Parallel Processing Letters
© World Scientific Publishing Company

FORMAL PROOFS OF FUNCTIONAL BSP PROGRAMS

FRÉDÉRIC GAVA
Laboratory of Algorithms, Complexity and Logic, 61, avenue du Général de Gaulle
94010 Créteil Cedex, France

Received April 2003
Revised July 2003
Communicated by Gaétan Hains and Frédéric Loulergue

ABSTRACT
The Bulk Synchronous Parallel ML (BSML) is a functional language for BSP programming, a model of computing which allows parallel programs to be ported to a wide range of architectures. It is based on an extension of the ML language by parallel operations on a parallel data structure called parallel vector, which is given by intention. We present a new approach to certifying BSML programs in the context of type theory. Given a specification and a program, an incomplete proof of the specification (of which algorithmic contents corresponds to the given program) is built in the type theory, in which gaps would correspond to the proof obligation. This development demonstrates the usefulness of higher-order logic in the process of software certification of parallel applications. It also shows that the proof of rather complex parallel algorithms may be done with inductive types without great difficulty by using existing certified programs. This work has been implemented in the Coq Proof Assistant, applied on non-trivial examples and is the basis of a certified library of BSML programs.

Keywords: Formal proofs; BSP; Functional language; Coq.

1. Introduction

Some problems require performance that can be provided only by massively parallel computers. Programming this kind of computer is still difficult. Bulk-Synchronous Parallel ML or *BSML* is an extension of ML for programming *direct-mode* Bulk-Synchronous Parallel algorithms as functional programs. *Bulk-Synchronous* Parallel (BSP) computing is a parallel programming model introduced by Valiant [23], designed to offer a high degree of abstraction as in PRAM models, yet to allow *portable* and *predictable* performance on a wide variety of architectures (for the sake of conciseness, we refer to [19] for more details). Such algorithms offer predictable and *scalable* performance and, in addition, BSML expresses them with a small set of primitives taken from the *confluent* BSλ calculus [15]: a constructor of parallel vectors, asynchronous parallel function application, synchronous global communications and a synchronous global conditional. This framework is a good tradeoff for parallel programming and proofs because we can design *purely* functional parallel languages. Without side-effects, programs are easier to prove, and to re-use for their formal properties (the semantics is compositional). We can also choose any *evaluation* strategy for the proof of their obligations. Based on BSP operations, programs are easy to port and their costs can be predicted (they are parametrized by the BSP parameters of the target architecture).

The `BSMLlib` library implements the BSML primitives using Objective Caml and MPI. This library is used as the basis for the CARAML project, which aims to use Objective Caml for grid computing. In such a context, *security* is an important issue, but in order to obtain security, *safety* must be first achieved. A polymorphic type system [9] which prohibits nesting of parallelism has been designed, for which type inference is possible. But it is not sufficient for critical programs which need *formal proofs* of the algorithms or for the libraries which need to be certified, i.e., programs must be shown to compute what is expected from them. A large part of the programming task is devoted to verifying that a program execution satisfies the programmer's intentions. Testing the programs on various inputs can help detect errors and increase our confidence, whereas only formal *mathematical methods* can guarantee *correctness.* A program is then documented and more easily portable.

The **Coq** System [1] is an environment for proof development based on the Calculus of Inductive Construction [6], a typed λ-calculus, extended with *inductive* definitions. Type theory is a particular framework for program development, well-suited for the proof of correctness of purely functional programs [17] and also provides expressive logics. Thus, the **Coq** Proof Assistant is a language of terms and types where terms can be seen as representing proofs or programs and types as representing *specifications* or data types.

Indeed, there is a *constructive interpretation* of proofs, i.e. proving a formula implies explicitly constructing a typed λ-term. This suggests a uniform framework for representing formulas, proving programs and identifying those notions [17]. Via the Curry-Howard isomorphism, a proof of a logical formula $\forall x.P(x) \Rightarrow \exists y.Q(x,y)$ (called a specification) needs to be derived in order to produce a *correct* program in such a formalism. A proof of a specification gives a method to transform an object i and a proof $P(i)$ into an object o and a proof of $Q(i,o)$. The program p, as $y = (p\ x)$, is obtained by *extraction* from the proof of the formula by forgetting everything concerning the logical correctness of the program [18].

