2013 First International Symposium on Computing and Networking

# Powerlists in Coq: Programming and Reasoning

Frédéric Loulergue*, Virginia Niculescu†, Simon Robillard‡
*Univ Orléans, ENSI de Bourges, LIFO EA 4022, Orléans, France,
Frederic.Loulergue@univ-orleans.fr
†Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania
vniculescu@cs.ubbcluj.ro
‡Univ Orléans, ENSI de Bourges, LIFO EA 4022, Orléans, France,
Simon.Robillard@univ-orleans.fr

***Abstract*—For parallel programs, correctness by construction is an essential feature since debugging is extremely difficult and costly. Building correct programs by construction is not a simple task, and usually the methodologies used for this purpose are rather theoretical and based on a pen-and-paper style. A better approach could be based on tools and theories that allow a user to develop an efficient parallel application by easily implementing simple programs satisfying conditions, ideally automatically proved. Powerlists theory and its variants represent a good theoretical base for such an approach, and the Coq proof assistant is a tool that could be used for automatic proofs. The goal of this paper is to model the powerlist theory in Coq, and to use this modelling to program and reason about parallel programs in Coq. This represents the first step in building a framework to ease the development of correct and verifiable parallel programs.**

***Keywords*—*Parallel Recursive Structures; Interactive Theorem Prover; Coq; Applications***

## I. Context and Motivation

Our general goal is to ease the development of correct and verifiable parallel programs with predictable performances using theories and tools to allow a user to develop an efficient application by implementing simple programs satisfying conditions easily, or ideally automatically, proved.

To attain this goal, our framework, as depicted in Figure 1 will be based on (1) high-level algebraic theories of parallel programs, (2) modelling of these theories inside interactive theorem provers (or proof assistants) such as Coq [1] or Isabelle [2] to be able to write programs and reason about them, and (3) axiomatisation of lower level parallel programming primitives and their use to implement the high-level primitives in order to extract [3] actual *parallel* code from the developments made within proof assistants. Each of these components is necessary for the framework. (3) makes the framework practical: It is possible to write and reason about parallel programs written in existing non verified parallel programming libraries inside a proof assistant and to extract code parameterised by the axiomatisation of these libraries. This code is then instantiated by actual implementations of the libraries and compiled in order to obtain executable parallel code. In this paper we focus on (2).

Among high-level algebraic theories, at least two seem suitable for parallel programming: the theory of lists [4] and parallel recursive structures such as powerlists [5]. The SDPP framework [6], a partial accomplishment of our long term goal, currently focuses on the theory of lists. Powerlists and variants are data structures introduced by J. Misra and J. Kornerup, which can be successfully used in a simple, provably correct, functional description of parallel programs, which are divide and conquer in nature. They allow working at a high level of abstraction, especially because the index notations are not used. The contribution of this paper is a modelling of powerlists in Coq, and its use to express and reason about programs within Coq. Our methodology is to obtain a small axiomatisation of this data structure, as close as possible to the pen-and-paper version, and then to build on it, including the definition and proof of induction principles.

The paper is organised as follows. We first give some elements on Coq (section II) before discussing powerlist axiomatisation (section III). From this axiomatisation, we prove an induction principle (section IV) and use it to reason about some functions, also defined using the axiomatisation (section V). We discuss related work (section VI) before concluding (section VII).

## II. Introduction to Coq

Coq [1] is a proof assistant based on the Curry-Howard correspondence [7] relating terms of a typed $\lambda$-calculus with proof trees of a logical system in natural deduction form. The calculus behind Coq is the calculus of (co)-inductive constructions, an extension of the initial Calculus of Constructions [8].

From a more practical side, Coq can be seen as a functional programming language, close to OCaml [9] or Haskell [10] but with a richer type system that allows to express logical properties. As a matter of fact, Coq is often used as follows: programs are developed in Coq and their properties are also proved in Coq. This is for example the case of the CompertC compiler, a compiler for the C language, implemented and certified using Coq [11].

In order to discuss our modelling of powerlists in Coq, we first introduce the main Coq features that we use, through examples on the list data structure. This short introduction complements the one in [12].

As in any programming language, we need data structures to work on. Data structures in Coq are defined by induction. For example the list data structure can be defined as follows:

```
Inductive list (A:Type):Type:=
  | nil: list A
  | cons: A → list A → list A.
```

