph | iv : graph -> nat -> nat2 -> nat -> nat -> graph.
```

Observers and graph invariant. A predicate, `exv g z`, for testing the existence of any `z:nat` in a graph `g` is recursively defined in Coq by a pattern matching on `graph` (`Prop` is the type of *propositions* and _ is a placeholder). Then, functions `mark` and `son` are also recursively written in functional style (`son0`, `son1` are compacted into a unique `son` parameterized by a label `k = 0` or `1`). However, to construct only *well-formed* graphs, the calls of `iv` must respect the precondition `prec_iv`. So, if necessary, `graph` may be constrained by the *invariant* `inv_graph` (~ is written for *not* and `<>` for $\neq$):

```
Fixpoint exv(g:graph)(z:nat): Prop:=
 match g with vg => False | iv g0 x _ _ _ => x = z \/ exv g0 z end.
Definition prec_iv(g:graph)(x:nat): Prop := ~ exv g x /\ x <> null /\ x <> undef.
Fixpoint inv_graph(g:graph): Prop :=
 match g with vg => True | iv g0 x m _ _ => inv_graph g0 /\ prec_iv g0 x end.
```

Other graph *observers* are similarly defined: `nv` is the number of vertices and `marksum` is the sum of the mark values of the existing vertices. Numerous results on them are proved, often by structural induction on `graph`, e.g. the lemma:

```
Lemma marksum_bound: forall g, marksum g <= 2 * nv g.
```

Mutators. Functions to update graphs are also written: `chm g z m` changes the mark of `z` into `m`, and `cha g k z zs` the `k`-th son (or *arc*, for `k = 0` or `1`) of `z` into `zs`. They *preserve* the graph invariant and enjoy properties of *idempotence*, *permutativity* and *absorption* which are essential in the following, e.g. (`eq_nat_dec` tests the equality in `nat`):

```
Lemma chm_chm: forall g z1 m1 z2 m2,
  chm (chm g z1 m1) z2 m2 = if eq_nat_dec z1 z2 then chm g z2 m2 else chm (chm g z2 m2) z1 m1.
Lemma chm_idem: forall g z, chm g z (mark g z) = g.
Lemma cha_chm: forall g x y z k m, k <= 1 -> cha (chm g z m) k x y = chm (cha g k x y) z m.
```

3 Specification of depth-first marking

Preliminaries. We slightly enlarge the traditional *marking* problem: (*i*) we deal with any graph `g`, i.e. equipped *with any marking* (between 0 and 2) and *any sons* (in the support of `g` or not); (*ii*) starting from any natural number `t`, the problem consists in traversing in depth-first order the *subgraph* of `g` of all the 0-marked vertices reachable from `t` and in marking them by 2. Fig. 1(Right) gives the final marking of the graph in Fig. 1(Left) when `t = 1`. With this setting, the *stopping condition* of the depth-first traversal from any `t` is:

```
Definition stop g t := ~ exv g t \/ mark g t <> 0.
```

Then, naming `stop_dec` the function which tests if `stop g t` is satisfied or not (`stop` is easily proved *decidable*), the entire problem is solved by the function which we name `df` and define in Coq syntax as follows (surrounded by quotes because this non-primitive recursive definition is not accepted as such by the Coq system):

```
"Definition df(g:graph)(t:nat): graph :=
   if stop_dec g t then g
   else let g0 := df (chm g t two) (son g 0 t) in df g0 (son g 1 t)."
```

As other authors [19, 33, 9], we consider that `df` explicitly states the problem as simply as possible, as if `g` was a binary tree. From now on we consider it as our *specification*. Unfortunately, such a recursive definition cannot be directly written in Coq without dealing with *termination*. Moreover, the *nested* (*double*) recursion adds a difficulty. But such problems of general recursion can be overcome in Coq [2] (p. 419-420, for numerical problems).

True Coq specification. First, we define a graph *measure*, `mes`, which will decrease at each recursive call. Then, we consider two binary relations on `graph`:

```
Definition mes g := 2 * nv g - marksum g.
Definition ltg g' g := mes g' < mes g.
Definition leg g' g := mes g' <= mes g.
```

They are a *strict* and a *large preorder*, `ltg` is *Noetherian* (or *well-founded*), and the use of `chm` inside `df`'s body decreases `mes`. In fact, the termination of `df` needs `ltg (chm g t two) g`, which is immediate, and `ltg g0 g`, which is satisfied if `leg g0 (chm g t two)`. This requires as result a graph, and also the fact that this graph is less than or equal to `g`. In Coq, such a result has the *existential* type *depending on* `g` denoted by {`g':graph | leg g' g`}, as for usual mathematical subsets. Then, an auxiliary function of `df`, named `df_aux`, with a result of this type, has itself a *functional type* which is defined by:

```
Definition df_aux_type := fun g:graph => nat -> {g':graph | leg g' g}.
```

So, `df_aux` must be a function which transforms a graph, `g:graph`, into a function which in turn transforms `t:nat` into a *pair*, `(g', H')`, where `g'` is the marked graph and `H'` a proof of `leg g' g`. The building of `df_aux` corresponds with the proof of a theorem. Indeed, Coq implements the *Curry-Howard correspondence*, stating that proofs and functions are isomorphic. The proof, which has roughly the skeleton of `df`'s informal specification, uses our results on the decreasing of `mes` in the recursive calls of `df`. We do not give the exact definition of `df_aux` which is rather technical, but the interested reader may consult [11]. Finally, remembering that `exist` is the Coq constructor of {`g':graph | leg g' g`}, the "true" `df` is obtained by extracting the *witness* of the result, i.e. the marked graph `g'`:

```
Definition df(g:graph)(t:nat): graph := match df_aux g t with exist g' _ => g' end.
```

Of course, the *termination* of `df_aux`, and of `df`, is *automatically ensured* by these constructions. The definition of `df` is rather mysterious for non-specialists, but the following properties are illuminating.

Properties of the Coq specification. Most properties of `df` are obtained by Noetherian induction on `df_aux` using built-in recursors. First of all, `df` preserves `inv_graph`, the initial graph *vertices* and *sons*, and the marking is always *increasing*. An important result − absent from all studies considering an initial marking with 0 only −, is that `df` is *idempotent*, i.e. reapplying it does not change the result. Finally, we exactly obtain the expected original definition of `df` by proving the *fixpoint equation* `df_eqpf`. So, since it possesses all the properties we want to prove, `df` is a solid reference for transformations towards a real program:

```
Lemma inv_graph_df: forall g t, inv_graph g -> inv_graph (df g t).
Lemma exv_df: forall g t z, exv (df g t) z <-> exv g z.
Lemma son_df: forall g t z k, son (df g t) k z = son g k z.
Lemma mark_le_mark_df: forall g t z, mark g z <= mark (df g t) z.
Lemma df_idem: forall g t, df (df g t) t = df g t.
Theorem df_eqpf: forall g t,
  df g t = if stop_dec g t then g
           else let g0 := df (chm g t two) (son g 0 t) in df g0 (son g 1 t).
```

4 Succession function, orbits, internal stack

Orbits. Now, we simulate an *(internal) stack* inside a graph `g` , thanks to a *total function* `succ`:

```
Definition succ g z :=
  if eq_nat_dec z null then null
  else if eq_nat_dec (mark g z) 0 then null else son g ((mark g z) - 1) z.
```

This function can be *iterated*: for any integer `k`, the `k`-th iterate of `succ g` from `z` is `zk := Iter (succ g) k z`, where `Iter` is the classical iteration functional (with `z0 = z`). The iterates form in `g`'s support a list that we call the *orbit* of `z`. We studied this notion in a general way [13]. Here, it is used to express that such a list always ends on `null`, outside `g`'s support.

Internal stack. For us, the orbit of `z` in `g`'s support − the orbit length is written `lenorb g z` − is an *internal stack* if it satisfies the following *invariant*:

```
Definition inv_istack g z : Prop :=
  let r := lenorb g z in let zr := Iter (succ g) r z in let zr_1 := Iter (succ g) (r-1) z in
    zr = null /\ (0 < r -> 1 <= mark g zr_1 <= 2).
```

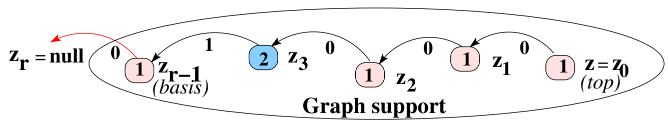

Fig. 2. Shape of a (non-empty) internal stack, with r = 5.

In Fig. 2, `r` (= 5) gives the internal stack *height*, whereas `z (= z0)` and `zr_1` can be viewed as its *top* and *basis* when the orbit is non-empty. Consequently, all the internal stack elements are (genuine) non-`zero` marked vertices of `g`. Internal stacks are affected by mark or son updates. For general orbits, the different updating cases are thoroughly analyzed [13] as in *shape analysis* [21]. However, the SW algorithm only uses some particular configurations which are related to three basic operations, which we present now.

Internal stack operations. They are defined as follows:

```
Definition ipush g t p := cha (chm g t one) 0 t p.
Definition iswing g t p := cha (cha (chm g p two) 0 p t) 1 p (succ g p).
Definition ipop g t p := cha g 1 p t.
```

• `ipush g t p` pushes a vertex `t` on an internal stack whose top is `p`, after a change of `t`'s mark into `one` (Fig. 3(a1)). Its precondition requires that `t` is a true `zero`-marked vertex. After `ipush g t p`, `p` remains the top of an internal stack, but `t` is also the top of another one including the former. The left son of `t` is now used to access to `t`'s successor, i.e. `p`, in the new stack.
• `iswing g t p` is a *rotation* at the top `p` of an internal stack to change its sons after change of its mark from `one` into `two`. This "stack" operation is emblematic of the SW algorithm (Fig. 3(b1)): `iswing g t p` replaces the left son which led to the successor in the internal stack by the right son, reestablishing the initial left son of `p` into `t`, `p` being no more father of its true right son.
• `ipop g t p` pops from an internal stack `p` its top (i.e. `p`), and reestablishes its right son. The precondition requires that `p`'s mark is `two` (so `exv g p` is verified) (Fig. 3(c1)): after `ipop g t p`, `succ g p` is the top of the remaining stack, whose height decreases by 1 and which might become empty.

It is proved that these operations preserve the graph and internal stack invariants, the graph vertices, and that `ipush` and `iswing` add 1 to the mark sum, whereas `ipop` leaves it unchanged.

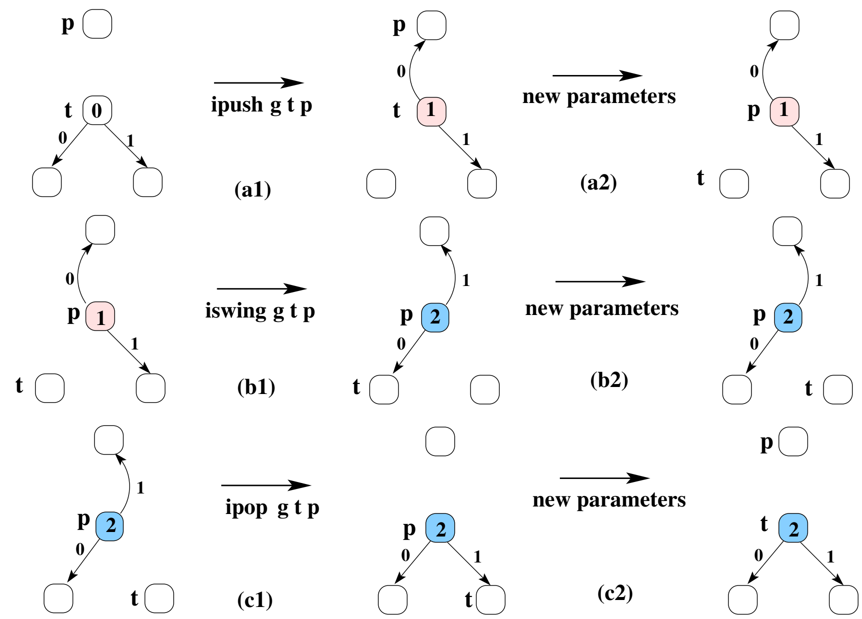

Fig. 3. Operations on internal stacks.

5 Depth-first marking using an internal stack

Cartesian product. To simulate the SW algorithm, we have to deal with the type, named `graphistack`, of the pairs `(g,p)` composed of a graph `g` and an internal stack top `p` (In Coq, `*` is the Cartesian type product, used with `%type` to remove ambiguities). We equip it with the invariant `inv_graphistack` (`fst` and `snd` are the classical projections). This invariant is satisfied with the empty internal stack and is preserved by each of the three operations defined in Sect. 4:

```
Definition graphistack := (graph * nat)%type.
Definition inv_graphistack(gp:graphistack) := inv_graph (fst gp) /\ inv_istack (fst gp) (snd gp).
```

Designing the algorithm. The algorithm we look for is *simply tail-recursive*. For the parameter `gp = (g,p)`, its termination will be warranted by the strict decreasing of the *measure* `2 * mes g + lenorb g p` at each recursive call involving one of the operations `ipush`, `iswing` or `ipop`. Our previous results entail this decreasing. So, a suitable binary relation on `graphistack` is `ltgip`, which is quickly proved to be a *Noetherian strict preorder*:

```
Definition ltgip (gp' gp:graphistack) := let (g',p') := gp' in let (g,p) := gp in
     2 * mes g' + lenorb g' p' < 2 * mes g + lenorb g p.
```

The same method as for `df` allows us to define `dfi`, our new recursive marking function *with internal stack*. A large preorder is useless since the algorithm is simply recursive. However, the proofs of measure decreasing need `inv_graphistack` at each recursive call. So, for the result of our auxiliary function `dfi_aux`, whose type is `dfi_aux_type`, we introduce the subtype {`gp:graphistack | inv_graphistack gp`}. We then complete the stopping predicate `stop` of `df` into `stopi`:

```
Definition dfi_aux_type :=
 fun gp:graphistack => inv_graphistack gp -> nat -> {gp':graphistack | inv_graphistack gp'}.
Definition stopi p g t := p = null /\ stop g t.
```

The algorithm stops when `p` is `null`, and `t` is not in `g` or has a non-zero mark, the corresponding testing function being `stopi_dec`. To construct `dfi_aux`, the method is similar to that of `df_aux` (Sect. 3), following the subsequent informal specification written in Coq *pseudo-code*. The new parameters `g, p, t` for the three recursive internal calls to `dfi_aux` corresponding to `ipop`, `iswing` and `ipush` are given in Fig. 3(a2,b2,c2):

