Calculating Parallel Programs in Coq using List Homomorphisms

Frédéric Loulergue, Wadoud Bousdira, Julien Tesson

Calculating Parallel Programs in Coq using List Homomorphisms

Frédéric Loulergue · Wadoud Bousdira · Julien Tesson

Abstract SyDPaCC is a set of libraries for the Coq proof assistant. It allows to write naive functional programs (i.e. with high complexity) that are considered as specifications, and to transform them into more efficient versions. These more efficient versions can then be automatically parallelised before being extracted from Coq into source code for the functional language OCaml together with calls to the Bulk Synchronous Parallel ML (BSML) library. In this paper we present a new core version of SyDPaCC for the development of parallel programs correct-by-construction using the theory of list homomorphisms and algorithmic skeletons implemented and verified in Coq. The framework is illustrated on the maximum prefix sum problem.

Keywords Parallel programming, algorithmic skeletons, constructive algorithms, proof assistant

1 Introduction

Nowadays parallel architectures are everywhere, but not parallel programmers. High-level programming abstractions and methods are needed, in particular for distributed memory models. Our goal is to provide a framework to *ease* the *systematic* development of *correct* parallel programs.

In the Bird Meertens Formalism (BMF) [2], an efficient program is obtained from a naive functional program considered as a specification, through program transformations. BMF can be applied to parallel programming (e.g. [5]). In

Frédéric Loulergue
Inria πr^2, PPS, Univ Paris Diderot, CNRS, Paris, France.

Frédéric Loulergue · Wadoud Bousdira
Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, France. E-mail: Firstname.Lastname@univ-orleans.fr

Julien Tesson
Université Paris Est, LACL, UPEC, F-94010, Créteil, France. E-mail: Julien.Tesson@lacl.fr

these approaches, however, the transformations are pen-and-paper proofs and the transition from an efficient functional expression to a parallel program (often in an imperative language, for example a library of algorithmic skeletons hosted in C++) is not grounded on a formal basis.

We develop the framework SyDPaCC [14,7] for the Coq proof assistant [22] to ease the use of methods based on program transformation and algorithmic skeletons and make them more reliable. More specifically the contributions of this paper are:

- the support of list homomorphisms in SyDPaCC that previously only supported BSP homomorphisms [14] and the GTA paradigm [7],
- improvements of the formalisation of the parallel functional language Bulk Synchronous Parallel ML (BSML) [13] and algorithmic skeletons in Coq,
- an application of the framework to produce a parallel program for the maximum prefix sum problem.

We first give an overview of the features of Coq. The paper does not assume any prior knowledge of Coq, but familiarity with functional programming is necessary. Then we present how a function written as a composition of sequential map and reduce could be automatically parallelised in SyDPaCC based on a formalisation of BSML (Section 3) and verified parallel versions of map and reduce (Section 4). In Section 5 we present the modelling in Coq of two important list homomorphism theorems that are used to prove that a function having three simple properties can be expressed as a composition of sequential map and reduce and therefore be parallelised. The framework is exemplified on the maximum prefix sum problem in Section 6. Comparison with related work (Section 7) and conclusion (Section 8) close the paper.

2 An Overview of the Coq Proof Assistant

Coq [22] is an interactive theorem prover. It is based on the Curry-Howard correspondence relating terms of a typed λ-calculus (the calculus of (co)-inductive constructions) with proof trees of a logical system in natural deduction form. From a more practical side, Coq can be seen as a functional programming language, close to OCaml or Haskell but with a richer type system that allows to express logical properties. There are many cases where programs are developed in Coq and their properties are also proved in Coq. This is for example the case of the CompCert compiler, a compiler for the C language, implemented and certified using Coq [12]. Our SyDPaCC system also uses this style.

In Coq, data structures are defined only by induction. For example the list data structure is defined in Coq, Haskell and OCaml in Figure 1. In all three cases, a list is built using the constructor for empty list (Nil), or the constructor that adds an element at the beginning of a given list (Cons). In Haskell and OCaml, a and α respectively are type variables. In the Coq version, A is simply a variable and we indicate this variable is a **Type**.

In Coq, values and types can be mixed together, and types may depend on values. For example the definition for lists of a given size begins with:

```
(* Coq *)
Set Implicit Arguments.
Inductive list (A:Type) : Type :=
  | Nil: list A
  | Cons: A→list A→list A.
Fixpoint map A B (f:A→B) xs :=
  match xs with
  | Nil      ⇒ Nil B
  | Cons x xs ⇒ Cons(f x)(map f xs)
  end.
```

```
-- Haskell
data List a = Nil | Cons a (List a)
map f Nil         = Nil
map f (Cons x xs) = Cons (f x) (map f xs)

(* OCaml *)
type α list = Nil | Cons of α * α list
let rec map f = function
  | Nil → Nil
  | Cons(x,xs) → Cons(f x,map f xs)
```

Fig. 1 Lists in Coq, Haskell & OCaml

```
Inductive t (A:Type) : nat→Type := nil: t A 0 | (*...*)
```

In this definition A is a parameter: all the applications of the inductive definition t should be applied only to A in the body of the definition of t. t has a second *argument*: a natural number indicating the size of the vector. As in the declaration of t different values are passed for this argument, it cannot be a parameter, hence its position after the : symbol. In summary in new type declarations in Coq, parameters/arguments of the type could be both types and values.

Figure 1 also shows the definitions of the map function in the three considered languages. All the definitions are recursive and by pattern matching on the list argument. They are quite similar but in their arguments: in Coq there are two arguments A and B that are *types*. Note that we could write (A B:**Type**) instead, but we prefer to let Coq infer the type. We also let Coq infer that the type of xs is list A. Considering the type arguments as regular arguments is very expressive. However, it may be verbose to explicitly apply a polymorphic function to the types it is parametric in. The **Set Implicit Arguments** command tells Coq to make this kind of arguments implicit (and inferred by Coq). In the recursive call of map these arguments are actually omitted and the code looks very similar to the Haskell and OCaml code.

If we request the various systems to give the type of map, Coq returns the type ∀ A B:**Type**,(A→B)→list A→list B whereas OCaml returns the type $(\alpha\to\beta)\to\alpha$ list$\to\beta$ list and Haskell (t→a)→List t→List a. The ∀(x:A), B construction is a primitive one, it is called the depend product. This type is the type of the expression fun(x:A)⇒e where e has type B considering x has type A. A→B is syntactic sugar when B does not contain free occurrences of x.

It is also possible to define usual OCaml (or Haskell) notations for lists in Coq. In the remaining of the paper we will use [] for the empty list, and :: in infix notation for Cons.