978-1-4799-2795-1/13 $31.00 © 2013 IEEE
DOI 10.1109/CANDAR.2013.17

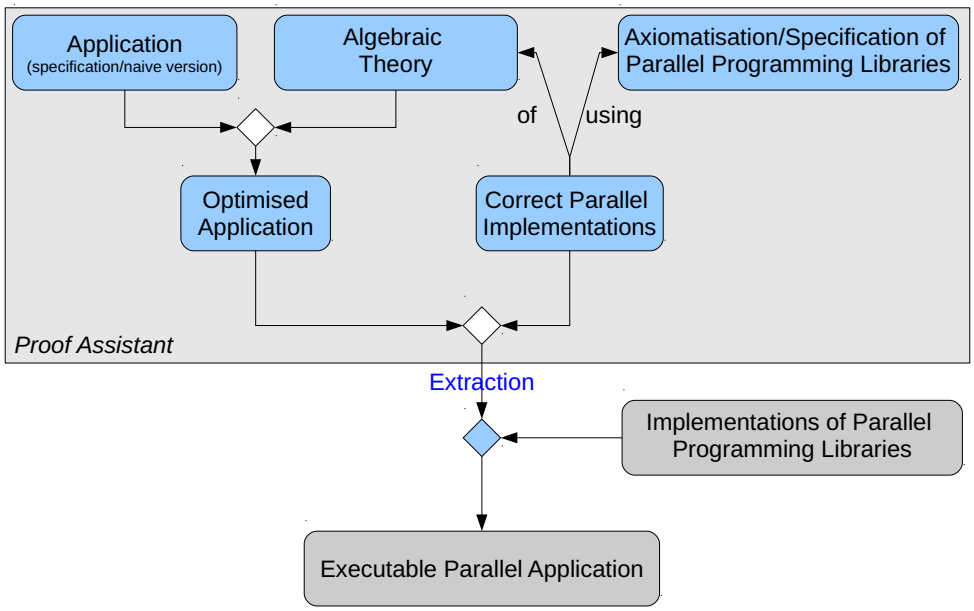


Figure 1. The framework

This data structure is a polymorphic data structure as it takes an argument `A` which type is `Type` meaning `A` is a type. Coq is a higher order $\lambda$-calculus as it allows to make terms dependent on types (and types dependent on terms). Types are themselves typed. There are two possibilities to build a list: either by the `nil` *constructor*, representing an empty list, or by the `cons` constructor that takes an element (of type `A`), a list (of type `list A`) and yields a new list augmented with the new element. Constructors and functions in Coq are basically in curried form.

We omit the details here[1], but it is possible in Coq to indicate that some of the arguments of functions or constructors should be implicit, meaning they should be inferred from the context. It is also possible to define usual notations for lists: $[x_1;\ldots;x_n]$ for lists by enumeration, and `::` in infix notation for `cons`. It is thus possible to define new values:

```
Definition l1 : list ℕ := [ 0; 1; 2; 3].

Definition l2 := [ "Hello"; "World" ].
```

In the first definition, we explicitly give the type of the value before defining it, for `l2` we let Coq infers the type and only give the definition. Now let us define functions on lists. We first define the concatenation on lists. This is a recursive definition, thus we use the `Fixpoint` keyword and we define it by pattern matching on the first list:

```
Fixpoint app (A:Type) (l1 l2:list A) :=
  match l1 with
    | [] ⇒ l2
    | x::xs ⇒ x::( xs ++ l2 )
  end
where "xs ++ ys" := (app xs ys).
```

[1]The full source code: http://traclifo.univ-orleans.fr/SDPP/wiki/PowerLists

We also simultaneously define the infix notation `++` for list concatenation. This definition is accepted by Coq as the recursive call is done on a strict sub-term of `l1`. In case the recursive call is not done on a sub-term of the arguments, Coq will reject the recursive definition. Yet if the function is actually terminating, more involved mechanisms may be used to define it. Similarly we can define the `length` function:

```
Fixpoint length (A : Type) (l : list A) :=
  match l with
    | []    ⇒ 0
    | x::xs ⇒ 1 + length xs
  end.
```

When stating a proposition (or a lemma, or a theorem), we want to specify the type (the logical assertion), but we prefer not to directly write the proof as a $\lambda$-term. It is easier to perform backward reasoning as we would do using a natural deduction proof system. This could be done in Coq using the proof mode and the Ltac language of tactics:

```
Lemma app_length:
  ∀(A : Type)(l1 l2: list A),
    length(l1++l2) = length l1 + length l2.
Proof.
  intros A l1.
  induction l1 as [ | x xs IH ].
  - (* case [] *)
    intros. simpl. reflexivity.
  - (* case x::xs *)
    intros. simpl. rewrite IH. trivial.
Qed.
```

After the `Proof` command, Coq enters in its interactive proof mode and requests the user to solve a *goal*, in this case the statement of the lemma. After we feed Coq with the two first tactics (`intros` and `induction`), we have two goals to prove, each corresponding to a case of the reasoning by induction on the first list:

```
2 subgoals, subgoal 1 (ID 37)
  A : Type
  ============================
   forall l3 : list A, length ([] ++ l3) =
                       length [] + length l3
subgoal 2 (ID 41) is:
 forall l3 : list A, length ((x::xs)++l3) =
                     length (x :: xs)+length l3
```

The text in green are comments, and each item solves the associated goal. Using the command `Print app_length` we can obtain the $\lambda$-term corresponding to this proof.

To define the function that returns the first element of a list, we could return an optional value of type `option A`:

```
Definition head0 (A:Type)(l:list A):option A:=
  match l with
    | [] ⇒ None
    | x::xs ⇒ Some x
  end.
```

so that this function applied to an empty list will return the value `None`, or we could define `head` only on non-empty lists. To do so, we could add an additional logical parameter which type states that the argument list is non empty:

```
Program Definition
  head (A:Type)(l:list A)(H:l<>[]): A :=
  match l with
    | [] ⇒ _
    | x::xs ⇒ x
  end.
```

Still when defining it, we should reason by case on the argument list. In the case of the empty list we would obtain a contradiction with hypothesis `H`. However this is a proof rather than a function definition: we would need here to mix the proof mode with the usual definition mode. This is possible using the `Program` feature of Coq. The _ symbol is a kind of hole that may be filled later in proof mode. Indeed Coq generates a proof obligation at this position, but as the `Program` feature comes with some automation, the proof obligation at hand is automatically discharged and no proof is required from the user.