```
"Definition dfi_aux (g, p) t :=
  if stopi_dec p g t then (g, p)
  else if stop_dec g t
       then if eq_nat_dec (mark g p) 2
            then dfi_aux (ipop g t p, son g 1 p) p
            else dfi_aux (iswing g t p, p) (son g 1 p)
       else dfi_aux (ipush g t p, t) (son g 0 t)"
```

Finally, `dfi` is obtained by the projection of `dfi_aux` on the `graph` component when starting with an empty stack (`inv_graphistack_null g hg` is a proof that `inv_graphistack` is satisfied for the genuine graph `g` with the empty stack):

```
Definition dfi (g:graph)(hg:inv_graph g)(t:nat) : graph:=
   match dfi_aux (g,null) (inv_graphistack_null g hg) t with exist (g',s) _ => g' end.
```

Note that `dfi` keeps a proof argument, `hg:inv_graph g`. Besides, a fixpoint equation similar to the above informal specification is proved for `dfi_aux` in the same way as for `df` (Sect. 3).

Total correctness of the algorithm. The *termination* of `dfi` being automatically ensured, the great question is the *partial correctness* of `dfi` with respect to `df`. In fact, our *fundamental result* is the *identity* between `dfi` and `df`: for the same `g` and `t`, they return the same graph *regardless of what the actual proof argument* `hg` for `dfi` is:

```
Theorem df_dfi : forall g hg t, dfi g hg t = df g t.
```

The proof uses a new iteration function on an argument `gp:graphistack` [11] and the general properties of *orbit* update operations, particularly the *mutation* [13]. The consequences are numerous, since all the nice properties of `df` are immediately transposed to `dfi`, e.g. *preservation of the graph invariant*, *preservation of the initial vertices and sons*, *mark growing*, *idempotence* (Sect. 3). As far as the SW algorithm, we could stop the study at this point, considering that the path is well traced towards an imperative iterative C-program for an experienced programmer, especially since `df` is simply tail-recursive. But a lot of implementation problems are still to be solved, particularly regarding pointers, because the graph which we use for convenience is a "ghost", i.e. it does not explicitly appear in the final program. Indeed, in imperative programming, a function like `dfi` should only be parameterized by an *address* `t`. So, we now model memories to translate our graph specification into a C-program.

6 Memory model

Cells and memory. Advanced memory models allow to capture allocator subtleties which are useful to prove the correctness of compilers or intricate programs with composite data [25]. Our present goal being to derive only one structured program on a unique datatype, our memory model is directly specialized towards a graph pointer representation. Memory *cells* are of the following type, `cell`, where `mkcell` is the constructor, and `val, s0, s1` are field selectors, for mark, left and right sons. An *exception* cell, `initcell`, is defined. Rather than giving a complex − dangerous in sense of consistency − axiom system, we found it safe to algebraically define the memory type `Mem` as follows:

```
Record cell:Type:= mkcell {val : nat2; s0 : nat; s1 : nat}.
Definition initcell := mkcell zero undef undef.
Inductive Mem:Type:= init : Mem  | alloc : Mem -> nat -> cell -> Mem.
```

The *addresses* are simulated by natural numbers, `init` returns the empty memory, and `alloc` inserts in a memory a cell value at a (new) address, during an *allocation.* Our memories are finite, unbounded, and allocations never fail.

Memory operations. Now, a predicate `exm` tests if an address is *valid* in a memory, i.e. corresponds to an allocated cell. Then, the usual functions, `load`, `free` and `mut`, respectively to *get* from an address a cell contents, to *free* a cell (and its address), and to *change* a cell contents giving its address, are easily defined by pattern matching [11]. However, allocations must satisfy the following precondition, which leads to an *invariant* `inv_Mem` for `Mem`:

```
Definition prec_alloc M a := ~exm M a /\ a <> undef /\ a <> null.
Fixpoint inv_Mem(M:Mem): Prop :=
  match M with init => True | alloc M0 a c => inv_Mem M0 /\ prec_alloc M0 a end.
```

A lot of lemmas about the behavior of the operations are proved by induction on `Mem`. We have mimicked more realistic programming primitives, particularly a C-like `malloc` returning from a memory a fresh address, thanks to an *address generator*, whose behavior is governed by a *dedicated axiom* [11].

7 Memory to graph, graph to memory

Abstraction and representation. We define two operations: `Abs`, to *abstract* a memory into a graph, and `Rep` to *represent* a graph as a memory, the *reversibility* of which is confirmed by the following theorems:

```
Fixpoint Abs(M:Mem): graph :=
  match M with init => vg | alloc M0 a c => iv (Abs M0) a (val c) (s0 c) (s1 c) end.
Fixpoint Rep (g:graph) : Mem :=
  match g with vg => init | iv g0 x m x0 x1 => alloc (Rep g0) x (mkcell m x0 x1) end.
Theorem Rep_Abs : forall M, Rep (Abs M) = M.
Theorem Abs_Rep : forall g, Abs (Rep g) = g.
Theorem inv_graph_Abs : forall M, inv_Mem M -> inv_graph (Abs M).
Theorem inv_Mem_Rep : forall g, inv_graph g -> inv_Mem (Rep g).
```

Transposition of operations and properties. Graph operations are implemented by `load` and `mut` into memory ones, here with the same name preceded by "R", e.g. `Rcha` and `Rchm`. In fact, `Abs` and `Rep` make `graph` and `Mem` *isomorphic*. So, the behavioral proofs of `graph` operations are simply carried on `Mem`:

```
Lemma Rchm_chm : forall M x m, Rchm M x m = Rep (chm (Abs M) x m).
Lemma chm_Rchm : forall g x m, chm g x m = Abs (Rchm (Rep g) x m).
Lemma Rcha_cha : forall M k x y, Rcha M k x y = Rep (cha (Abs M) k x y).
Lemma cha_Rcha : forall g k x y, cha g k x y = Abs (Rcha (Rep g) k x y).
```

8 Depth-first marking in memory

Specification of marking in a memory. The predicates `stop` and `ltg` become `Rstop` and `Rltg` for memories. The lemmas we had for `df` are transposed to specify the (nested) recursive depth-first marking `Rdf` in memories, with exchange theorems. Consequently, all the properties of `df` in `graph` are transposed to `Rdf` in `Mem`, e.g. we have a *fixpoint equation*, `Rdf_eqpf`, similar to `df_eqpf`:

```
Theorem df_Rdf : forall g t, df g t = Abs (Rdf (Rep g) t).
Theorem Rdf_df : forall M t, Rdf M t = Rep (df (Abs M) t).
```

Depth-first memory marking with internal stack. Operations `ipush`, `iswing` and `ipop` are easily transposed for `Mem` into `Ripush`, `Riswing` and `Ripop` with the same properties. Then, the counterpart of `graphistack` is `Memistack`, with the invariant `inv_Memistack`:

```
Definition Memistack := (Mem * nat) %type.
Definition inv_Memistack(Mp:Memistack) := inv_Mem (fst Mp) /\ inv_Ristack (fst Mp) (snd Mp).
```

At `stopi` and `ltgip` correspond `Rstopi` and `Rltgip`. The definition of the *marking in memory with internal address stack*, i.e. `Rdfi` (with `Rdfi_aux`), follows.

Total correctness. Of course, `Rdfi` is *terminating*. Then, by our isomorphism `graph` - `Mem`, we transpose in `Mem` our proof of correctness of `dfi` w.r.t. `df` into a proof of correctness of `Rdfi` w.r.t. `dfi`. Better, we have for free the *correctness* of `Rdfi` w.r.t. our specification `df` in graphs:

```
Theorem Rdfi_dfi : forall (M : Mem) (hM : inv_Mem M) (t : nat),
   Rdfi M hM t = Rep (dfi (Abs M) (inv_graph_Abs M hM) t).
Theorem Rdfi_df : forall (M : Mem) (hM : inv_Mem M) (t : nat),
   Rdfi M hM t = Rep (df (Abs M) t).
```

9 Towards concrete programming

Extraction in OCaml. The *extraction-of-functional-program* Coq tool [2] leads to an OCaml version of our development. Hence, after an elementary substitution, we get the following program for `Rdfi_aux` and `Rdfi` (in OCaml, "R" and "M" are in lower case, the Coq decision functions, `Rstopi_dec` and `Rstop_dec`, become Boolean functions, and the natural numbers are in Peano notation). As usual, the extraction removes all the proof-terms and retains the common data only. A *functional form* of the SW algorithm follows. Since `rdfi_aux` is *tail-recursive*, it will be easy to write it *iteratively* without a stack:

```
let rec rdfi_aux m p t =
  if rstopi_dec p m t then (m, p)
  else if rstop_dec m t
       then if eq_nat_dec (rmark m p) (S (S O))
            then rdfi_aux (ripop m t p) (rson m (S O) p) p
            else rdfi_aux (riswing m t p) p (rson m (S O) p)
       else rdfi_aux (ripush m t p) x (rson m O t)
let rdfi m t = fst (rdfi_aux m null t)
```

Derivation of a C-program. From the OCaml version, we derive graph imperative operations. We first define in C the types of cells and addresses, which were integers (`nat2` is suppressed for simplicity):

```
typedef struct strcell {nat val; struct strcell * s0; struct strcell * s1;} cell, * address;
```

As usual in C, `null` is written `NULL`, the memory is *implicit* and modified by *side-effects*. As far as the SW algorithm, Fig. 3(a2,b2,c2) explains how the parameter pair `(p, t)` mutates by `ripush`, `riswing` and `ripop`, like in the functional version. An *auxiliary variable*, `q`, is used to serialize C assignments. We can as usual replace `exm t` by `t != NULL`, and the way `undef` is translated is not important. Finally, we transform the tail-recursion into an iteration, unfold all internal functions, and the imperative iterative (ingenious) SW procedure looks like a variant of the C version in [22], where each mark is coded by two bits. The procedure works correctly *regardless of what the initial marking is*, the standard situation − *all marks are* 0 − being just a particular case:

```
void rdfi(address t){
   address p = NULL, q;
   while (!(p == NULL && (t == NULL || t->val != 0)))){
     if(t == NULL || t->val != 0){
       if(p->val==2) {q = p->s1; p->s1 = t; t = p; p = q;}
       else {p->val = 2; q = p->s0; p->s0 = t; t = p->s1; p->s1 = q;}
     }
     else {t->val = 1; q = t->s0; t->s0 = p; p = t; t = q;}
   }
 }
```

10 Back to the specification

Although the starting point of numerous studies, `df` can be considered as *too constructive* w.r.t. the *reachability* (Sect. 3) [19, 27, 22, 26, 24]. If `reachable g t z` means that, in `g`, `z` can be reached from `t` only via zero-marked vertices, its definition can be (`nat2_to_nat` maps `nat2 into nat`):

```
Fixpoint reachable(g:graph)(t z:nat): Prop : match g with
     vg => False
   | iv g0 x m x0 x1 => reachable g0 t z \/ nat2_to_nat m = 0 /\
      (x = t /\ x = z \/ (x = t \/ reachable g0 t x)
      /\ (x0 = z \/ reachable g0 x0 z \/ x1 = z \/ reachable g0 x1 z))
end.
```

Under some simple conditions, `reachable g` is proved *decidable* (with decision function `reachable_dec`), *reflexive* and *transitive*. The specifications `reachable` and `df` should be compared. We did it through a simply recursive marking, named `dfs`, using a classical *external vertex stack* and enjoying the same behavior as `dfi`. So `df = dfs`, and since `df = dfi` (Sect. 5), then `df = dfs = dfi`. Finally, the following theorem *fully characterizes* the effect of `dfs`, and `df`, on all `g`'s vertices. It also entails the *correctness* of `dfi`, and `Rdfi`, *with respect to reachability*:

```
Theorem reachable_dfs : forall g hg t z, mark (dfs g t hg) z =
    if reachable_dec g t z then if stop_dec g z then mark g z else two else mark g z.
```

In summary, the whole derivation process is synthesized in Fig. 4 where all functions, relations, equalities, isomorphisms and equivalences appear.

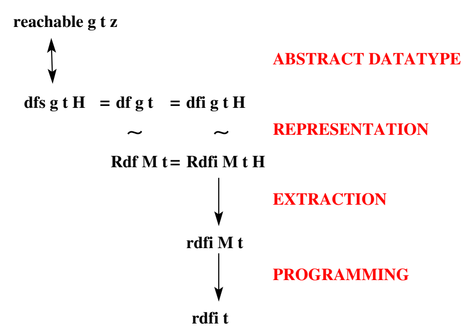

Fig. 4. Derivation levels, functions and relations.

11 Work related to Schorr-Waite algorithm

Pioneering work. The SW algorithm, discovered independently by Deutsch ([23], p. 417) was published as a routine for garbage collection [31]. Many *program constructions by derivation*, e.g. by Griffiths [20], start with a doubly recursive imperative procedure, introduce progressively (internal) stack elements, and show that transformations preserve good properties.

Topor and Suzuki give the first formal proofs "by hand" [33, 32]. Topor introduces predicates and procedures comparable to `df`, `dfs` and `dfi`, but acting on sets and lists with side-effects. The proof applies the *intermittent assertions* method, with an induction on the data structure size that our graph inductions sometimes remind. Suzuki develops an automatic program verifier able to deal with pointers, but his attempt on the SW algorithm remains incomplete.

Gries publishes a correctness proof of the SW program using the *assertions method* with *weakest preconditions* [19]. In a vertex array simulating the memory, the graph is represented by a set of paths. Morris writes a proof in the same spirit using Hoare logic [28]. Gerhart [17] proposes a proof *by derivation* from an abstract problem of transitive closure to Gries's program using sets, sequences and arrays. The proof using the *assertions method* is partially verified by Affirm. Following Topor's proof, de Roever [8] illustrates the *greatest fixpoint theory* by the total correctness of a SW algorithm which is far enough a way from the C program. Dershowitz revisits in rather informal style the SW algorithm *derivation* and proof for vertices with d sons [9]. He starts with a recursive procedure having an internal loop, progressively introduces counters, then an internal stack, and ends with a version including two *goto*'s. Ward uses a *transformational model-based method* to set the problem then to derive in WSL and prove the SW algorithm [34]. It uses transformation rules which are proved correct, thus avoiding to prove the correctness of the derivation itself.

Broy and Pepper use *algebraic specifications* to derive and prove the total correctness of the algorithm [4]. They specify marked vertex sets, then 2-graphs as sets with 2 functions. An axiom of *permutativity* forces to use an *equality modulo* for graphs. The same in Coq would alter *Leibniz equality* and prevent proofs of equality for functions returning graphs. This explains our focus on the `graph` specification. The starting point is a doubly recursive procedure acting on a set and a graph. They algebraically specify generic arrays to simulate memories. Several imperative procedures are obtained thanks to a generic transformation rule eliminating double recursions. The last version mentions a set and a path and is still far from the C program. Our study can be viewed as a logical continuation of this work.

Work using automated tools. Following Burstall [6], Bornat [3] gives a rationale to prove pointer programs in *Hoare logic* with semantic models of *stack* and *heap*. In a memory (heap) viewed as an array, he follows iterated addresses by an *f* function, defines *f-linked sequences*, and studies their dynamic behavior. The SW algorithm is partially verified in the proof editor Jape [3].

Abrial uses the model-based *Event B method* to refine and merge (in 8 steps) specifications given by separate elementary assignments into a final pointer program [1]. Invariants, with pre-postconditions on sets and relations are progressively built with proof *obligations*. The Atelier B is used to prove the partial correctness, 70% automatically.

Mehta and Nipkow propose an Isabelle framework to prove pointer programs in *higher-order logic* [27]. They implement a small language for annotated programs and tools to reason in *Hoare logic* with a semantic model of *heap* and *stack*. A special attention is paid to capture *separation* properties [3, 30] with list and path abstractions. They prove the *partial correctness* of two versions of the SW algorithm from Bornat's work [3].

Loginov et al. elaborate a completely automated proof of total correctness using *three-valued logic*, with deep analysis of reachability in pointer structures, but only for binary trees or dags [26]. Hubert and Marché use the *assertion method* in the Caduceus system for a direct proof of a C source version of the SW algorithm [22]. A big invariant concerns the evolution of reachability, marking, stack, sons, paths, etc. They automatically prove about 60% of the correctness, the rest, e.g. termination, being left to Coq (about 3000 lines). Bubel relates a proof part of a Java implementation. The specification in Java Card DL is based on reachability, the proofs use the KeY system but do not mention termination [5].

Leino describes in Dafny a very performing implementation. Big pre-, post-conditions and loop invariant group four kinds of properties. The total correctness verification is automatic (in a few sec.) thanks to SMT solvers [24]. However, the author says he finally prefers a method by refinement, like [1]. Yang uses the relational separation logic to show that the SW algorithm is equivalent to a depth-first traversing, but he mentions no automation [35].

Giorgino et al. study a *method by refinement*, first based on spanning trees then enriched to graphs, for the total correctness of the SW algorithm, using Isabelle/HOL [15]. Finally, they use state-transformers and monads (in Isabelle) to deal with imperative programs. Proteasa and Back present the *invariant based programming*, a refinement approach by *predicate transformers* supported by *invariant diagrams* [29]. A diagram contains the information necessary to verify that each derivation towards the SW algorithm is totally correct. The process has been verified by Isabelle.

12 Conclusion

Coq development. We derived a graph library and the SW algorithm, and proved their *total correctness* with Coq. The development *from scratch* represents about 8,400 lines, with 480 definitions, lemmas or theorems. That is the price for such a complete study with a general proof assistant.