Dependent product, function abstraction, inductive definitions and (dependent) pattern matching are the key constructions of Coq. Logical elements can be built using them. We refer to [22] for the details, and just show an example of a lemma statement[1] from the Coq standard library about lists:

[1] In Coq all the following commands are synonym: Fact, Lemma, Proposition, Theorem.

```
Lemma in_inv : ∀ A (a b:A) (l:list A), In b (a :: l) → a = b ∨  In b l.
```

In this statement, → may be thought as logical implication rather than functional arrow (actually these notions are in correspondence by the Curry-Howard correspondence). The predicate In could be defined as:

```
Inductive In A (x:A): list A→Prop := (* Prop is the kind of ''logical'' types *)
  | Inhead: ∀ (xs:list A), In x (x::xs)
  | Intail: ∀ (y:A) (xs:list A), In x xs→In x (y::xs).
```

In the Curry-Howard correspondence, a type and a logical statement are the same and a program and a proof are the same. The notation in_inv : *(*...*)* indicates that the name in_inv has what follows the colon as type. For the definition of a function or a value, the body of the definition (for example for function map) follows after symbol :=; it is actually a syntax friendly λ-term of the calculus of inductive constructions. If we follow the Curry-Howard correspondence, the proof of the lemma in_inv could be also written as a λ-term/program. However, it is not convenient to do so and Coq provides a language of tactics to write scripts that build λ-terms that are proofs. For in_inv one possible script is:

```
Proof. intros A a b l H. inversion H; subst; [left | right]; trivial. Qed.
```

We will not explain what exactly this script means, but only emphasise that Coq enters an interactive proof mode that helps the user to write the script and that just before **Qed** a λ-term is built. The command **Qed** then checks that this term has actually the type given as the statement of lemma in_inv. Only after this check is done, the lemma is added to Coq definitions. This means the only part of Coq that should be trusted is the type checking part: the kernel. Buggy tactics could build incorrect proofs but these proofs would be rejected by the kernel.

One additional strength of Coq is that the functional programming realm and the logical realm could be mixed together. The simplest example of doing so is the dependent pair:

```
Inductive sig (A:Type)(P:A→Prop): Type := exist : ∀ x : A, P x → sig P.
```

For a predicate P over value of type A, a value of type[2] sig A P is a pair composed of: a *value* a of type A together with a *proof* that P a holds. There exists a Coq notation for sig P A which is quite intuitive: {x:A | P x}. In particular, this type could be used to express pre- and post-conditions of functions in Coq. For example, the head of a list is only defined on non empty lists, but only total and terminating functions can be defined in Coq. Therefore one can use a pre-condition and write:

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

[2] actually A is implicit but making it explicit makes the explanation clearer

Here we use the **Program** feature of Coq that allows to consider that xs is of type list A in the definition of the body of the function and to omit some parts of the body using the _ symbol on the right of ⇒ for the empty list case. **Program** then generates a proof obligation, for each hole in the body. In this case we do not have to write a proof script for the proof obligation because Coq proves it by itself. This proof is a proof by contradiction: l is both empty (from pattern matching) and non-empty (pre-condition).

Coq has a module system similar to OCaml's. A module is a collection of inductive definitions, value and function definitions, theorems and so on. A module type is a collection of the same items that can be found in a module, plus more abstract elements: "parameters" that are names together with types but without a body. For a value or a function this corresponds to the usual notion of type signature. For a theorem, it means a theorem without a proof, in other words an axiom. The general syntax for parameters is **Parameter** name : type and a synonym is **Axiom**. An example of module type is given in Figure 4.

A module realises a module type if it provides the bodies of all the parameters of the module type. A module can be parametric, i.e it can depend on the implementation of one or several other modules. For example a module implementing a data-structure of sets using trees could depend on a module defining the type and ordering of the elements. The definition of such a module would look like **Module Set** (E : OrderedType). *(*...*)* **End Set**. where OrderedType is a module type.

As a motivation for introducing the last feature of Coq we need, let us consider join-lists. Join-lists and list homomorphisms suit very well to the divide-and-conquer paradigm and are therefore very interesting as formal basis for a parallel program development system. The SYDPACC framework uses this kind of functions as a foundation.

A join-list is a finite sequence of values of the same type. It can be: the empty list [], a singleton $[a]$ (for some element a) or a concatenation $xs +\!\!+ ys$ of two lists xs and ys. The concatenation is associative and [] is its identity element. map and reduce can be defined on join-lists as:

$$
\begin{array}{lll}
map\ f\ [\,] & = [\,] & \qquad reduce\ \oplus\ [\,] = id_\oplus \\
map\ f\ [x] & = [f\ x] & \qquad reduce\ \oplus\ [x] = [x] \\
map\ f\ (xs +\!\!+ ys) = & & \qquad reduce\ \oplus\ (xs +\!\!+ ys) = \\
\quad (map\ f\ xs) +\!\!+ (map\ f\ ys) & & \qquad (reduce\ \oplus\ xs) \oplus (reduce\ \oplus\ ys)
\end{array}
$$

where $\oplus$ is an associative operator with identity element $id_\oplus$.

Of course in Coq join-list cannot be defined as is because this definition has logical properties on the constructors; However, join-lists are isomorphic to cons-lists as defined in Figure 1. Function *map* corresponds to function map in Coq. For *reduce* there are two difficulties: first we assume that $\oplus$ and $i_\oplus$ form a monoid, and second reduce is defined, as it is traditional in the Bird Meertens Formalism, as a binary operation taking $\oplus$ and a list. The first difficulty can be simply overcome: these logical properties could be added as a pre-condition to an argument containing both $\oplus$ and $i_\oplus$:

```
Fixpoint reduce A (m:{(op,e):(A→A→A)∗A|Monoid A op e})(l:list A):=(∗...∗)
```

As A would be implicit, reduce would be a binary operation where the m argument is a complex value containing the binary operator, its identity element, and the fact that they form a monoid with A. A better solution would be:

```
Fixpoint reduce A (op:A→A→A) (H:Monoid A op e) (l:list A) : A := (∗...∗)
```

in such a way that H is made implicit: the identity element of op and the fact they form a monoid with A should come from some information of the context where reduce is used.

Monoid could be defined as a conjunction of properties about op and e. A convenient way to do so is to use a record that is in fact syntactic sugar for an inductive definition plus the dot notation to access the fields:

```
Record Monoid A (op:A→A→A) (e:A) : Prop :=
 { left_unit: ∀ a:A, op e a = a;
   right_unit: ∀ a:A, op a e = a;
   associative: ∀ a b c : A, op a (op b c) = op (op a b) c }.
```

Note that all the fields are logical statements. Therefore we use the proof mode to define a value of type Monoid:

```
Definition monoid_plus_O : Monoid plus O. Proof. constructor; auto. Defined.
```

It is possible to use the notation {name:type} instead of (name:type) in a definition to make an argument implicit. However even if we do that in the proposed signature of reduce, Coq will have no way to infer this implicit argument. The necessary Coq feature are type classes. A class is similar to a record but values of these classes are defined by the keyword **Instance** that defines them and *stores* them into a database that is queried by the Coq inference mechanism when it encounters an implicit argument whose type is a class.