It is possible to extract actual programs from Coq developments. Build-in support includes OCaml, Haskell and Scheme, the former being the default. The command `Recursive Extraction head` yields the following OCaml program, that could be compiled to native code:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list

let head l = match l with
  | Nil  → assert false (* absurd case *)
  | Cons (x, xs)  → x
```

Coq support modular development. Signatures could be specified in *module types*, and *modules* could realise these module types. It is also possible to build a module (or module type) parameterised by another module: we then have a higher-order module. For example, the beginning of an axiomatisation of join-lists could be:

```
Module Type JLIST.
  Parameter t : Type → Type.
  Parameter empty : ∀ (A:Type), t A.
  Parameter single: ∀ (A:Type)(a:A), t A.
  Parameter app: ∀(A:Type), t A → t A → t A.
  Axiom id: ∀(A:Type)(l:t A),
            app empty l=l ∧ app l empty=l.
End JLIST.
```

All kinds of definitions are replaced by `Parameter` declarations. Lemmas, propositions, theorems are replaced by `Axiom`s. Instead of defining by induction a data structure of join-lists we could implement it using the already defined cons-list:

```
Module JoinList : JLIST.
  Definition t := list.
  Definition empty (A:Type):= @nil A.
  Definition single (A:Type)(a:A) :=
    cons a empty.
  Definition app := app.
  Theorem id: ∀(A:Type)(l:t A),
              app empty l=l ∧ app l empty=l.
  Proof. split; induction l; simpl;
         f_equal; auto. Qed.
End JoinList.
```

In case the element in the module type is an axiom, the module should provide a proof of this "axiom" that becomes a lemma or a theorem.

`Parameter` and `Axiom` could also be used at the top-level of Coq. However this means we add axioms to Coq's logic, without any guarantee that these axioms are consistent with Coq's logic. Using module types we can use axioms in a more local way, and by providing modules we can verify that the axiomatisation is not contradictory with Coq's logic: this is the good way to axiomatise data structures, without being tied by a specific implementation. Nevertheless this approach cannot be used for axioms that could not be proved with Coq's logic, such as some classical logic statements.

## III. Axiomatisation of Powerlists

Many known parallel algorithms – FFT, Batcher Merge, prefix sum, embedding arrays in hypercubes [5], [13], etc. – have concise descriptions using powerlists. Powerlists have simple algebraic properties that allow deduction of the properties of these algorithms by employing structural induction.

There are two different ways in which powerlists could be joined to create a longer powerlist. If $p; q$ are powerlists of the same length then

- *tie*: $p|q$ is the powerlist formed by concatenating p and q, and
- *zip*: $p \bowtie q$ is the powerlist formed by successively taking alternate items from $p$ and $q$, starting with $p$.

For example, if $p = \langle 0\ 1 \rangle$ and $q = \langle 2\ 3 \rangle$ then

$$\langle 0\ 1 \rangle | \langle 2\ 3 \rangle = \langle 0\ 1\ 2\ 3 \rangle$$
$$\langle 0\ 1 \rangle \bowtie \langle 2\ 3 \rangle = \langle 0\ 2\ 1\ 3 \rangle$$

According to J. Misra [5], a general recursive definition of powerlists is:

*If $S$ is a scalar, $P$ is a powerlist, and $u, v$ are similar powerlists then*

$$\langle S\rangle, \langle P\rangle, \text{ and } u \mid v, u \bowtie v$$

*are powerlists, too.*
*Two scalars are similar if they are of the same type. Two powerlists are similar if they have the same length and any element of one is similar to any element of the other.*

It could be considered strange to allow two different ways of constructing the same list using *tie* or *zip*. (From this we have two induction principles that could be equally used.) Yet this flexibility is essential since many parallel algorithms (e.g. Fast Fourier Transform) exploit both forms of construction.

In order to formally define these structures, Misra defined a set of algebraic laws:

- L0 (identical base cases): $\langle x\rangle|\langle y\rangle = \langle x\rangle \bowtie \langle y\rangle$
- L1 (dual deconstruction): if $P$ is a non-singleton powerlist there exist $r, s, u, v$ similar powerlists such that:
$$\begin{array}{ll}(a) & P = r \mid s\\ (b) & P = u \bowtie v\end{array}$$
- L2 (unique deconstruction):
$$\begin{array}{lll}(a) & (\langle x\rangle = \langle y\rangle) & \equiv \quad (x = y)\\ (b) & (p \mid q = u|v) & \equiv \quad (p = u \wedge q = v)\\ (c) & (p \bowtie q = u \bowtie v) & \equiv \quad (p = u \wedge q = v)\end{array}$$
- L3 (commutativity):
$$(p \mid q) \bowtie (u \mid v) = (p \bowtie u) \mid (q \bowtie v)$$

These laws can be derived by suitably defining *tie* and *zip*, using the standard functions from the linear list theory. One possible strategy is to define tie as the concatenation of two equal length lists and then, use the Laws L0 and L3 as the definition of zip; Laws L1, L2 can be derived next. Alternatively, these laws may be regarded as axioms relating *tie* and *zip*.

Law L0 is often used in proving base cases of algebraic identities. Laws L1, L2 allow us to uniquely deconstruct a non-singleton powerlist using either $\mid$ or $\bowtie$. Law L3 is important since it defines the only possibility to relate the two construction operators, $\mid$ and $\bowtie$. Hence, it should be applied in proofs by structural induction where both constructors play a role.

Following the axioms, a powerlist could only have a length $2^n$, with $n \geq 0$. $n$ is called the *logarithmic length* of the powerlist.

Coq axiomatisation of powerlist is given in Figure 2. The powerlist data structure could not be defined as an inductive type as it would be in this case mandatory to define functions for all the possible constructors of the powerlist. It is in contradiction with the fact that it is convenient to define functions on powerlist using the singleton constructor and one of the other constructors, but not both. Therefore the powerlist data structure is modelled by the polymorphic type `powerlist` that takes the type of elements and the logarithmic length of the powerlist (a natural number, in Coq, the successor for values of type $\mathbb{N}$ is written `S`) as parameters. We naturally define the `singleton`, `zip` and `tie` operations, and associated notations, close to pen-and-paper notations.

The most important property is that any powerlist can be deconstructed. In Misra's axiomatisation, this is stated as the fact that all powerlists of logarithmic length 0 are singletons, and by axiom L2, stating that a non-singleton powerlist is either the *tie* or *zip* of two smaller powerlists. However axiom L2 is non-constructive in the sense that the existence of powerlists is stated, but these powerlists are not exhibited. In a Coq axiomatisation this would hinder the definition of actual functions on powerlists. As we cannot define them by induction on the structure of the powerlist (in our axiomatisation, powerlists are not defined as an inductive type), we need an effective way to deconstruct a powerlist. That is why we have the `unsingleton`, `unzip` and `untie` operations and their associated properties. Each property simply states that applying primitive $f$ to the result(s) of un-$f$ applied to a powerlist, actually yields the initial powerlist, with $f$ being `singleton`, `zip` or `tie`. These functions and their characteristic property indeed model axiom L1 in a more constructive way. Axiom L2 of Misra actually states that `singleton`, `zip` and `tie` are injective functions. This is modelled as three axioms in the Coq version. Using Coq notations, some axioms are very close to Misra's pen-and-paper axioms: This is the case for `L0` and for `L3`.

## IV. Inductive Reasoning on Powerlists

Based on the axiomatisation of the previous section, we now prove some results. When defining new inductive data structures in Coq, such as the list structure of section II, the Coq system automatically states and prove an associated induction principle. This induction principle is a theorem that is automatically applied when we use the `induction` tactics when writing proofs by induction.

It would be very convenient to have such and induction principle devoted to reason on powerlists. However the `powerlist` type is axiomatised, not defined by induction. Therefore the Coq system does not generated any induction principle automatically. Still we can ourselves define such an induction principle and proof it. We can do that in a functor, named here `Properties`. This module takes as argument a implementation of the `SPECIFICATION` module type (Figure 2), i.e. a realisation of the powerlist axiomatisation in Coq. This proposition states that to prove a property on all powerlists, it is sufficient to prove the property on singleton powerlists (1), and to prove the property on either the `zip` (2) *or* the `tie` (3) of two powerlists for which the property holds:

```
Module Properties (Import PL:TYPE).

  Proposition inductionPrinciple:
    ∀(A:Type)(P:∀(n:ℕ),powerlist A n→Prop),
      (∀(x:A), P 0 (⟨ x ⟩)) (* (1) *) →
      ( (∀(n:ℕ)(p q:powerlist A n),
           P n p → P n q →
           P (S n) (p⋈q))          (* (2) *) ∨
        (∀(n:ℕ)(p q:powerlist A n),
           P n p → P n q →
           P (S n) (p|q))          (* (3) *) ) →
      (∀(n:ℕ)(pl:powerlist A n), P n pl).
```

```
Module Type SPECIFICATION.

  Parameter powerlist:  Type → ℕ → Type.

  Parameter singleton: ∀{A:Type}, A → powerlist A 0.

  Global Notation "⟨ a ⟩" := (singleton a) (at level 30).

  Parameter unsingleton: ∀ {A:Type}(l:powerlist A 0), A.

  Axiom unsingletonProperty: ∀{A:Type}(l: powerlist A 0), l = ⟨ unsingleton l ⟩.

  Axiom singletonInjective: ∀{A:Type}(x y : A), ⟨ x ⟩  = ⟨ y ⟩ → x = y.

  Parameter zip : ∀{A:Type}{n:ℕ}, powerlist A n → powerlist A n → powerlist A (S n).

  Global Notation "l1 ⋈ l2" := (zip l1 l2) (at level 31).

  Axiom zipUnique: ∀{A:Type}{n:ℕ}(p q u v: powerlist A n), ( p ⋈ q = u ⋈ v) → p = u ∧  q = v.

  Parameter tie : ∀{A:Type}{n:ℕ}, powerlist A n → powerlist A n → powerlist A (S n).

  Global Infix  "|" := tie (at level 32).

  Axiom tieUnique: ∀{A:Type}{n:ℕ}(p q u v: powerlist A n), p | q = u | v → p = u ∧  q = v.

  Axiom L0: ∀(A:Type)(x y: A), ⟨ x ⟩ ⋈ ⟨ y ⟩ =  ⟨ x ⟩ | ⟨ y ⟩.

  Parameter unzip : ∀ {A:Type}{n:ℕ}, powerlist A (S n) → (powerlist A n)*(powerlist A n).

  Axiom unzipProperty: ∀ {A:Type}{n:ℕ}(l:powerlist A (S n)),
    let p := unzip l in l = (fst p) ⋈ (snd p).

  Parameter untie : ∀ {A:Type}{n:ℕ}, powerlist A (S n) → (powerlist A n)*(powerlist A n).

  Axiom untieProperty: ∀ {A:Type}{n:ℕ}(l:powerlist A (S n)),
    let p := untie l in l = tie (fst p) (snd p).

  Axiom L3: ∀{A:Type}{n:ℕ}(p q u v: powerlist A n), (p | q) ⋈ (u | v) = (p ⋈ u) | (q ⋈ v).

End SPECIFICATION.
```

Figure 2. Powerlists in Coq

```
  Proof.
    intros A P Hsingleton H n.
    induction n; intro pl.
    - rewrite unsingletonProperty.
      apply Hsingleton.
    - destruct H as [H | H].
      + rewrite unzipProperty.
        now apply H.
      + rewrite untieProperty.
        now apply H.
  Qed.

