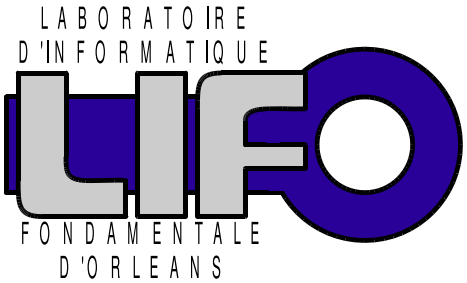

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Programming and Reasonning with PowerLists in Coq

Frédéric Loulergue
Virginia Niculescu

Rapport n^o **RR-2013-02**

Programming and Reasonning with PowerLists in Coq

Frédéric Loulergue[1] Virginia Niculescu[2]

March 2013

[1] LIFO, Université d'Orléans, France,
Frederic.Loulergue@univ-orleans.fr
[2] Faculty of Mathematics and Computer Science
Babeş-Bolyai University, Cluj-Napoca
vniculescu@cs.ubbcluj.ro

Abstract

For parallel programs correctness by construction is an essential feature since debugging is almost impossible. To build correct programs by constructions is not a simple task, and usually the methodologies used for this purpose are rather theoretical 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 implementing easily simple programs satisfying conditions, ideally automatically, proved. PowerLists theory and the variants represent a good theoretical base for an approach like this, and 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 on parallel programs in Coq. This represents the first step in building a framework that ease the development of correct and verifiable parallel programs.
Keywords: Parallel recursive structures, interactive proofs, Coq

1 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 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 [19], to be able to write and reason on programs, and (3) axiomatisation of lower level parallel programming primitives and their use to implement the high-level primitives in order to extract [15] actual *parallel* code from the developments make inside proof assistants.

Among high-level algebraic theories, at least two seem suitable for parallel programming: the theory of lists [3] and parallel recursive structures such as PowerLists [17]. The SDPP framework [6], a partial realisation of our long term goal, currently focuses on the theory of lists. PowerList and variants are data structures introduced by J. Misra and J. Kornerup, which can be successfully used in a simple and provable 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 program and reason on 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 2) before discussing PowerList axiomatisation (section 3). From this axiomatisation, we proof an induction principle (section 4) and use it to reason about some functions, also defined using the axiomatisation (section 5). We discuss related work (section 6) before concluding (section 7).

2 Introduction to Coq

Coq [1] is a proof assistant based on the Curry-Howard correspondence [9] relating terms of a typed λ-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 [4].

From a more practical side, Coq can be seen as a functional programming language, close to OCaml [14] or Haskell [20] 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 [13].

In order to discuss our modelling of PowerLists in Coq, we first introduce the Coq main features we use, through examples on the list data structure. This short introduction complements the one in [10].

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.
```

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 λ-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 nat := [ 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).
```

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 λ-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
```

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

```
  ============================
   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 λ-term associated 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 fill 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 *module* could realise these module types. It is also possible to make a module (or module type) parametrised by another module: We have then 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, and lemmas, proposition, theorem are replaced by `Axiom`s. Instead of defining by induction a data structure of join-lists we could use the cons-list we have to implement this module type. 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 localised 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.

3 Axiomatisation of Power Lists

Many of the known parallel algorithms – FFT, Batcher Merge, prefix sum, embedding arrays in hypercubes, etc. – have concise descriptions using PowerLists. PowerLists have simple algebraic properties which permit properties deduction 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*: $p\#q$ 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

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

According to J. Misra [17], a general recursive definition of PowerLists is:
If S is a scalar, P is a PowerList, and u, v are similar PowerLists then

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

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 for constructing the same list using *tie* or *zip*. (From this we have two induction principles that could be equally used.) But, 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) $\langle x\rangle|\langle y\rangle = \langle x\rangle\#\langle y\rangle$

L1 (dual deconstruction) if P is a non-singleton PowerList there exist r, s, u, v similar PowerLists such that:

(a) $P = r \mid s$

(b) $P = u \# v$

L2 (unique deconstruction)