The class Monoid only differs from the record Monoid in using the command **Class** instead of **Record**. Then we can define instances as follows:

```
Instance m_plus_O : Monoid plus O. Proof. (∗...∗). Defined.
Instance m_app_nil A: Monoid (@app A) []. (∗ @ makes all arguments explicit∗)
```

where app is the concatenation of lists as defined in the Coq standard library.

One possible definition for reduce is then:

```
Unset Implicit Arguments. (∗ We prefer to control the implicit status ∗)
Fixpoint reduce {A} (op:A→A→A) {e} {H:Monoid op e} (l:list A) : A :=
  match l with | [] ⇒ e | x::xs ⇒ op x (reduce op xs) end.
```

It is possible to compute in Coq: **Eval** compute **in** reduce plus [1;2;3]. Coq answers = 6 : nat to this last command. First note that reduce is used as a binary operation and that its first argument is also a binary operator (here the addition on natural numbers). To infer the implicit arguments, Coq internals detect that the argument name H has as type a type class Monoid. Therefore it looks into its database of instances: it finds the last defined instance which has plus as an operator. In our setting it is the instance m_plus_O. This instance provides e and therefore all the implicit arguments are inferred.

$$
\begin{array}{ll}
\textbf{mkpar}\ f & = \langle\, f\ 0\ ,\ldots,\ f\ (p-1)\,\rangle \\
\textbf{apply}\ \langle\, f_0\ ,\ldots,\ f_{p-1}\,\rangle\ \langle\, v_0\ ,\ldots,\ v_{p-1}\,\rangle & = \langle\, f_0\ v_0\ ,\ldots,\ f_{p-1}\ v_{p-1}\,\rangle \\
\textbf{proj}\ \langle\, v_0\ ,\ldots,\ v_{p-1}\,\rangle & = \lambda i \rightarrow v_i \\
\textbf{put}\ \langle\, f_0\ ,\ldots,\ f_{p-1}\,\rangle & = \langle\, \lambda j \rightarrow f_j\ 0\ ,\ldots,\ \lambda j \rightarrow f_j\ (p-1)\,\rangle
\end{array}
$$

Fig. 2 BSML Primitives – Informal Semantics

3 Bulk Synchronous Parallel ML in Coq

An Overview of BSML. Bulk Synchronous Parallel ML or BSML is a functional programming language currently implemented as a library for the functional programming language OCaml. The BSML library [13] provides two different implementations of the BSML primitives, a sequential implementation and a parallel implementation on top of MPI, as well as a standard library of parallel functions. Compared to a full language, the library does not provide the specific type system useful to ensure some properties about BSML programs [10]. We present here the classic syntax of BSML. A revised, more friendly syntax exists, but as programming in this style is based on OCaml pre-processing, it is not possible to have it in Coq.

BSML is based on the BSP model [23] and there is a cost model for each primitive, however for the sake of conciseness, we omit the costs in this presentation, as our framework currently does not use them. A BSP computer is a set of processor and memory pairs interconnected through a network together with a global synchronisation unit. A BSP program is a sequence of super-steps, each one being composed of: a pure computation phase where processors compute using only the data they have in local memory, a communication phase where processors exchange data, and a synchronisation barrier that ensures that all data exchanges are completed during the super-step.

BSML offers a global view of programs: a parallel program is structured as a usual sequential program but operates on a parallel data structure through dedicated primitives. This parallel data structure is called "parallel vector". A parallel vector is composed of p values, one per processor. p is the number of processes of the underlying BSP computer. p does not change during the execution of a program. From the typing point-of-view, a parallel vector has type α par: all the processors have values of the same type. Nesting of parallel vectors is not allowed, i.e. α cannot contain a parallel type: This is one of the properties enforced by the BSML type system. In the remaining, we informally write $\langle\, v_0\ ,\ldots,\ v_{p-1}\,\rangle$ for a parallel vector.

BSML provides four constants to access the BSP parameters of the parallel machine. The only one used in this paper is **bsp_p**, the number of processors of the BSP computer. BSML also provides four primitives to manipulate parallel vectors. **mkpar** and **apply** are evaluated in the pure computation phases of BSP supersteps whereas **proj** and **put** need a full super-step to be evaluated, in particular there is an implicit synchronisation barrier at the end of **proj** and **put**. The informal semantics of BSML primitives is given in Figure 2.

mkpar:(int→α)→α par is used to create a parallel vector from a function describing its components. For readers familiar with OCaml standard library, it

is similar to the function Array.init: int→(int→α)→α array but missing the first argument (the size of the create array), as the size of parallel vectors is always **bsp_p**. OCaml being a higher-order functional language, it is possible to have a parallel vector of functions. Such a parallel vector cannot be applied to values as a parallel vector of functions is not a function, thus BSML provides the primitive **apply**:(α→β)par→α par→β par to apply a parallel vector of functions to a parallel vector of values.

proj:α par→(int→α) is the inverse of **mkpar**: it creates a function from a parallel vector. Note that as processor names are represented by values of type int in OCaml, **proj** is only the inverse of **mkpar** on the interval [0,**bsp_p**-1]. The communication pattern of **proj** is a total exchange (each processor as to make its value available to all other processors). For more finely tuned communication patterns, BSML provides **put**:(int→α)par→(int→α)par. This primitive takes as input a parallel vector of functions describing the messages to be sent to other processors. For example if at processor 2, this parallel vector contains a function f_2, and that f_2 4 returns a certain value v, it means that processor 2 will send message v to processor 4. The result of **put** is also a parallel vector of functions, but these functions describe the messages *received* by the processors. Continuing the same example, after having performed the **put**, the function g_4 at processor 4 is such that g_4 2 returns v: processor 4 received message v from processor 2. By convention, the first constant constructor of a type is considered to represent the absence of message and therefore does not incur any communication. The empty list is such a value.

BSML in Coq. There are two main ways to model the semantics of programming languages in a proof assistant: shallow embedding and deep embedding [24]. Deep embedding is well suited to reason about the general properties of the language such as typing is preserved by reduction/evaluation. However it is much less convenient to reason about individual programs.