End Properties.
```

The proof is done by induction on the logarithmic length of the powerlist. This induction principle is the first induction principle on powerlists that is given in [13, page 25]. This induction principle is quite weak and it is of no use when we need to reason about sub-powerlists which logarithmic sizes are not just the predecessor of the logarithmic size of the powerlist on which the property should be proved. Of course more involved principles could be proved and we provide another in the Coq source related to this paper.

But as a matter of fact, as the proof of the induction principle above shows, it is quite convenient to reason by induction on the logarithmic size of the powerlists. For more general induction principles it is very easy in Coq to use one of the induction principles on naturals, provided by the Coq library. In particular the following induction principle is very useful:

```
Lemma lt_wf_ind : ∀ (n:ℕ) (P:ℕ→Prop),
     (∀ n':ℕ, (∀ m:ℕ, m<n'→P m)→P n')→P n
```

## V. Programming with Powerlists

### A. Basic Examples

Based on the axiomatisation of section III, we show how to program functions on powerlists. It is convenient to use the `Program` feature of Coq because `powerlist` is a dependent type including the logarithmic length of the powerlist. In many

cases we have to prove some equivalences between logarithmic lengths or combination of logarithmic lengths of the manipulated powerlists. Without the `Program` feature this would pollute the writing of functions on powerlists. Moreover the `Program` library is able to automatically prove the generated obligations in the simple following cases.

We first define the function `map` on `powerlists`:

```
Program Fixpoint map
  {A B:Type}(f:A→B){n:ℕ}(pl:powerlist A n):
  powerlist B n :=
  match n with
    | 0 ⇒ let x := unsingleton pl in
           ⟨ f x ⟩
    | S n ⇒ let pair:=unzip pl in
            let (p,q):=(fst pair,snd pair) in
            map f p ⋈ map f q
  end.
```

This recursive definition is written by induction on the logarithmic length of the powerlist. In both cases we need to deconstruct the argument powerlist. In case of a singleton powerlist we use the `unsingleton` operator, in case of a non-singleton list we use the `unzip` operator and we combine the results of the two recursive calls using the `zip` powerlist constructor. A pen-and-paper definition of `map` could be:

$$\begin{array}{rcl} \text{map } f\ \langle x\rangle & = & \langle f\ x\rangle \\ \text{map } f\ (p \bowtie q) & = & \text{map } f\ p \bowtie \text{map } f\ q \end{array}$$

In the Coq definition we choose a style where we explicit all the types: we could be less verbose and let the system infers some types. The lines beginning by `let` are however necessary as the `powerlist` type is abstract.

In the same way, we could define the `rev` function:

```
Program Fixpoint rev
  {A:Type}{n:ℕ}(pl:powerlist A n) :
  powerlist A n :=
  match n with
    | 0 ⇒ pl
    | S n ⇒ let pair:=unzip pl in
            let (p,q):=(fst pair,snd pair) in
            rev q ⋈ rev p
  end.
```

As an illustration of the usage of the induction principle of section IV, we show that `map` and `rev` commute:

```
Lemma maprev:
  ∀(A B:Type)(f:A→B)(n:ℕ)(pl:powerlist A n),
    map f (rev pl) = rev(map f pl).
Proof. intros A B f.
  apply inductionPrinciple.
  - intros x. simpl. trivial.
  - left. intros n p q Hp Hq.
    simpl. repeat rewrite zipUnZip; simpl.
    rewrite Hp, Hq. reflexivity.
Qed.
```

The `zipUnzip` lemma states that `unzip (p ⋈q) = (p, q).`

`map` could have been defined using `untie` and `tie`. It is possible to write such a map function and prove the equivalence with the map we defined, or similarly we could prove that:

```
Lemma mapZipComm:
 ∀ {A B:Type}(f:A→B){n:ℕ}(p q:powerlist A n),
   map f (p ⋈ q) = (map f p) ⋈ (map f q).
```

The induction principle defined in the previous section is not enough to prove this result. We give in the following the full proof of this lemma, to show that we defined a specific tactic named `pl_simpl` used to simplify powerlist expressions in particular compositions of zip, unzip or tie, untie or singleton, unsingleton, and to show that the proof for the non-trivial case follows exactly the structure of the pen-and-paper proof (for e.g. in [, ]). To be even closer to the pen-and-paper proof we could have used the tactics proposed in [14].