In this way, the specification of a program can be represented by a logical formula and the program itself can be extracted from the constructive proof of the specification. In this paper, we look at the possibility of developing certified functional Bulk-Synchronous programs and we show that type theory is appropriate to specify and prove these programs. The functional approach of the BSP model allows the re-use of suitable techniques for formal proof from functional languages because a small number of parallel operators is needed for an explicit parallel extension of a functional language and it keeps the advantages of the BSP model: no deadlock, static cost formula and cost predictions. We shall show how easy it is to extend the Calculus of Inductive Construction (in our implementation, the **Coq** System) with the parallel operators of the BSλ-calculus, to make formal proofs and to have a certified library of common functional BSP programs.

This paper describes our first work in this direction. Section 2 briefly presents the **Coq** System. The presentation and the formalization of the BSML operators in the **Coq** Proof Assistant is given in Section 3. It is used for developing several

`[x:A]B`	Abstraction of B w.r.t. x of type A;
`(A B)`	Application of A to B;
`(x:A)B`	Product, or dependent type, of B w.r.t. x of type A;($\equiv \forall x{:}A.B(x)$)
`A -> B`	Type of function from `A` to `B` ($\equiv$ `(x:A)B` when `x` not in `B`);
`Prop,Set`	Type of propositions; Type of specifications or data types.

Figure 1: Formations of terms and types in **Coq**

certificated BSML programs which are part of the standard library of `BSMLlib` (Section 4). Section 5 presents a non-trivial application of this method. We end with related work (Section 6), future work and conclusion (Section 7).

2. The Coq Proof Assistant

The system **Coq** is a proof assistant, an implementation of a particular type theory called the Calculus of Construction enhanced with inductive definitions. It provides mechanisms to write definitions, statements and to make formal proofs. Following Curry-Howard isomorphism, types are seen as propositions and terms as proofs. The proof engine builds terms by means of *tactics* given by the users. The safety of the **Coq** System is a consequence of the *consistence* (confluence and strong normalization) of the logical framework and of the type checker of completed proofs. We shall briefly present the syntax and the principles of **Coq**, in order to explain how we implemented the decision procedure for functional BSP programs.

Types can be defined by their *arity*. For example, the type of natural numbers can be defined as `nat:Set` with the following inductive type:

```
Inductive nat : Set := O : nat | S : nat->nat.
```

Objects on these types can be defined. For example, the predecessor function can be defined as `pred:nat`$\rightarrow$`nat`. Then, predicates on these objects can be defined. The predicate less-or-equal, for example, can be defined as an inductive predicate of type: `nat`$\rightarrow$`nat`$\rightarrow$`Prop` which could be applied to natural numbers: `(le n (S  n)):Prop` ($n < n{+}1$). Now, predicates and objects can be mixed in specifications. For example, `(n:nat)` {`m:nat | m=n*2`} is a specification to express $\forall n \Rightarrow \exists m$ as $m = n * 2$. A proof of such a specification is inhabited by a pair consisting of a natural number and a proof that this number satisfies the specification. The program extracted from the proof of this specification is naturally the "double" function. In Figure 1, we give the different possible formations of terms and types (with the exception of inductive types; for sake of conciseness we refer to [1] for more details). All other notations are syntactic sugar for these formations and special inductive types.

When a specification has been given, we must prove it. For this, the **Coq** Proof Assistant enters a special and interactive mode where the user must prove the different goals of the specification (in the *context* in which we make the proof) with the help of tactics which apply one or more rules of the Calculus of Constructions to the current goal. Thus, we are always ensured to stay in a coherent state, and to have a sequence which can be derived from the initial one.

As we explained earlier, proofs of specifications can be developed and programs can be extracted from these proofs. Such a program is called a *realization* of the specification. The extraction allows the elimination of non-computational parts of proofs, including dependent types [18]. Some other systems such as HOL [10] or PVS [3] offer similar possibilities. However, both of them use untyped theories and the extracted terms are untyped or do not allow the termination (or provide a redundance of hidden information). **Coq** is thus a good framework for our method of building proofs from a specification and producing a certified library from the extraction of the proofs. Extracted programs are given in two programming languages: Objective Caml and Haskell. Thus, in the first language, we could use the `BSMLlib` library to compile extracted programs and the development of [16], in the second.