When the base language is a functional one, another possibility is to use the proof assistant as a functional language, and add the new primitives and their semantics as signatures and axioms: This is a shallow embedding. In this setting, one cannot reason about general properties of the language as it would mean to reason about the interactive prover itself, but it is much more convenient for writing and reasoning about individual programs. All the proof assistant libraries could be used.

For modelling BSML in Coq we follow the shallow embedding approach. Moreover rather than adding signatures and axioms at the top-level of Coq, which may lead to logical inconsistencies, we write a *module type*. All the applications of BSML in Coq are then written as parametric modules that take an implementation of such a module type. After we extract from Coq to OCaml, we obtain OCaml functors and we apply them to the module implementing BSML: either the sequential implementation or the parallel implementation. Note that we also provide an implementation of this module type in Coq: it is meant to check that the axioms we provide are consistent with Coq's logic, and as a by-product it is a verified sequential implementation of BSML.

```
Module Type BSP_PARAMETERS.
  Parameter p : nat.
  Axiom p_spec : 0 < p.
End BSP_PARAMETERS.
```

Fig. 3 BSP Parameters in Coq

```
Module Type BSML.  (* ... *)
  Section Parallel_vectors.
    Parameter par : Type → Type.
    Parameter get: ∀ {A: Type}, par A → pid → A.
    Axiom par_eq : ∀ {A:Type} (v v': par A), (∀ (i: pid), get v i = get v' i) → v = v'.
  End Parallel_vectors.
  Section Primitives.
    Parameter mkpar: ∀ {A:Type} (f:pid → A), par A.
    Axiom mkpar_spec: ∀ (A:Type)(f:pid→A)(i:pid), get (mkpar f) i = f i.
    Parameter apply: ∀ {A B: Type}(vf: par(A→B))(vx:par A), par B.
    Axiom apply_spec: ∀ (A B: Type) (vf: par (A → B)) (vx: par A) (i: pid),
      get (apply vf vx) i = (get vf i) (get vx i).
    Parameter put: ∀ {A:Type}(vf:par(pid→A)), par(pid→A).
    Axiom put_spec: ∀ (A:Type)(vf:par(pid→A))(i j:pid), get (put vf) i j = get vf j i.
    Parameter proj: ∀ {A:Type}(v: par A), pid → A.
    Axiom proj_spec: ∀ (A:Type)(v: par A)(i: pid), (proj v) i = get v i.
  End Primitives.
End BSML.
```

Fig. 4 BSML Primitives in Coq

The presented formalisation improves on previous ones [9,21]. It is defined as a module type and it avoids to rely too much on sigma types. Proofs using this model are easier to do than using the previous ones.

First, there is a module type for BSP parameters (Figure 3), we only consider the bsp_p parameter here, and we assume at least one processor in the BSP machine. The module type BSML models the BSML primitives. It contains a module Bsp of module type BSP_PARAMETERS and the following notation: **Notation** pid := { n:nat | ltb n Bsp.p = true } meaning that a pid is a natural number n together with a proof that n is strictly smaller than Bsp.p. ltb:nat→nat→bool is a function testing the strict ordering of two natural numbers. The remaining of the module type BSML is given in full in Figure 4.

Section Parallel_vectors models the type of parallel vectors. We called it par as in the OCaml implementation of BSML. It is an abstract type and the notation differs from an abstract type in OCaml: it takes as input a type (the type of the sequential components) and returns the type of parallel vectors of this sequential type. In order to model the semantics of BSML primitives, we need to be able to describe the value at a specific processor in a parallel vector: that is what the get operation does. Note that this operation is *not* a programming primitive of BSML. It should only be used in the logical parts. The axiom associated to this operation means that two parallel vectors are

equal if and only if their components are equal. Some representation of parallel vectors (for example functions) may not satisfy this property, thus it is important to explicitly state this property as we want to have it.

Section Primitives describe the BSML primitives. For each primitive we give its signature in a style very close to their signature in OCaml, but in the handling of polymorphism. The semantics of the primitive is given by an axiom whose name ends by _spec. For example the axiom of mkpar, ∀ i:pid, get (mkpar f) i = f i clearly states that for all process identifiers i, the value held at processor i in parallel vector mkpar f is equal to f i. It is exactly a quantified version of the informal semantics of mkpar shown in Figure 2.

Using this module type, it is possible to write BSML programs in Coq and to reason about their semantics using the specifications of the BSML primitives. One possible application is to program BSML algorithmic skeletons in Coq.

4 Algorithmic Skeletons and Automatic Parallelisation in Coq

Algorithmic Skeletons in Coq. Writing algorithmic skeletons using BSML primitives in Coq is actually very close to doing the same in OCaml and BSML. We illustrate this on the implementation of map and reduce skeletons. To implement programs using the BSML primitives, one needs to implement a parametric module taking as argument a module that implements the BSML primitives:

```
Module MapReduce (Bsml : BSML).
```

We use two utility functions for that:

```
Definition replicate A (x:A) : par A := mkpar(fun _ ⇒ x).
Lemma replicate_spec: ∀ A (x:A) i, get (replicate x) i = x.
Definition parfun A B (f:A→B)(v:par A) : par B := apply (replicate f) v.
Lemma parfun_spec: ∀ A B (f:A→B) v i, get(parfun f v) i=f(get v i).
```

For these functions we follow the same modelling convention as for the primitives, but here the terms are given, by first defining a function then a lemma describing its specification. We omit the proof scripts but using automated tactics, they are very short. It is then easy to define a parallel map on distributed lists (here distributed lists means parallel vectors of lists):

```
Definition par_map A B (f:A→B) : par(list A)→par(list B) := parfun (map' f).
```

where map' is a tail recursive version of map. The parallel reduce can be defined using the proj primitive for communications:

```
Definition par_reduce'(op:A→A→A)'{H:Monoid op e}(v:par(list A)) : A :=
  let local := parfun (reduce op) v in       (* local reductions *)
  let list  := List.map (proj local) pids in (* vector → list    *)
  reduce op list.          (* reduction of the partial reductions *)
```

In the previous definition, symbol ‘ in ‘{var:ty} indicates that the free variables in ty should be generalised, i.e. introduced before {var:ty} as additional implicit arguments. In this version of par_reduce the final result is not a parallel value. We can also have an alternative version where the result is a parallel vector with the same value on each of the processors. We choose here a simple algorithm, using put:

```
Program Definition par_reduce' {A}(op:A→A→A){e:A}{H:Monoid op e}
  (v: par(list A)) : {v:par A | ∀ i, get v i = get v first } :=
  let local := parfun (reduce op) v in
  let msgs  := put(apply(mkpar(fun pid msg dst⇒msg)) local) in
  let lists := parfun (fun f⇒List.map f pids) msgs in
  parfun (reduce op) lists.
Next Obligation. (* omitted, 9 lines *) Qed.
```

In this alternative version, the return type is a parallel vector, and we use the ability of Coq to express post-conditions: this vector has the same value on all the processors. The **Program** feature of Coq allows to define the function as if no post-condition was present. It generates a proof obligation (corresponding here to this post-condition). The proof script is omitted here, but is short.