Advantages of our approach. We deal with a *single* powerful logical framework, i.e. CiC and Coq, at abstract and concrete levels. Coq allows us to simulate algebraic datatypes with *inductive types* equipped with preconditions and invariants. It offers good facilities for *general recursive functions* if proof parameters are added to address nested recursions [2]. This is facilitated by the mechanism of *dependent type*.

Our approach is *global* because, at the two levels, graph types and operations have to be specified, implemented and proved correct all together. Constraints are distributed among invariants, preconditions and proof-parameters. So, big complex invariants, as in monolithic proofs of the SW algorithm, are broken in several pieces easier to manage.

Besides, *orbit* features allow to express predicates about data separation or collision at high and low levels in a synthetical way [13, 14].

Abstraction and representation *morphisms* carry on operations and properties, which are *proved once*, and, with *extensionality*, help to prove the *equality* of functions. The final step towards programming uses the *extraction-from-proof* mechanism and classical elementary program transformations.

Limitations and future work. Complex algebraic data must be studied to see how equalities of objects and functions will behave. For instance, dependent constructors force to *congruences*, which are difficult to deal with in Coq, even with *setoids*, and we could sometimes be happy with *observational equalities*.

The transformation of a functional recursive version with memory into an iterative imperative program is classical and has good solutions in well-defined cases. However, it should be computer-aided, even automated in a *compiler*.

Our approach prevents the help of program verification tools based on Hoare logic, e.g. Why3 [16] or Bedrock [7], which also use Coq. However, the introduction of our orbits in such frameworks must be considered to write predicates about separation and collision, as in [13, 14].

Finally, as our predecessors, we found the total correctness proof of the SW algorithm to be hard work. But the memory management is still simple in this algorithm, since it *does not include allocation nor deallocation*. In fact, the most delicate was not to do proofs, but to find how the problem should be posed.

References

1. J.-R. Abrial. Event Based Sequential Program Development: Application to Constructing a Pointer Program. In *FME, LNCS 2805, Springer*, pages 51–74, 2003.
2. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004.
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. M. Broy and P. Pepper. Combining Algebraic and Algorithmic Reasoning: An Approach to the Schorr-Waite Algorithm. *ACM-TOPLAS*, 4(3):362–381, 1982.
5. R. Bubel. The Schorr-Waite Algorithm. In *Verification of Object-Oriented Software: The {KeY} Approach*, volume 4334 of *LNCS*, pages 569–587. 2007.
6. R.M. Burstall. Some techniques for proving correctness of programs which alters data structures. *Machine Intelligence*, 7:23–50, 1972.
7. A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245, 2011.
8. W.-P. de Roever. On Backtracking and Greatest Fixpoints. In *4th ICALP, LNCS 57, Springer*, pages 412–429, 1977.
9. N. Dershowitz. The Schorr-Waite Marking Algorithm Revisited. *Inf. Proc. Lett.*, 11(3):141–143, 1980.
10. J.-F. Dufourd. Polyhedra genus theorem and Euler formula: A hypermap-formalized intuitionistic proof. *Theor. Comp. Sci.*, 403(2-3):133–159, 2008.
11. J.-F. Dufourd. Dérivation de l'algorithme de Schorr-Waite en Coq par une méthode algébrique. In *JFLA'2012, INRIA*. http://hal.inria.fr/hal-00665909, 2012.

12. J.-F. Dufourd. *Schorr-Waite Coq Development On-line Documentation.* http://dpt-info.u-strasbg.fr/~jfd/SW-LIB-PUBLI.tar.gz, 2013.
13. J.-F. Dufourd. Formal Study of Functional Orbits in Finite Domains. *submitted*, 2013, 35 pages.
14. J.-F. Dufourd. Hypermap Specification and Certified Linked Implementation using Orbits. In *ITP'2014, LNCS 8558, Springer*, 2014 (*to appear*).
15. M. Giorgino et al. Verification of the Schorr-Waite algorithm - From trees to graphs. In *20th LOPSTR'2010, LNCS 5464, Springer*, pages 67–83, 2010.
16. J.-C. Filliâtre. Verifying Two Lines of C with Why3. In *VSTTE*, pages 83–97, 2012.
17. S.L. Gerhardt. A derivation-oriented proof of the Schorr-Waite algorithm. In *Program Construction, LNCS 69, Springer*, pages 472–492, 1979.
18. G. Gonthier. Formal Proof - The Four-Color Theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.
19. D. Gries. The Schorr-Waite Graph Marking Algorithm. *Acta Informatica*, 11:223–232, 1979.
20. M. Griffiths. Development of the Schorr-Waite algorithm. In *Program Construction, LNCS 69, Springer*, pages 464–471, 1979.
21. B. Hackett and R. Rugina. Region-Based Shape Analysis with Tracked Locations. In *32th ACM POPL'05*, pages 310–323, 2005.
22. T. Hubert and C. Marché. A case study of C source code verification; the Schorr-Waite algorithm. In *3rd IEEE SEFM'05*, pages 190–199, 2005.
23. D.E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* Add.-Wesley, 1968.
24. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, volume 6355 of *LNCS*, pages 348–370, 2010.
25. X. Leroy and S. Blazy. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *JAR*, 41(1):1–31, 2008.
26. A. Loginov, T. Reps, and M. Sagiv. Automatic Verification of the Deutsch-Schorr-Waite Tree Traversal Algorithm. In *13th SAS'2006, LNCS 4134, Springer*, pages 261–274, 2006.
27. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Info. and Comp.*, 199(1-2):200–227, 2005.
28. J.M. Morris. A Proof of the Schorr-Waite Algorithm. In *TFPM*, volume 91, pages 43–51. NATO, D. Reidel, 1982.
29. V. Preoteasa and R.-J. Back. Invariant diagrams with data refinement. *FAC*, 24(1):67–95, 2012.
30. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS'02*, pages 55–74, 2002.
31. H. Schorr and W.R. Waite. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *CACM*, 10(8):501–506, 1967.
32. N. Suzuki. *Automatic Verification of Programs with Complex Data Structures.* PhD Th., Dept. of CS, Stanford, 1976.
33. R.W. Topor. The Correctness of the Schorr-Waite List Marking Algorithm. *Acta Inf.*, 11:211–221, 1979.
34. M. Ward. Derivation of Data Intensive Algorithms by Formal Transformation. *IEEE-TOSE.*, 22(9):665–686, 1996.
35. H. Yang. Relational separation logic. *TCS*, 375(1-3):308–334, 2007.