3. Formalization of `BSMLlib` in Coq

To represent our parallel language and to have a specification-extraction of functional BSP programs, we have chosen a classical approach: an *axiomatization* of the parallel operators. Thus, operations are given by *parameters* and do not depend on the implementation (sequential or parallel). This formalization is based on the BSλ-calculus programming model and the `BSMLlib` elements. This model has been used for testing the consistency of the axioms: in order to accept a judgment as an axiom we have first informally verified that it is satisfied by a judgment of this confluent (untyped) calculus [15]. We first give the informal description of the elements of the `BSMLlib` library and, next, its axiomatization.

3.1. The `BSMLlib` *library*

There is an abstract polymorphic type `'a par` which represents the type of p-wide parallel vectors of objects of type `'a`, one per process where `bsp_p()` is p, the static number of processes of the parallel machine. The parallel constructs of BSML operate on parallel vectors. Those parallel vectors are created by:

```
mkpar:  (int -> 'a) -> 'a par
```

so that `(mkpar f)` stores `(f i)` on process i for i between 0 and $(p-1)$. Asynchronous phases are programmed with `mkpar` and with:

```
apply:  ('a -> 'b) par -> 'a par -> 'b par
```

`apply (mkpar f) (mkpar e)` stores `(f i) (e i)` on process i. The communication and synchronization phases are expressed by:

```
put:(int->'a option) par -> (int->'a option) par
```

Consider the expression: `put(mkpar(fun i->fs`$_i$`)) (*)`. To send a value `v` from process `j` to process `i`, the function `fs`$_j$ at process `j` must be such as `(fs`$_j$ `i)` evaluates to `Some v`. To send no value from process `j` to process `i`, `(fs`$_j$ `i)` must evaluate to `None`. Expression (*) evaluates to a parallel vector containing a function `fd`$_i$ of delivered messages on every process. At process `i`, `(fd`$_i$ `j)` evaluates to `None` if process `j` sent no message to process `i` or evaluates to `Some v` if process `j` sent the value `v` to the process `i`. The full language would also contain a synchronous global conditional operation:

```
ifat:  (bool par) * int * 'a * 'a -> 'a
```

such that `ifat (v,i,v1,v2)` will evaluate to `v1` or `v2` depending on the value of `v` at process `i`. Without this, the global control cannot take into account locally computed data. But this synchronous conditional operation cannot be defined as a function. That is why the core `BSMLlib` contains the function: `at:bool par -> int -> bool` to be used only in the construction: `if (at vec pid) then... else...` . `if at` expresses communication and synchronization phases.

*3.2. Axiomatization in **Coq***

The process number, `bsp_p()`, is naturally an integer given by the parameters. An axiom indicates that it is greater than 0. Parallel vectors are indexed over type Z (**Coq**'s integer), starting from 0 to the constant process number (in **Coq**, `tt` correspond to the ML constant `()`). They are represented in the logical world by an abstract dependent type `Vector T` where `T` is the type of its elements. This abstract type is manipulated only by one of the parameters. All the axiomatization of the parallel operators is given in Figure 2 (we give for exemple, the `mkpar` parameter). `at` is an abstract "access" function for the parallel vectors. It gives the local value contained in a process and it would be used in the specification of programs to give the values contained in the parallel vector result. It has a dependent type to verify that the number `i` is a valid process number.

The asynchronous operators: the constructor `mkpar` of parallel vectors and the global application are axiomatized with `mkpar_def` and `apply_def`. For a function `f`, `mkpar_def` stores `(f i)` on the process `i` by using the access function `at` and in the same manner `apply_def` stores `(V1 i) (V2 i)`. The parallel vector result is described with an equality which has the type `Prop`. The result is given for a parameter `i` which must be proved to be a valid process number. The communication of values, the `put` operator, is axiomatized with `put_def`. It transforms a functional vector to another functional vector which guides communication using the functional parameter `j` to read values from distant processes. The parameter `j` is tested with the function `within_bound` of type:

```
(i:Z) {'0<=i<(bsp_p tt)'}+{~'0<=i<(bsp_p tt)'}
```

which tells whether an integer is a valid process number or not and which gives the proof. If it is valid, the value on the process `i` is read on process `j` with the access function (`j` is a valid process number; the proof is given by `within_bound`). Otherwise, an empty constant is returned. In a real implementation, the values to emit are first calculated and then exchanged with an optimal algorithm.