To verify that the implementations correspond to the semantics we have in mind, we could write _spec lemmas associated to each of these functions. However, it is not convenient to express the specification of par_map and par_reduce by describing the content of their output at each processor. As these functions are algorithmic skeletons, we rather prefer to express the correspondence they have with sequential functions.

Types and Functions Correspondences. To do so, we rely on two main type classes (defined in the Core.Parallelisation library of SyDPaCC). First we need to express that a parallel type is a correct parallelisation of a sequential type.

Definition 1 (Type correspondence) There is a correspondence between a type A and a parallel type A_p (i.e. that contains at least one par) if and only if there exists a total surjective function $join_A : A_p \to A$.

The totality of the function ensures that we can construct a sequential value for any value in A_p and its surjectivity ensures the existence of at least one parallel representation for any sequential element.

In Coq we formalise these notions as type classes:

```
Class Surjective A B (f:A→B) :=
  { surjective :  ∀ y : B,  ∃ x : A,  f x = y  }.
Class TypeCorr (seq_type:Type) (par_type:Type) (join:par_type→seq_type) :=
  { type_corr :> Surjective join }.
```

Examples of type correspondences are given in Figure 5. In module ParList, the join function transforms a parallel vector of lists into a list. This sequential list also provides a “global view” of the parallel vector of lists: We think of the parallel vector as a big list obtained by concatenation of the lists hold by the

```
Module ParList (Import Bsml: BSML)(Import Pid : Pid.TYPE Bsml.Bsp).
  Program Definition join A (v:par(list A)) : list A :=
    List.flat_map (proj v) pids.
  Program Instance surjective_join A : Surjective (join(A:=A)).
  Next Obligation. (* Omitted, 12 lines *) Qed.
  Program Instance list_par_list_corr A : TypeCorr (@join A).
End ParList.

Module ReplicatePar (Import Bsml: BSML)(Import Pid : Pid.TYPE Bsml.Bsp).
  Program Definition join A (r:{v:par A|∀ i,get v i=get v first}) : A :=
    (proj (proj1_sig r)) first.
  Program Instance surjective_join A : Surjective (join (A:=A)).
  Next Obligation. (* Omitted, 5 lines *) Qed.
  Program Instance replicate_par_corr A : TypeCorr (@join A).
End ReplicatePar.
```

Fig. 5 Type Correspondences

processors, in increasing order of processor identifier. The proof that join is surjective is quite easy: For a sequential list, it is enough to build the parallel vector that contains this list at one specific processor (for example the first one) and empty lists on all other processors.

In module ReplicatePar, join takes as input parallel vectors that contain the same value at all the processors, and returns this value. It is straightforward to show join is surjective, as it is enough to replicate the sequential value.

Using type correspondence, we can define a relation between sequential and parallel functions.

Definition 2 (Function correspondence) Given two types A and B, and their parallel correspondent A_p and B_p, there is a correspondence between the sequential function $f : A \to B$ and the parallel function $f_p : A_p \to B_p$ if and only if $join_B \circ f_p = f \circ join_A$.

In other words, the following diagram commutes:

$$\begin{array}{ccc} A_p & \xrightarrow{f_p} & B_p \\ {\scriptstyle join_A}\downarrow & & \downarrow{\scriptstyle join_B} \\ A & \xrightarrow{f} & B \end{array}$$

This notion is also modelled as a type class:

```
Class FunCorr `{ACorr:TypeCorr A Ap join_A}`{BCorr:TypeCorr B Bp join_B}
  (f:A→B)  (fp:Ap→Bp) := { fun_corr : ∀ ap,  join_B (fp ap) =  f (join_A ap) }.
```

Of course a parallel function may take a sequential value as input and returns a parallel value, or may take as input a parallel value and returns a sequential value (such as the first version of par_reduce does). In these cases the diagrams are:

and the type classes are:

```
Class LeftFunCorr `{ACorr:TypeCorr A Ap}{B}`(f:A→B)(fp:Ap→B) :=
  { left_fun_corr : ∀ ap, fp ap = f (join ap) }.
Class RightFunCorr {A}`{BCorr:TypeCorr B Bp}`(f:A→B)(fp:A→Bp) :=
  { right_fun_corr : ∀ a, join (fp a) = f a }.
```

There are two kinds of instances for the function correspondences: general ones and instances stating the correspondence of a specific parallel function and a specific sequential function. The former are also given in the library Parallelisation, the other are together with the skeleton definitions. There are several general instances that state facts about the correspondence of compositions. For two instances of FunCorr, we have for example:

```
Program Instance fun_corr_comp_fun_corr `{ACorr : TypeCorr A Ap join_A}
      `{BCorr : TypeCorr B Bp join_B} `{CCorr : TypeCorr C Cp join_C}
      `{fCorr : @FunCorr A Ap join_A ACorr B Bp join_B BCorr f fp}
      `{gCorr : @FunCorr B Bp join_B BCorr C Cp join_C CCorr g gp} :
  FunCorr (compose g f) (compose gp fp).
Next Obligation. (* Omitted, 3 lines *) Qed.
```

The composition of the two sequential functions corresponds to the composition of the two corresponding parallel functions. Note that the directions of the arrows in our diagrams are important to have such a compositional notion.

Now that we can express the correspondence between a sequential function and a skeleton, we can do it for par_map:

```
Program Instance map_par A B (f:A→B): FunCorr (map f) (par_map f).
Next Obligation. (* Omitted, 5 lines *) Qed.
```

This piece of code assumes that the instance defined in the module ParList described above (Figure 5) is in the context. It is this type correspondence that is used both for input and output.

In the case of par_reduce, the same type correspondence is used only for input, as the output is the same type A in the sequential and parallel cases:

```
Program Instance reduce_par_reduce `(op:A→A→A) `{Monoid A op e} :
  LeftFunCorr (reduce op) (par_reduce op).
Next Obligation. (* Omitted, 8 lines *) Qed.
```

The correctness of the last skeleton par_reduce' uses the type correspondence defined in the module ReplicatePar (Figure 5):

```
Program Instance reduce_par_reduce' `(op:A→A→A) `{Monoid A op e} :
  FunCorr (reduce op) (par_reduce' op).
```

Note that as the instances we have do not overlap, the instance resolution mechanism has no problem to find the instance we expect. In a richer library with overlapping instances (e.g. with different joins on parallel vectors of lists) the function correspondence might need to be specified more carefully by the user (i.e. some implicit arguments might need to be made explicit). However, as the library is organised in a modular way, the user has control over the visibility of overlapping instances.