(a) $(\langle x\rangle = \langle y\rangle) \equiv (x = y)$

(b) $(p \mid q = u|v) \equiv (p = u \wedge q = v)$

(c) $(p \# q = u\#v) \equiv p = u \wedge q = v)$

L3 (commutativity) $(p \mid q) \# (u \mid v) = (p \# u) \mid (q \# v)$

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 $|$ or $\#$. Law L3 is important since it defines the only possibility to relate the two construction operators, $|$ and $\#$. 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 1. 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 of the fact that it is convenient to define functions on PowerList using the singleton constructor and one of this 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 `nat` is written `S`). We naturally define the `singleton`, `zip` and `tie` operations, and associated notation, close to pen-and-paper notations.

The most important property is that for any PowerList, we should be able to deconstruct it. In Misra's axiomatisation, this is stated as the fact that all PowerList of logarithmic length 0 are singletons, and by axiom L2 that a non-singleton PowerList is either the *tie* or *zip* of two smaller PowerLists. However axiom L2 is non-constructive in the sense an existence of PowerLists is stated, but these PowerLists are not exhibited. In a Coq axiomatisation it 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, PowerList 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 in a more constructive way axiom L1. 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`.

4 Inductive Reasoning on Power Lists

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 2, 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 `POWERLIST` module type (Figure 1), 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:nat), powerlist A n → Prop),
      (∀(x:A), P 0 (<| x |>)) (* (1) *) →
      ( (∀(n:nat)(p q:powerlist A n),
           P n p → P n q → P (1+n) (p#q)) (* (2) *) ∨
        (∀(n:nat)(p q:powerlist A n),
           P n p → P n q → P (1+n) (p!q)) (* (3) *) ) →
      (∀(n:nat)(pl:powerlist A n), P n pl).
  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 short proof is done by induction on the logarithmic length of the PowerList. Of course more involved principles could be proved. Other principles could be found in the Coq source related to this paper.

5 Programming with Power Lists

Based on the axiomatisation of section 3, 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

```
Module Type POWERLIST.
 Parameter powerlist: Type → nat → Type.
 Parameter singleton: ∀{A:Type}, A → powerlist A 0.
 Notation "<| a |> " := (singleton a) (at level 70).
 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:nat},
     powerlist A n → powerlist A n → powerlist A (1+n).
 Notation "l1 # l2" := (zip l1 l2) (at level 40).
 Axiom zipUnique: ∀{A:Type}{n:nat}(p q u v: powerlist A n),
    ( p # q = u # v) → p = u ∧  q = v.
 Parameter tie: ∀{A:Type}{n:nat},
     powerlist A n → powerlist A n → powerlist A (1+n).
 Notation "l1 ! l2" := (tie l1 l2) (at level 40).
 Axiom tieUnique: ∀{A:Type}{n:nat}(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:nat},
     powerlist A (S n) → (powerlist A n)*(powerlist A n).
 Axiom unzipProperty: ∀ (A:Type)(n:nat)(l:powerlist A (S n)),
     let p := unzip l in l = (fst p) # (snd p).
 Parameter untie: ∀ {A:Type}{n:nat},
     powerlist A (S n) → (powerlist A n)*(powerlist A n).
 Axiom untieProperty: ∀ (A:Type)(n:nat)(l:powerlist A (S n)),
     let p := untie l in l = tie (fst p) (snd p).
 Axiom L3: ∀{A:Type}{n:nat}(p q u v: powerlist A n),
     ( p ! q ) # ( u ! v ) = ( p # u ) ! ( q # v ).
End POWERLIST.
```

Figure 1: PowerList in Coq

have to proof 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}{n:nat}(f:A→B)(l:powerlist A n):
  powerlist B n:=
  match n with
    | 0 ⇒ <| f (unsingleton l) |>
    | S n ⇒ let p:= unzip l in (map f (fst p))#(map f (snd p))
  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. `map` could have been defined using `untie` and `tie`. Using an adequate induction principle, we show the equivalence of these two definitions.