The full axiomatization would also contain the synchronous conditional. Its axiomatization is easy to express using the `Bool` libraries of the **Coq** system. `sumbool_of_bool` is a function that transforms a boolean to the proof that is true or not. For proofs of programs, it is easier to directly manipulate proofs than primitive constants. The parameter `n` needs to be a valid process number and this proof is given to the "access" function by a dependent type. Since `ifat` is a function, in this axiomatization, expressions using a global conditional would also be extracted by the **Coq** System as a function. Thus, the extracted code needs to be parsed in order to be transformed into a suitable conditional.

We have also axiomatized an extension of the `BSMLlib` library, the parallel superposition [14] (`super`) which allows us to have in a pure functional setting a constructor that can be used to run two "BSP threads" and to express directly parallel divide-and-conquer algorithms. Informally it means that a term (whose evaluation

```
Parameters  bsp_p: unit -> Z;  Vector: Set -> Set;
            mkpar: (T:Set) (Z -> T) -> (Vector T).

Axiom good_bsp_p:‘0 < (bsp_p tt)‘.
Axiom at: (T:Set) (Vector T) -> (i:Z) ‘0<=i<(bsp_p tt)‘ -> T.

Axiom mkpar_def: (T:Set) (f:Z -> T)(i:Z) (H:‘0 <= i < (bsp_p tt)‘)
                    (at T (mkpar T f) i H)=(f i).

Axiom apply_def: (T1,T2:Set) (V1:(Vector (T1 -> T2)))
                 (V2:(Vector T1)) (i:Z) (H:‘0 <= i < (bsp_p tt)‘)
(at T2 (apply T1 T2 V1 V2)i H)=((at (T1->T2) V1 i H)(at T1 V2 i H)).

Axiom put_def: (T:Set)(Vf:(Vector (Z -> (option T))))
               (i:Z) (H:‘0 <= i < (bsp_p tt)‘)
((at (Z->(option T)) (put T Vf) i H)=([j:Z]
   if (within_bound j) then [H1]((at (Z-> (option T)) Vf j H1) i)
                       else [H2](None T))).

Axiom ifat_def: (T:Set)(V:(Vector bool)) (n:Z)(R_IF,R_ELSE:T)
(Hyp_n: ‘0 <= n < (bsp_p tt)‘) ((ifat T V n R_IF R_ELSE)=
        (if (sumbool_of_bool (at bool V n Hyp_n)) then [H1] R_IF
           else [H2] R_ELSE)).

Axiom super_def: (T1,T2:Set) (v1: unit -> (Vector T1))
  (v2: unit -> (Vector T2)) (super T1 T2 v1 v2)=((v1 tt),(v2 tt)).
```

Figure 2: Axiomatization of the BSML language

has been first “frozen”) will be reduced concurrently to another one (also “freeze”) and the result is the pair of the above two thread evaluations. It is a first step towards the use of BSML for grid computing. Note that we define all our operators with equalities. For the proofs of programs, this enables a proof of a specification to recognize what the value contained on a process is. We will illustrate this and use this set of axioms, to develop certified functional BSP programs and to verify their formal properties.

4. Certification of BSML programs

We present here the proof of some parallel algorithms in the **Coq** system. In order to simplify the presentation and to ease the formal reasoning, we limit our study and formal proofs to the expressions that are very common in a BSP algorithm. Some of them have been used in [15], to prove the capacity of the equational theory of the BSλ-calculus. But without a proof assistant, the authors have made some mistakes in their proofs. Those case studies have demonstrated the relevance of the use of the **Coq** System in the proof of parallel programs *correctness*. In particular, we used defined predicates and the **Coq** libraries several times in the developments. We also used higher lemmas to define new principles and to prove correctness of programs [17] (in our case, BSML programs). The formal developments described in this section are freely available [2].