Automatic Parallelisation. If a sequential function h is defined as a composition of maps and reduces, the instances we defined can actually serve as an automatic parallelisation mechanism. Consider the following function:

```
Definition parallel '(f:A→B)
  '{ACorr: TypeCorr A Ap joinA} '{BCorr : TypeCorr B Bp joinB}
  '{fCorr: @FunCorr A Ap joinA ACorr B Bp joinB BCorr f fp} : Ap→Bp := fp.
```

This function seems uninteresting as the result it returns is one of its arguments. However all the arguments, except the sequential function f, are *implicit* and are *instances* of type classes. It means that **Definition** par_h:=parallel h launches the instance resolution mechanism of Coq. Function h will first be decomposed if it is a composition, until instances of correspondence between a sequential function and a skeleton are found, and the parallel version par_h of h will be automatically built as a composition of skeletons.

The last component of the framework we need is thus a way to prove that certain classes of functions can be written as compositions of sequential functions that correspond to skeletons. The Bird Meertens Formalism, with the theory of list homomorphisms is a convenient formalism for this purpose.

5 List Homomorphisms and their Theorems in Coq

In our parallelisation framework, we need functions defined as compositions of map and reduce. We consider the class of $\oplus$-homomorphic functions.

Definition 3 ($\oplus$-homomorphic) A function h on lists is $\oplus$-homomorphic if for all lists x and y, $h(x +\!\!+ y) = (h\ x) \oplus (h\ y)$.

Fact 1 *If h is $\oplus$-homomorphic, then $(img\, h,\ \odot,\ h\,[])$ is a monoid.*

Note this it is not true in general on the codomain of h. There are two useful theorems for our framework: the first and the third homomorphism theorems [11].

Theorem 1 (Third Homomorphism Theorem) *If a function h is both:*

- *$\oplus$-leftwards, i.e. there exists $\oplus$ such that $h([x] +\!\!+ xs) = x \oplus (h\ xs)$,*
- *$\otimes$-rightwards, i.e. there exists $\otimes$ such that $h(xs +\!\!+ [x]) = (h\ xs) \otimes x$,*

then there exist $\odot$ such that h is $\odot$-homomorphic.

Theorem 2 (First Homomorphism Theorem) *If h is $\odot$-homomorphic then $h = (reduce\ \odot)\ \circ\ (map\ f)$ where $\forall x, fx = h[x]$.*

In this first theorem, we consider the restriction of $\odot$ on the image of h, as in general $(codomain\ h,\ \odot,\ h\ [])$ is not a monoid (and *reduce* requires a monoid).

In Coq, we model the homomorphic property as a class:

```
Class Homomorphic `(h:list A→B) `(op:B→B→B) :=
  { homomorphic : ∀ x y, h (x++y) = op (h x) (h y) }.
```

Fact 1 is actually a result that is not trivial in Coq. First we need to define the image of h, and definitions to build the restriction of $\oplus$ (named op from now on) to the image of h:

```
Definition img `(h:list A→B) := { b:B | ∃ l, h l = b }.
Program Definition restrict_op `(op:B→B→B) `{Homomorphic A B h op} :
  img h → img h → img h := op. (* ... *)
Definition to_img A B (h:list A→B)(xs:list A) : img h := (*...*).
```

Fact 1 is then written:

```
Program Instance homomorphic_restrict_op_monoid `{Homomorphic A B h op} :
  Monoid (A:=(img h)) (restrict_op op) (to_img h []).
```

Actually, we need a slightly more general result, where we replace the restriction of the operator by a function that is extensionally equivalent:

```
Program Definition restrict `{Homomorphic A B h op}
 `(Eq:∀ a b, op' a b= ` (restrict_op op a b)): img h→img h→img h:=op'(* ... *)
Program Instance homomorphic_restrict_monoid `{Homomorphic A B h op}
 `(Eq:∀ a b, op' a b = ` (restrict_op op a b)) : Monoid (restrict Eq) (h []).
(* 20 lines+usage of several omitted lemmas *)
```

This allows to replace the output of the third homomorphism theorem by a simplified version of the binary operator.

To express the first homomorphism theorem, we define a function that produces the composition of map and reduce from an op-homomorphic function and possibly optimised versions of op and of fun (x:A)⇒h[a]:

```
Definition hom_to_map_reduce `(h:list A→B) `{H:Homomorphic A B h op}
 `{@Optimised_op A B h op H}`{@Optimised_f A B h op H} : list A→img h :=
 (reduce (optimised_op h)) ∘ (List.map (optimised_f h)).
```

There are default instances of the Optimised_op and Optimised_f that relate op and fun (x:A)⇒h[a] to themselves. We omit the details of the optimisation aspects and refer to the source code of the framework. The theorem then checks that this function produces a function equal to h:

```
Theorem first_homomorphism_theorem: ∀ `{H:Homomorphic A B h op}
   `{@Optimised_op A B h op H}`{@Optimised_f A B h op H},
 ∀ l, h l = of_img (hom_to_map_reduce h l).
Proof. (* 12 lines *) Qed.
```

The classical proof of the third homomorphism theorem relies on the notion of weak right inverse, i.e. a function h' such that for all x, $h\,x = h(h'(h\,x))$. In [11], a lemma states that for every computable total function h with enumerable domain, there exists a weak right inverse of h. The proof however is the following: "*To compute $g\,t$ for some t simply enumerate the domain of h and return the first x such that $h\,x = t$. If t is in the range of h then this process terminates.*" The problem with this proof is that: function g may not terminate but it is not possible to define non-terminating functions in Coq. Moreover even if it could be defined, its efficiency would be in general catastrophic.

Therefore we state and prove a weak form of the third homomorphism theorem using the following classes:

```
Class Rightwards `(h:list A→B)`(op:B→A→B)`(e:B) :=
  { rightwards: ∀ l, h l = List.fold_left op l e }.
Class Leftwards `(h:list A→B)`(op:A→B→B)`(e:B) :=
  { leftwards: ∀ l, h l = List.fold_right op e l }.
Class Right_inverse `(h:list A →B)(h':B→list A) :=
  { right_inverse: ∀ l, h l = h(h'(h l)) }.

Instance third_homomorphism_theorem `{h:list A→B}
`{inv:Right_inverse A B h h'}
`{Hl:Leftwards A B h opl e}`{Hr:Rightwards A B h opr e} :
  Homomorphic h (fun l r ⇒h( (h' l)++(h' r))).
Proof. (* 20 lines + usage of a 10 lines lemma + lemmas on folds *) Qed.
```

To parallelise a function h, we should prove instances of Rightwards h oplus, Leftwards h otimes and Right_inverse h h' and call the hom_to_map_reduce function on h. The resolution instance mechanism of Coq would then produce an instance of Homomorphic for h using the third homomorphism theorem.