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

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

As an illustration of the usage of the induction principle of section 4, we show that `map` and `rev` commute:

```
Lemma maprev:
  ∀(A B:Type)(f:A→B)(n:nat)(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)`.

The full Coq code is available at:

`http://traclifo.univ-orleans.fr/SDPP/wiki/PowerLists`

6 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 works with multidimensional data in 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 allow the definition of parallel programs with different scale 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 for shapely programs based on divide & conquer paradigm, but not only. The division partition is controlled in order to obtain a good work-balance, and this is done in 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 combination between 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" [17].

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[12] 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 [17], 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 [11] adder circuits specified using PowerLists are proved correct with respect to addition on the natural numbers. The attempt done in [5] 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. Also, the approach is illustrated in two case studies involving Batcher sorting and prefix sums, but it is not proved as it could be used as a general framework.

BMF formalism [2, 21] is based on recursion too, and there the notion of homeomorphism is essential. It could be considered in a way more abstract then 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. But, since interleaving operator (zip) could be very useful in developing some efficient parallel program, an extension towards PowerLists has been proposed in [7, 8], by restricting list concatenation to lists of the same length. They have been categorised as *distributable homomorphisms*. In [6, 22], 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 [16, 23], with this framework we can derive a program from a specification to OCaml [14] and BSML code, compiled to native-code and run on parallel machines.

7 Conclusion and Future Work

The Coq proof assistant is an adequate tool for programming and reasonning 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 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 form 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 [16, 23] 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 [17] considers inductive reasoning based on the `depth` and functions, the former being defined by recursion on the type of elements of the PowerList. Modeling in Coq such a function in a general but still convenient to use way is not a trivial task, as it requires to modify all the modelling of PowerLists.

References

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

[2] Bird, R.S.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design, pp. 3–42. Springer-Verlag (1987)

[3] Cole, M.: Parallel Programming with List Homomorphisms. Parallel Processing Letters 5(2), 191–203 (1995)

[4] Coquand, T., Huet, G.: The Calculus of Constructions. Information and Computation 37(2-3) (1988)

[5] Gamboa, R.A.: A formalization of powerlist algebra in acl2. J. Autom. Reason. 43(2), 139–172 (Aug 2009), `http://dx.doi.org/10.1007/s10817-009-9140-y`

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

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

[8] Gorlatch, S.: SAT: A Programming Methodology with Skeletons and Collective Operations. In: Rabhi, F.A., Gorlatch, S. (eds.) Patterns and Skeletons for Parallel and Distributed Computing, pp. 29–64. Springer (2003)

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

[10] Javed, N., Loulergue, F.: A Formal Programming Model of Orléans Skeleton Library. In: Malyshkin, V. (ed.) 11th International Conference on Parallel Computing Technologies (PaCT). pp. 40–52. LNCS 6873, Springer (2011)

[11] Kapur, D., Narendran, P.: Matching, unification and complexity. Journal of Automated Reasoning (1988), preprint

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

[13] Leroy, X.: Formal verification of a realistic compiler. CACM 52(7), 107–115 (2009)

[14] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml System release 4.00.0. `http://caml.inria.fr` (2012)

[15] Letouzey, P.: Coq Extraction, an Overview. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008. LNCS 5028, Springer (2008)

[16] Loulergue, F., Gava, F., Billiet, D.: Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS. LNCS, vol. 3515, pp. 1046–1054. Springer (2005)

[17] Misra, J.: Powerlist: A structure for parallel recursion. ACM Transactions on Programming Languages and Systems 16(6), 1737–1767 (November 1994)

[18] Niculescu, V.: PARES – A Model for Parallel Recursive Programs. Romanian Journal of Information Science and Technology (ROMJIST) 14(2), 159–182 (2011), `http://www.imt.ro/romjist/Volum14/Number14_2/pdf/06-VNiculescu.pdf`

[19] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS 2283, Springer (2002)

[20] O'Sullivan, B., Stewart, D., Goerzen, J.: Real World Haskell. O'Reilly (2008)

[21] Skillicorn, D.: Foundations of Parallel Programming. Cambridge University Press (1994)

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

[23] Tesson, J., Loulergue, F.: A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation. In: International Conference on Computational Science (ICCS). pp. 36–45. Procedia Computer Science, Elsevier (2011)