4.1. Replicate

The first case study we present is one of the most simple functional BSP expressions: the *replication* of the same value on each process. It is often used at the beginning of BSP algorithms to replicate all the parameters. As we explained before, the specification of `replicate` could be read as the following formula:

$$\forall \text{Data}\ \forall a : \text{Data} \Rightarrow \exists res \text{ as } \forall i\ (0 \leq i < p)\ (\text{res}[i] = a)$$

where $res[i]$ is an abbreviation for the function access of local data (we do not write the type of an element when it is intuitive). To prove this trivial specification, we need only apply the `mkpar_def` axiom and the extracted program will fit the desired one: `let replicate a = mkpar (fun x -> a).`

4.2. Broadcast

The second case is a classical example: the *broadcast* of an element held by a process *root*. *Exchange* of values is the critical point of parallel algorithms and needs to be certified. To limit its use and to avoid the failure of the program, *root* needs to be a valid process number. If not, we model this exception by returning the `False` proofs. Thus, the fully specified function should have the following type:

$$\forall \text{Data}\ \forall root\ (0 \leq root < p)\ \forall vv \Rightarrow \exists res \text{ as } \forall i(\ 0 \leq i < p)\ (\text{res}[i] = \text{vv}[root]).$$

To prove this specification, we can use the direct algorithms of broadcasting (see [2] for the realization). For *Data* that is a `list`, we can use the classical two-phase algorithm. The certification of this algorithm needs the `scatter` certified function (see next) for the first phase, the `totex` function (which makes a total exchange of values) for the second and a function for the partition of the list (for the sake of conciseness, we refer to [2] for the proof of the specification). The only difficulty comes from the development of the partition function since properties of the division are not automatically solved in **Coq** and technical lemmas are required.

4.3. Scatter a parallel vector

The `scatter` function is a useful program of exchange of values. For example, as we saw before, it is used for the development of the two-phase broadcasting algorithm. It uses a `partition` function for the data of the parallel vector. Informally, (`scatter partition root` $\langle x_0, \ldots, x_{p-1} \rangle$) scatters the value $x_{\texttt{from}}$. For each number of process i, if (`partition` $x_{\texttt{root}}$ i)$=y_i$ then this value is held at processor i. The formal specification of this function suggested by this description is:

$$\begin{gathered}\forall \text{Data1,Data2}\ \forall partition\ \forall root\ (0 \leq root < p)\ \forall vect \Rightarrow \exists res \text{ as} \\ \forall i(\ 0 \leq i < p)\ (Some\ \text{res}[i]) = (partition\ \text{vect}[root]\ i)\end{gathered}$$

The proof of this specification has be made by case on the value *root* and the equality between *root* and the number of the process. With the dependent type of **Coq**, we can easily prove that the "access" function used a valid process number.

```
Inductive  k_first_R  [T:Set][R:T->T->T] [v:(Vector T)]
                : (k:Z) `0<=k<(bsp_p tt)` -> (res:T) Prop:=
   k_zero: (k_first_R T R v `0` H_0 (at T v `0` H_0))

 | k_rec : (k:Z) (H_r:`0<=k<(bsp_p tt)`)  (a:T) (H_sub:`k>0`)
                 (k_first_R T R v `k-1` (p_sub k H_r H_sub)  a)
              -> (k_first_R T R v `k` H_r (R a (at T v `k` H_r))).
```

Figure 3: Inductive type used in the specification of the parallel `scan` programs