```
Proof.
  intros A B f. induction n using lt_wf_ind.
  intros p q. destruct n.
  - repeat (pl_simpl; rewrite L0).
  - set(p1:=fst(untie p)).
    set(p2:=snd(untie p)).
    set(q1:=fst(untie q)).
    set(q2:=snd(untie q)).
    replace (map f p ⋈ map f q)
    with     (map f (p1|p2) ⋈ map f (q1|q2))
      by (unfold p1, p2, q1, q2; simpl;
          repeat rewrite <- untieProperty;
          fold p1; trivial).
    replace (map f (p1|p2) ⋈ map f (q1|q2))
    with     (map f (p1⋈q1) | map f (p2⋈q2))
      by (simpl; rewrite L3;
          repeat rewrite <-H; auto;
          simpl; repeat rewrite tieUntie;
          auto).
    replace (map f (p1⋈q1) | map f (p2⋈q2))
    with     (map f ((p1⋈q1)|(p2 ⋈ q2)))
      by (pl_simpl).
    replace ( (p1⋈q1) | (p2⋈q2) )
    with     ( (p1|p2) ⋈ (q1|q2) )
      by (apply L3).
    unfold p1, p2, q1, q2;
    repeat rewrite <- untieProperty;
    trivial.
Qed.
```

As a final basic example, let us formalise one of the functions defined in [15, page 62]:

$$\begin{array}{ll} \text{join}\, n\, (p|q)\, (r|s) = p\,|\,s & \text{if } n{=}\#p \\ \text{join}\, n\, (p|q)\, (r|s) = \text{join}\, n\, p\, r\,|\,\text{join}\, n\, p\, s & \text{if } n{<}\#p \end{array} \quad (1)$$

where $\#p$ is the length of $p$. This function takes two powerlists of same length, and returns a new powerlist consisting of alternate slices of the input powerlists, each of length $n = 2^i$ with $0 \leq i < \log_2\ \#p$.

The `join` function is a partial function. There are at least four ways to deal with that in Coq where we can only define total functions.

The first possibility is to add pre-conditions on appropriate parameters. In the case of `join`, the `n` argument of type $\mathbb{N}$ could be replaced by `(n:ℕ|n<=sp)` where `sp` represents the length of `p`. However using such a parameter with a dependent type makes the definition either much more complicated to write, without using the `Program` feature of Coq, or is difficult to manipulate once it is defined using `Program` as it would contain complicated dependent pattern matching.

The second possibility is to return an "optional" value using a type similar to OCaml `'a option` type or Haskell `maybe` type. But then function composition is not so convenient, as one needs to test if the result is `Some x` and pattern match to get `x` before applying another function. A Haskell-style monadic notation could ease the writing of compositions. However the new cases introduced by the optional type are to be considered when reasoning on it.

The third possibility is to add an additional argument, a default value to be returned when the input values are outside the domain. A variant of this possibility is to extend the function to a total function without needing such an additional argument. This is the solution we chose, as `join` has an intuitive extension to the full domain.

Apart from the totalisation of `join`, its definition is a bit different than the paper definition in order to accommodate in a simple way the fact that our powerlist type contains the logarithmic length of the powerlist, and also to have a definition which is structurally recursive on the logarithmic size of the input powerlists. In the paper, for the definition given by equations (1), the argument that this recursive definition actually terminates is that the measure (length of $p$ minus $n$) is decreasing. It is actually possible to define the function in a similar way in Coq, but as for the pre-condition on the parameter, using the `Program` feature, this would lead to a definition quite difficult to manipulate.

```
Program Fixpoint join {A:Type}
  (n:ℕ){sp:ℕ}(p₁ p₂:powerlist A (S sp)) :
  powerlist A (S sp) :=
  let p:=fst(untie p₁) in
  let q:=snd(untie p₁) in
  let r:=fst(untie p₂) in
  let s:=snd(untie p₂) in
  if n ≤_d sp then
    if n =_d sp then p | s
    else match sp with
         | 0 ⇒ p₁
         | S l ⇒ join n p r | join n q s
         end
  else p₁.
```

In order to mimic the paper proofs later on, we can prove lemmas on `join` that are indeed the equations (1). The proofs of these lemmas are made very simple by the definition of a tactic to handle easily functions defined using `if`:

```
Lemma joinLt: ∀{A:Type}(m:ℕ){n:ℕ}
   (p q r s:powerlist A (S n)),
  m < (S n) →
  join m (p|q) (r|s)=(join m p r)|(join m q s).
Proof.
  intros A m n; intros;
  destruct n; destruct m; if_simpl.
Qed.

Lemma joinEq:
  ∀{A:Type}(m:ℕ){n:ℕ}(p q r s:powerlist A n),
    m = n →
    join n (p | q) (r | s) = p | s.
Proof.
  intros A m n; intros;
  destruct n; destruct m; if_simpl.
Qed.
```

### B. Sorting

Divide & conquer algorithms on powerlists can be very easily expressed. In this section we discuss the implementation and proofs of two such sorting algorithms. Both are based on a basic merge sort principle:

```
Program Fixpoint sort {n:ℕ}(p:powerlist A n) :
  powerlist A n :=
  match n with
    | O ⇒ p
    | S m ⇒ let pair := untie p in
            let (p,q):=(fst pair, snd pair) in
            merge (sort p) (sort q)
  end.
```

where `merge` is a function that, applied to two sorted powerlists, returns a sorted powerlist of every element in its arguments. Misra gives the expression of two such functions, initially described as sorting networks in [16]. Both functions use the following comparison operation:

```
Definition comp {n:ℕ}(p₁ p₂: powerlist A n) :
  powerlist A (S n) :=
  (plmin p₁ p₂) ⋈ (plmax p₁ p₂).
```

`plmin` (resp. `plmax`) is a function that takes two lists $p$ and $q$ of equal length and returns the list containing, for each $i$, $\min\, p_i\, q_i$ (resp. $\max\, p_i\, q_i$).

The first merging algorithm, called batcher merge, is given by the following function:

```
Program Fixpoint bm {n:ℕ}(p q:powerlist A n):
  powerlist A (S n) :=
  match n with
    | O ⇒ comp p q
    | S m ⇒
      let p₁ := fst (unzip p) in
      let p₂ := snd (unzip p) in
      let q₁ := fst (unzip q) in
      let q₂ := snd (unzip q) in
      comp (bm p₁ q₂) (bm p₂ q₁)
  end.
```

Another merging scheme uses the bitonic sort. A bitonic sort takes a bitonic list as argument and returns a sorted list. A bitonic list is one that is constructed by appending a sorted non-decreasing list and a sorted non-increasing list (appending is here to be taken as the usual definition for linear lists) or any rotation of such a list. If two lists `p` and `q` are sorted, then the list `p|rev q` is trivially bitonic, so we can use `b_sort` to merge lists.

Misra gives an expression of the bitonic sort, written in Coq as follow:

```
Program Fixpoint b_sort {n: ℕ}
  (p: powerlist A n) : powerlist A n :=
  match n with
    | O ⇒ p
    | S m ⇒
      let r := fst (unzip p) in
      let s := snd (unzip p) in
      comp (b_sort r) (b_sort s)
  end.
```

The original description of the bitonic sort, as a sorting network, is slightly different from the function `b_sort` above. Instead it translates as the following function on powerlists:

```
Program Fixpoint b_sort' {n: ℕ}
  (p: powerlist A n) : powerlist A n :=
  match n with
    | O ⇒ p
    | S m ⇒
      let r := fst (untie p) in
      let s := snd (untie p) in
      b_sort' (plmin r s) | b_sort' (plmax r s)
  end.
```

It is thankfully possible to prove the equality of both expressions.

We can prove that `bm p q = b_sort (p|rev q)`. The formal proof of this lemma precisely follows the informal one given in [5]. Therefore properties of sorting and permutation need only be proven for one of the two functions.

Some care must be taken when expressing inductive properties on powerlists, due to the dependence of type `powerlist` on type ℕ. For the same reason, inversion of such properties may produce undesirable artefacts. Thankfully Coq provides a language to write custom tactics, a feature that was used in order to provide transparent inversion of inductive properties such as sorting or permutation of powerlists.

The Coq proof that `bm` produces a permutation of its two arguments is relatively straightforward. One only needs to prove this property of all the functions involved, namely `comp`, `plmin` and `plmax`.

For the proof that `bm` outputs a sorted list, we cannot reuse the proof of Misra. This proof uses the fact that "compare and swap" sorting algorithms are correct if and only if they are correct on lists containing only zeros and ones. In order to reuse this theorem, we would have to express the concept of "compare and swap" algorithm formally, and of course provide a Coq proof of the result. However, to characterise such algorithms, we would have to formalise the syntax and semantics of algorithms, as inductive definitions, and the algorithms could no longer be defined as functional programs. Such a formalisation allows a broader range of reasoning, but it much more complex and is then a deep embedding rather than a shallow embedding [17]. Therefore we preferred to stick to the view of Coq as a rich functional programming language and we relied on a more basic approach to the proof. We start by proving the correctness of the bitonic sort, closely following the proof given by Batcher, then we rewrite `bm` as an application of `b_sort` to prove its correctness.

The full Coq code is available at:
http://traclifo.univ-orleans.fr/SDPP/wiki/PowerLists

## VI. Related Work

Powerlist data structure could be extended to ParList and PList structures in order to deal with lists of ordinary length, respectively to deal with more general variants for division – in the context of divide & conquer paradigm – beside binary decomposition. These data structures can be easily extended to the multidimensional case to allow the specification of recursive parallel programs that work with multidimensional data in a more efficient and suggestive way. All these theories can be considered the base for a model of parallel computation with a very high level of abstraction. In [18] a model PARES (*Parallel Recursive Structures*) is proposed that includes all these theories, together with the data-distributions defined on them, that allows the definition of parallel programs with different scales of granularity. The programs are defined as functions on the specified structures and their correctness is assured by using structural induction principles.

The model is very appropriate to shape programs based on divide & conquer paradigm, but this is not its only advantage. The division partition is controlled in order to obtain a good work-balance, and this is done in a formalized way (based on defined algebras). Division could be done in two equal parts (powerlist, PowerArray), different number of equal parts (PList, PArray), or almost equal parts (ParList,ParArray). Still, other paradigms such as direct parallelization ("embarrassingly parallel computations", ParFor) and pipeline, could also be expressed in this model.

In order to allow sequential computation to be also expressed in this model, we may combine classical functional programming with the model described before. Misra also suggested the combining sequential and parallel computation inside these theories: "A mixture of linear lists and powerlists can exploit the various combinations of sequential and parallel computing. A powerlist consisting of linear lists as components admits of parallel processing in which each component is processed sequentially (PAR-SEQ). A linear list whose elements are powerlists suggests a sequential computation where each step can be applied in parallel (SEQ-PAR). Powerlists of powerlists allow multidimensional parallel computations, whereas a linear list of linear lists may represent a hierarchy of sequential computations" [5].

The possibility of using powerlists to prove the correctness of several algorithms has encouraged some researchers to pursue automated proofs of theorems about powerlists. Kapur and Subramaniam [19] have implemented the powerlist notation for the purpose of automatic theorem proving. They have proved many of the algorithms described by Misra using an inductive theorem prover, called RRL (Rewrite Rule Laboratory). They use powerlist structures to prove some of the theorems from [5], but the theorems themselves, are designed to simplify the powerlist proofs, rather than to certify an algorithm's correctness with respect to an absolute specification. In [20] adder circuits specified using powerlists are proved correct with respect to addition on the natural numbers. The attempt done in [21] is closer to our approach. There it is shown how ACL2 can be used to verify theorems about powerlists. Still, the considered powerlists are not the regular structures defined by Misra, but structures corresponding to binary trees, which are not necessarily balanced. Our approach is closer to pen-and-paper reasoning. One stated advantage of the ACL2 formalisation is that functions could be evaluated. In our formalisation, up to a point, symbolic computations could be performed. It is also of course possible to implement the `SPECIFICATION` module type, for example using usuals lists and operations on them. In this case, computation could be performance inside the Coq proof assistant and moreover the extraction mechanism of Coq could be used to obtain more efficent functional programs.