The obtained composition could then be parallelised using parallel (or variant left_parallel) explained in the previous section. We illustrate this process on an application example in the following section.

6 An Example: Maximum Prefix Sum

The goal is to obtain a parallel function that solves the maximum prefix sum problem. An example of evaluation of the sequential function *mps* follows (the prefix whose sum is maximum is underlined): $mps\ [\underline{1;2;-1;2;-1;3};-4] = 6$.

A trivial solution that we consider to be a specification follows, where prefix returns the list of all the prefixes of its input and t is the type of an abstract representation of mathematical integers (module type Number):

```
Program Definition sum : list t → t := reduce add.
Definition maximum : ∀(l:list t), NonEmpty l→t := NE.reduce max.
Program Definition mps_spec : list t → t := maximum∘'(map sum) ∘''prefix.
```

maximum is not defined on the empty list: we use a version of reduce that also requires to be applied to a non-empty list. The property NonEmpty is a class:

```
Class NonEmpty '(l:list A) := { non_emptiness : l<> [] }.
```

The variants of composition handle the additional arguments concerning non-emptiness. This is transparent to the user as there are instances of the NonEmpty class that state that prefix always returns a non-empty list and that map preserves non-emptiness.

mps_spec is not leftwards. But it is possible to tuple mps_spec with sum:

```
Definition tupling A B C(f:A→B)(g:A→C) := fun x ⇒ (f x, g x).
Definition ms_spec := tupling mps_spec sum.
```

ms_spec is opl-leftwards and opr-rigthwards:

```
Definition opl (a:t) (b:t*t) : t*t := ( max 0 (a + fst b), a + (snd b) ).
Instance ms_leftwards : Leftwards ms_spec opl (0,0).  Proof. (* ... *) Qed.
Definition opr (a:t*t) (b:t) : t*t := (max (fst a) ((snd a)+b),(snd a)+b).
Instance ms_rightwards : Rightwards ms_spec opr (0,0).
```

The proofs of the instances are about 20 lines long and they both use two 10 lines lemmas about maximum and sum. Then we need to find a weak right inverse of ms_spec:

```
Definition ms' (p:t*t) := let (m,s) := p in [ m; s + −m].
Program Instance ms_right_inverse : Right_inverse ms_spec ms'.
Next Obligation. (* 25 lines *) Qed.
```

The third homomorphism is applied to show that ms_spec is $\odot$-homomorphic where $\odot$ =fun l r⇒ms_spec(ms' l++ms' r). In the pen-and-paper proof, usually one shows that $(a_m, a_s)\odot(b_m, b_s) = (0 \uparrow a_m \uparrow (a_s{+}b_m), a_s{+}b_s)$ where $\uparrow$ returns the maximum of two numbers. Using the Optimised_op class, it is also possible to do it in Coq, in a process similar to the pen-and-paper proof (i.e. the final version is not known before the proof starts). This requires about 30 more lines using several short lemmas on mps_spec and sum.

To parallelise, we need to instantiate the parametric modules presented in the previsous sections and then call the left_parallel function (or the parallel function using the ReplicatePar in addition to ParList). Finally to obtain only the mps component of the result, we compose with projections (Figure 6).

The Coq code can then be extracted towards OCaml. The extraction mechanisms removes the logical parts to keep only the computational parts. What we obtain is an OCaml functor:

```
module MPS=functor(Bsml:BSML)→functor(N:Number)→struct (*...*) end
```

A parallel implementation on top of MPI results from applying this functor to a Bsml module that is a wrapper around the Bsmlmpi uncertified implementation of BSML in OCaml, C and MPI. The main difference between Bsml and Bsmlmpi is that in the former pid is the type nat and in the latter it is type int. As we dealt with an abstract representation of integers, we also need to provide an OCaml module that follows the extraction of the module type Number

```
Module MPS (Import Bsml: Core.BSML)(N: Number).
  Module Pid      := Pid.Make Bsml.Bsp.
  Module ParList  := Correspondences.ParList Bsml Pid.
  Module MapReduce := MapReduce.Make Bsml Pid ParList ReplPar.
  Module Mps := Make N.
  Definition par_ms := Eval sydpacc in left_parallel (hom_to_map_reduce Mps.ms_spec).
  Definition par_mps := Eval simpl in fst ∘ of_img ∘ par_ms.
End MPS.
```

Fig. 6 Parallelisation

(basically a type for numbers plus basic operations on them). The application module would look as follows: **module** App = Mps.MPS_Parallel (Bsml) (Nint).

The framework and results of scalability experiments are available online[3]. The framework only requires Coq 8.4, OCaml 4 or above, and an MPI library.

7 Related Work

To our knowledge SyDPaCC is the only approach in which actual source code of scalable and correct parallel programs can be obtained from a development in a proof assistant. The theory of constructive parallel algorithms provides various ways to ease the systematic development of correct parallel programs. However, the transformations are not mechanically verified and the transition from an efficient functional expression to a parallel program is not based on a formal basis. Most often, with the exception of [4,1], the semantics of algorithmic skeletons remains informal.

There exist several logics to reason about BSP programs, for example [19], but none of them is mechanised in a proof assistant, and only one of them is related to a tool that can provide actual source code [26]. The main differences with our approach is that: LOGS starts from an imperative and local view of parts of the program to build a larger one by parallel composition whereas we start from a functional and global view and our framework is in Coq. BSP-Why [8] is an extension of the Why2 system for the deductive verification of imperative BSP programs. The tool is based on the generation of verification conditions from users annotations, those conditions being automatically proved by SMT solvers, or interactively with Coq. BSP-Why does not support program transformation and higher-order functions.

Although there are systems to support program calculation (for example [25]) they lack the rich set of theories that exist for Coq and that can be reused for program transformation. Moreover Coq offers a more trusted framework through its kernel. The work on polytypic programming and program transformation in Coq [17] and Agda [18] is also related. Our framework follows more closely a simple functional programming style with additional

[3] `http://traclifo.univ-orleans.fr/SyDPaCC`, version core-0.2

pre/post conditions or statements about the simple functional programs. Moreover for the parallel aspects it would be a challenge to provide correspondence between polytypic sequential functions and efficient parallel ones.