4.4. Gather a parallel vector

The opposite of scattering a parallel vector is naturally to gather it. The specification and the development of this function is more complex than that of the `scatter` function. (`gather_list` *root* $\langle x_0, \ldots, x_{p-1}\rangle$) evaluates to a parallel vector whose value at processor *root* is the list $[x_0; \ldots; x_{p-1}]$. For this, we used an intermediate function, `gather`, which gives a parallel vector of functions where the function at process *root* gives the values from the process *dest*, if it is a valid process number. Otherwise, it returns `None`. Or, the function at process i returns `None` for each value of *dest*. Now, at processor *root*, we must apply this function to a list containing the process number, or else to an empty list. Thus, we have our desired parallel vector. The specification of `gather_list` is:

$$\forall \text{Data}\ \forall root\ (0 \leq root < p)\ \forall vect \Rightarrow \exists res \text{ as}$$
$$\exists l{:}\,list \text{ as } l = res[root] \text{ and } H{:}\,(length\ l) = p \text{ and } \forall j\ (0 \leq j < p)\ (nth\ l\ H\ j) = vv[j]$$
$$\text{and } \forall i(\ 0 \leq i < p)\ (i \neq root)\ res[i] = nil$$

where `nth` is the function of the nth value of the list l (using the proof that $j \leq (length\ l)$). This specification could be proved using the property of `gather`, by case on $i = root$ and using a technical lemma which gives the property that the length of the list of processes is p.

4.5. A certified `scan`

The `scan` function is also a classical parallel algorithm. It uses an operator R and (`scan` R $\langle x_0, \ldots, x_{p-1}\rangle$) evaluates to $\langle s_0, \ldots, s_{p-1}\rangle$ where $s_i = R_{k \leq i} x_k$. The specification of this function is:

$$\forall \text{Data}\ \forall vect\ \forall R : Data \rightarrow Data \rightarrow Data \Rightarrow$$
$$\exists res \text{ as } \forall i\ H : (0 \leq i < p)\ (k\_first\_R\ T\ R\ res\ i\ H\ res[i])$$

using the inductive type describe in Figure 3 which gives the application of the operator R to the first k values of the parallel vector (`p_sub` is the a technical lemma which transforms a proof of $0 \leq k < p$ and $k > 0$ to $0 \leq k-1 < p$). Because it is a logical inductive, it does not appear in the program at the extraction time. It is used only to verify formal properties. So we can use the "access" function without problem of nested parallelism. There are different proofs (with its computational part) of this specification: direct algorithm, divide-and-conquer algorithm and one with a logarithmic number of super-steps. The direct algorithm is naturally the most simple to prove. For the other two, we used a "well-founded relation" [1] on the number which increases from 0 to p and for the divide-and-conquer algorithm, we used the `super` operator [2] [14].

All the above parallel algorithms (and some others such as the parallel shift, `fold_left` or the proof of the implementation of the dual `put` operator: get [15] [2]) are the main elements of the `BSMLlib` library and are the basis for more complex algorithms. Now, we present how a certified parallel sort could be developed using these basic programs and our axiomatization.

5. Development of a parallel sorting algorithm

Sorting is a classical problem of parallel algorithms which is complex and covers a huge range of program constructions. It is not so easy to write a correct implementation of parallel sorting algorithms, even without any optimization, since a small mistake in such complex algorithms immediately has some catastrophic consequences. To validate our study, we give here our final example, the *sampling sort algorithm* (PSRS) of Schaeffer in its BSP version [22]. The PSRS algorithm proceeds as follows. First, the lists of the parallel vector (we assume that their lengths are $\geq p^3$) are sorted independently with a sequential sort algorithm. The problem now consists of merging the p sorted list. Each process selects from its list $p+1$ elements for the primary sample and there is a total exchange of these values. In the second super-step, each process reads the $p \times (p+1)$ primary samples, sorts them and selects p secondary samples. In the third super-step, a processor picks a secondary block, collects its elements, merges the primary blocks, and discards the values that do not belong to the assigned secondary block. In order to do this, a processor, at the second super-step, sends all primary blocks that may intersect with the assigned primary blocks.