BMF formalism [22], [23] is based on recursion too, and there the notion of homomorphism is essential. It could be considered in a way more abstract than the powerlist theories, however this higher order of abstraction may also implies some difficulties in developing efficient solutions. In BMF the data-distributions have been introduced as simple functions that transform a list into a list of lists. However since the interleaving operator (zip) could be very useful in developing some efficient parallel programs, an extension towards powerlists has been proposed in [24], [25], by restricting list concatenation to lists of the same length. They have been categorised as *distributable homomorphisms*. In [6], [14], we present a new kind of homomorphism called Bulk Synchronous Parallel homomorphism or BH, as well as a Coq framework for reasoning in the BMF style in Coq. Combined with a verified (in Coq) implementation of a BH Skeleton with the functional parallel programming language BSML [26], [27], in this framework we can derive a program from a specification, to OCaml [9] and BSML code, compiled to native-code and run on parallel machines.

## VII. Conclusion and Future Work

The Coq proof assistant is an adequate tool for programming and reasoning with powerlists, and in general with parallel recursive structures. The work presented in this paper is a first, but necessary, step. The next step is to provide a parallel implementation of the basic building blocks of powerlists, that may be not (only) the constructors. This implementation should be proved correct with respect to the specification that could be either the axiomatisation of powerlists or other derived elements from this axiomatisation. We can then use the Coq extraction mechanism to obtain real parallel programs and experiment with them. We will first focus on a BSML [26], [27] implementation.

Future research also include work on other parallel recursive structures: ParLists, PLists, PowerArrays, ParArrays and PArrays. For all these structures, additional induction principles may be considered. Misra [5] considers inductive reasoning based on the `depth` and `lgl` (logarithmic length) functions, the former being defined by recursion *on the type of elements* of the powerlist. Modelling such a function in Coq in a way that is both general and convenient to use is not a trivial task, as it requires modifying the whole modelling of powerlists.

## Acknowledgements

This work is partly supported by the INEX project funded by the *Conseil Général du Loiret*.

## References

[1] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development*. Springer, 2004.

[2] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS 2283. Springer, 2002.

[3] P. Letouzey, "Coq Extraction, an Overview," in *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, ser. LNCS 5028, A. Beckmann, C. Dimitracopoulos, and B. Löwe, Eds. Springer, 2008.

[4] M. Cole, "Parallel Programming with List Homomorphisms," *Parallel Processing Letters*, vol. 5, no. 2, pp. 191–203, 1995.

[5] J. Misra, "Powerlist: A structure for parallel recursion," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1737–1767, November 1994.

[6] L. Gesbert, Z. Hu, F. Loulergue, K. Matsuzaki, and J. Tesson, "Systematic Development of Correct Bulk Synchronous Parallel Programs," in *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 2010, pp. 334–340.

[7] W. A. Howard, "The formulae-as-types notion of construction," in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin and J. R. Hindley, Eds. Academic Press, 1980, pp. 479–490.

[8] T. Coquand and G. Huet, "The Calculus of Constructions," *Information and Computation*, vol. 37, no. 2-3, 1988.

[9] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, "The OCaml System release 4.00.0," http://caml.inria.fr, 2012.

[10] B. O'Sullivan, D. Stewart, and J. Goerzen, *Real World Haskell*. O'Reilly, 2008.

[11] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[12] N. Javed and F. Loulergue, "A Formal Programming Model of Orléans Skeleton Library," in *11th International Conference on Parallel Computing Technologies (PaCT)*, ser. LNCS 6873, V. Malyshkin, Ed. Springer, 2011, pp. 40–52.

[13] J. Kornerup, "Data structures for parallel recursion," Ph.D. dissertation, University of Texas, 1997.

[14] J. Tesson, H. Hashimoto, Z. Hu, F. Loulergue, and M. Takeichi, "Program Calculation in Coq," in *Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST2010)*, ser. LNCS 6486. Springer, 2010, pp. 163–179.

[15] K. Achatz and W. Schulte, "Massive parallelization of divide-and-conquer algorithms over powerlists," vol. 26, pp. 59–78, 1996.

[16] K. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.

[17] M. Wildmoser and T. Nipkow, "Certifying machine code safety: Shallow versus deep embedding," in *Theorem Proving in Higher Order Logics*, ser. LNCS, K. Slind, A. Bunker, and G. Gopalakrishnan, Eds. Springer Berlin / Heidelberg, 2004, vol. 3223, pp. 133–142.

[18] V. Niculescu, "PARES – A Model for Parallel Recursive Programs," *Romanian Journal of Information Science and Technology (ROMJIST)*, vol. 14, no. 2, pp. 159–182, 2011. [Online]. Available: http://www.imt.ro/romjist/Volum14/Number14_2/pdf/06-VNiculescu.pdf

[19] D. Kapur and M. Subramaniam, "Automated reasoning about parallel algorithms using powerlists," State University of New York at Alban, Tech. Rep. TR-95-14, 1995.

[20] D. Kapur and P. Narendran, "Matching, unification and complexity." *Journal of Automated Reasoning*, 1988, preprint.

[21] R. A. Gamboa, "A formalization of powerlist algebra in ACL2," *J. Autom. Reason.*, vol. 43, no. 2, pp. 139–172, 2009.

[22] R. S. Bird, "An introduction to the theory of lists," in *Logic of Programming and Calculi of Discrete Design*, M. Broy, Ed. Springer-Verlag, 1987, pp. 3–42.

[23] D. Skillicorn, *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[24] S. Gorlatch, Ed., *First International Workshop on Constructive Methods for Parallel Programming (CMPP'98)*, ser. Research Report MIP-9805. University of Passau, May 1998.

[25] S. Gorlatch, "SAT: A Programming Methodology with Skeletons and Collective Operations," in *Patterns and Skeletons for Parallel and Distributed Computing*, F. A. Rabhi and S. Gorlatch, Eds. Springer, 2003, pp. 29–64.

[26] F. Loulergue, F. Gava, and D. Billiet, "Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction," in *ICCS*, ser. LNCS, V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds., vol. 3515. Springer, 2005, pp. 1046–1054.

[27] J. Tesson and F. Loulergue, "A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation," in *International Conference on Computational Science (ICCS)*, ser. Procedia Computer Science. Elsevier, 2011, pp. 36–45.