To our knowledge, there are only three works associating data parallelism and proof assistants. The operational semantics of a type safe subset of Data Parallel C is formalised in Isabell/HOL [6]. In this approach Isabelle/HOL expressions that represent programs are generated and manipulated. It is therefore a deep embedding approach. Swierstra [20] formalised mutable arrays in Agda, and added explicit distributions to these arrays. He used it to reason about a distributed map and distributed sum on these arrays. In BSML the distribution of parallel vectors is fixed but it is possible to define a higher-level data structure on top of parallel vector and consider various distributions of the data structure in parallel vectors [3]. Moreover it is possible to formalise mutable arrays, and even extract such imperative programs to OCaml as done by Malecha et al. [16]. Lupinski et al. [15] formalised the semantics of a skeletal parallel programming language. It is also a deep embedding that models both the high-level semantics of skeletons and their implementations in a formalised JoCaml. BSML shallow embedding is more convenient.

8 Conclusion

The SyDPaCC framework shows that the Coq proof assistant is suited for program calculation in the Bird Meertens Formalism tradition, and that when using the right features of Coq, the additional work to have machine checked proofs rather than pen-and-paper proofs is quite manageable. Coq also allows automatic parallelisation using algorithmic skeletons. The key points in this respect is that we provide a shallow embedding of a pure functional parallel programming library in Coq; that one can program and verify algorithmic skeletons in Coq based on this embedding; and that expressing the correctness of algorithmic skeletons with respect to usually sequential functions in a compositional way as instances of type classes makes possible the use of the instance resolution mechanism of Coq for automatic parallelisation. Actual programs are obtained through the extraction feature. The extracted code is combined with the unverified parallel implementation of BSML in OCaml and C+MPI. The core SyDPaCC presented in this paper is very concise (2 kLoC of Coq, 600 LoC of OCaml 120 LoC of C). Experimental results on the generated programs are presented in [7,14,21] and in the SyDPaCC website.

Acknowledgements This work is partly supported by ANR (France) and JST (Japan) (project PaPDAS ANR-2010-INTB-0205-02 and JST 10102704).

References

1. Aldinucci, M., Danelutto, M.: Skeleton-based parallel programming: Functional and parallel semantics in a single shot. Comput Lang Syst Str 33(3-4), 179–192 (2007)

2. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall (1996)
3. Bousdira, W., Loulergue, F., Tesson, J.: A Verified Library of Algorithmic Skeletons on Evenly Distributed Arrays. In: Algorithms and Architectures for Parallel Processing (ICA3PP). pp. 218–232. No. 7439 in LNCS, Springer, Fukuoka, Japan (2012)
4. Cavarra, A., Riccobene, E., Zavanella, A.: A formal model for the parallel semantics of P3L. In: ACM Symposium on Applied Computing (SAC). pp. 804–812. ACM (2000)
5. Cole, M.: Parallel Programming with List Homomorphisms. Parallel Processing Letters 5(2), 191–203 (1995)
6. Daum, M.: Reasoning on Data-Parallel Programs in Isabelle/Hol. In: C/C++ Verification Workshop (2007)
7. Emoto, K., Loulergue, F., Tesson, J.: A Verified Generate-Test-Aggregate Coq Library for Parallel Programs Extraction. In: Interactive Theorem Proving (ITP). pp. 258–274. No. 8558 in LNCS, Springer, Wien, Austria (2014)
8. Fortin, J., Gava, F.: BSP-Why: A tool for deductive verification of BSP algorithms with subgroup synchronisation. Int J Parallel Prog pp. 1–24 (2015)
9. Gava, F.: Formal Proofs of Functional BSP Programs. Parallel Processing Letters 13(3), 365–376 (2003)
10. Gava, F., Gesbert, L., Loulergue, F.: Type System for a Safe Execution of Parallel Programs in BSML. In: 5th ACM SIGPLAN workshop on High-Level Parallel Programming and Applications. pp. 27–34. ACM (2011)
11. Gibbons, J.: The third homomorphism theorem. Journal of Functional Programming 6(4), 657–665 (1996)
12. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
13. Loulergue, F., Gava, F., Billiet, D.: Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In: International Conference on Computational Science (ICCS). LNCS, vol. 3515, pp. 1046–1054. Springer (2005)
14. Loulergue, F., Robillard, S., Tesson, J., Légaux, J., Hu, Z.: Formal Derivation and Extraction of a Parallel Program for the All Nearest Smaller Values Problem. In: ACM Symposium on Applied Computing (SAC). pp. 1577–1584. ACM, Gyeongju, Korea (2014)
15. Lupinski, N., Falcou, J., Paulin-Mohring, C.: Sémantique d'une langage de squelettes. `http://www.lri.fr/~paulin/Skel/article.pdf` (2012)
16. Malecha, G., Morrisett, G., Wisnesky, R.: Trace-based verification of imperative programs with i/o. J. Symb. Comput. 46(2), 95–118 (2011)
17. Mu, S.C., Ko, H.S., Jansson, P.: Algebra of programming using dependent types. In: Audebaud, P., Paulin-Mohring, C. (eds.) Mathematics of Program Construction, LNCS, vol. 5133, pp. 268–283. Springer (2008)
18. Mu, S., Ko, H., Jansson, P.: Algebra of programming in Agda: Dependent types for relational program derivation. J Funct Program 19(5), 545–579 (2009)
19. Stewart, A., Clint, M., Gabarró, J.: Barrier synchronisation: Axiomatisation and relaxation. Formal Aspects of Computing 16(1), 36–50 (2004)
20. Swierstra, W.: More dependent types for distributed arrays. Higher-Order and Symbolic Computation 23(4), 489–506 (2010)
21. Tesson, J., Loulergue, F.: A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation. In: International Conference on Computational Science (ICCS). pp. 36–45. Elsevier, Singapore (2011)
22. The Coq Development Team: The Coq Proof Assistant. `http://coq.inria.fr`
23. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM 33(8), 103 (1990)
24. Wildmoser, M., Nipkow, T.: Certifying machine code safety: Shallow versus deep embedding. In: Slind, K., Bunker, A., Gopalakrishnan, G. (eds.) Theorem Proving in Higher Order Logics, LNCS, vol. 3223, pp. 133–142. Springer (2004)
25. Yokoyama, T., Hu, Z., Takeichi, M.: Yicho: A system for programming program calculations. Tech. Rep. METR 2002–07, Department of Mathematical Engineering, University of Tokyo (Jun 2002)
26. Zhou, J., Chen, Y.: Generating C code from LOGS specifications. In: 2nd International Colloquium on Theoretical Aspects of Computing (ICTAC'05). pp. 195–210. No. 3407 in LNCS, Springer (2005)