As in a sequential sort, the PSRS algorithm needs a *relation* (`inf`) which is decidable, transitive and anti-symetric. Its also needs a sequential sort and the "merge" of two lists (`treesort` and `merge` which could be done using **Coq** [1] [17]). When specifying functional parallel sorting programs, one must express two facts: first, obviously, the lists of the parallel vector are *sorted* after the evaluation of the programs, but, also that the values contained in the initial parallel vector are *preserved* by the programs, which possibly permute them but do not modify any of them. This expresses a list by `sorted` and `permutation` as described in [1]. Thus, we must also express it for our parallel vectors (Figure 4): `PSRS_sorted` (`inf_list` is a logical predicate of inferiority: each element of the first list is inferior to the elements of the second list) which could be read as "every list of the vector is sorted and the maximum of the ith list is inferior to the minimum of the i+1th" (trivial property of `inf_list`) and, `PSRS_exchange` which express that every list of elements in the first parallel vector appears in the second (`list_in_list` is a logical predicate of inclusion). Now, we can define `PSRS_permut` as the smallest equivalence relation which contains transpositions, i.e. exchange of elements. In the second super-step, the algorithm does not change the vector of lists and so the reflexivity of the permutation is needed. In the same manner, the transitivity keeps this logical predicate for the sequence of super-steps.

```
Inductive PSRS_sorted [T:Set; v:(Vector (list T))] : Prop :=
 PSRS_sor: (i:Z)(H:`0<=i<(bsp_p tt)`)((sorted T inf (at ? v i H)) /\
           ((H':`i<>(bsp_p tt)-1`) (inf_list T inf (at T v i H)
                          (at T v `i+1` H''))))) -> (PSRS_sorted T v).

Inductive PSRS_exchange [T:Set; v1,v2:(Vector (list T))] : Prop :=
    PSRS_exch : (i:Z) (H_i:`0<=i<(bsp_p tt)`) (l:(list T))
(list_in_list T l (at T v1 i H_i))->(Ex [j:Z](H_j:`0<=j<(bsp_p tt)`)
     (list_in_list T l (at T v1 j H_j))) -> (PSRS_exchange T v1 v2).

Inductive PSRS_permut: (Vector (list T))->(Vector (list T))->Prop :=
 PSRS_exch_is_permut : (v,v':(Vector(list T)))(PSRS_exchange T v v')
                             ->(PSRS_permut T v v')
| PSRS_permut_refl :  (v:(Vector (list T)))(PSRS_permut T v v)
| PSRS_permut_sym : (v,v':(Vector (list T)))(PSRS_permut T v v')
                        -> (PSRS_permut T v' v)
| PSRS_permut_trans: (v,v',v'':(Vector(list T)))(PSRS_permut T v v')
        -> (PSRS_permut T v' v'') -> (PSRS_permut T v v'').
```

Figure 4: Inductive specification of the parallel sampling sort

Thus, the specification of a parallel sort is naturally the existence of a parallel vector of lists which is sorted and which is a permutation of the initial one. To certify the PSRS algorithm, we must first develop a certified sampling function (it is the main difficulty). Then we must prove technical lemmas which say that the first super-step keeps the PSRS_permut predicate, that the third super-step keeps the PSRS_permut and permits the PSRS_sorted predicate. Those demonstrations are made with the use of the above property of our certified programs and by case on the length of the exchanged lists. Using the arithmetic operators of the **Coq** librairies, the extracted program is slower than an handmade one. But the parallel parts are the same.

6. Related work

The first formal semantics of BSP was an axiomatic logic and parallel explicit semantics of the BSP model for a shared-memory programming implementation [12]. Programs are given together with their formal specifications where each step is fully justified by mathematical reasoning. They are based on mapping each process to a trace sequence which may be combined to determine composite behavior. Unfortunately, no program has been certified with this set of algebraic laws and no implementation of the mathematical model of the specifications has been made. But the idea suggested the possibility of proving parallel algorithms with the BSP models while preserving the cost model. In another approach [21], the reasoning is made by using a sequence of global (parallel) state transformations. It makes the reasoning easy because whole parallel operations can be described as global state transformations. This paper shows the refinement of a sequential version of the

Floyd's shortest path algorithm to a BSP version. [20] presents an extention of the Refinement Calculus (weakest precondition calculus) to permit the derivation of programs in the BSP style. All those approaches are based on imperative languages with handmade proofs mechanically unassisted.

Our approach has the following advantages. It is based on a functional language so it eases the reasoning. Using the **Coq** Proof Assistant our proofs are partially automated which is not the case in other approaches. Moreover we can generate programs using the **Coq** System and benefit from prior developments which have been made in **Coq** (the users' contributions [1]).

7. Conclusion and Future Work

This article has demonstrated the adequacy of the **Coq** Proof Assistant (and the type theory) in the process of certifying parallel programs. We have formalized the parallel operations of the BSML language and, using this formalization, we have developed certified often used BSML programs which constitute an important subset of the `BSMLlib` library. This allows us to validate these programs and even to find mistakes in the hand-written proofs. Our axioms are informally related to the BSλ-calculus. A model which realizes our set axioms seems not easy to find since our axioms interact with λ-calculus but this will be a future work.

Several directions can be followed for future work: validation of more complex BSP programs for scientific computing [5] [22] etc. Our goal is to have a certified `BSMLlib` library (including its standard library), to make an extension of this framework including imperative features (as in [12]), using work done on sequential programs [7] and to have a software for certification of BSML programs (as in [4]) as well as to study the possibility of proving cost formulas.

We also continue our work on the certification of the environment. In particular, the parallel abstract machine [8] must be proved correct with respect to a version of the BSλ-calculus with explicit substitutions in order to follow the methodology presented in [11]. Thus, the certified `BSMLlib` library could be used together with this abstract machine and our type system for programming functional BSP algorithms in a safe and certified environment.

Future work will also consider extensions of the BSML framework with parallel juxtaposition [13] which allows us to divide the networks into subnetworks but still follows the "flat" BSP model which forbids subset synchronisation. This new construction allows to write parallel divide-and-conquer algorithms and thus is an important step towards the use of BSML for grid computing. This extention of our framework could be also axiomatized (the number of process would be directly expressed in specification not as a constant, but as a propositional parameter). Therefore, future "grid algorithms" could be developed and automatically generated in a certified way.

Acknowledgements

This work is supported by a grant from the French Ministry of Research and the ACI Grid program, under the project CARAML. The author wish to thank the anonymous referees, Frédéric Loulergue and Mitzu Modard for their comments.

References

[1] *The Coq Proof Assistant.* http://coq.inria.fr/, 2003.

[2] *Formal Proofs of BSML Programs.* http://www.univ-paris12.fr/lacl/gava/, 2003.

[3] *The PVS Specification and Verification System.* http://pvs.csl.sri.com, 2003.

[4] *Why: a software verification tool.* http://why.lri.fr/, 2003.

[5] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94, Vol. I*, volume 51, pages 509–514. Elsevier, 1994.

[6] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 37(2-3), 1988.

[7] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4), 2003.

[8] F. Gava and F. Loulergue. A Parallel Virtual Machine for BSML. In Peter M. A. Sloot and al., editors, *ICCS 2003*, number 2657 LNCS, pages 155–164. Springer, 2003.

[9] F. Gava and F. Loulergue. A Polymorphic Type System for BSML. In *PaCT 2003*, LNCS. Springer, 2003. to appear.

[10] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[11] T. Hardin, L. Maranget, and L. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.

[12] He Jifeng, Quentin Miller, and Lei Chen. Algebraic Laws for BSP Programming. In L. Bouge and Y. Robert, editors, *Euro-Par'96*, number 1124 in LNCS, pages 359–368, Springer, 1996.

[13] F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. In Harald Kosch.et. al, editor, *Euro-Par 2003*, LNCS. Springer, 2003.

[14] F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *ICCS 2003*, number 2659 in LNCS, pages 223–232. Springer, 2003.

[15] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.

[16] Quentin Miller. BSP in a Lazy Functional Context. In S. Gilmore, editor, *Trends in Functional Programming, Volume 3*, pages 1–13. Intellect Books, 2002.

[17] C. Parent. Developing certified programs in the system coq: The program tactic. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, pages 291–312. Springer, Berlin, Heidelberg, 1993.

[18] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[19] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3), 1997.

[20] D.B. Skillicorn. Building BSP Progams Using the Refinement Calculus . In *Formal Methods for Parallel Programming and Applications workshop at IPPS/SPDP'98*, 1998.

[21] A. Stewart, M. Clint, and J. Gabarro. Algebraic Rules for Reasoning about BSP Programs. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming.* Nova Science Publishers, august 2002.

[22] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms.* PhD thesis, Oxford University Computing Laboratory, 1998.

[23] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.