Verified Scalable Parallel Computing with Why3

Olivia Proust and Frédéric Loulergue(✉)

Univ Orléans, INSA CVL, LIFO EA 4022, Orléans, France
olivia.proust@etu.univ-orleans.fr, frederic.loulergue@univ-orleans.fr

Abstract. BSML is a pure functional library for the multi-paradigm language OCaml. BSML embodies the principles of the Bulk Synchronous Parallel (BSP) model, a model of scalable parallel computing. We propose a formalization of BSML primitives with WhyML, the specification language of Why3 and specify and prove the correctness of most of the BSML standard library. Finally, we develop and verify the correctness of a small BSML application.

Keywords: scalable parallel computing · functional programming · deductive verification · interactive theorem proving

1 Introduction

High-level approaches to big data analytics such as Hadoop MapReduce [26] or Apache Spark [1] are often inspired by bulk synchronous parallelism (BSP) [25] a model of scalable parallel computing. In this context, scalable means that the number of processors of the parallel machines running BSP programs could range from a few to several thousand cores or more. Bulk Synchronous Parallel ML (BSML) [19] is a pure functional library for the multi-paradigm language OCaml[1]. BSML embodies the principles of the BSP model, at a higher level than libraries such as the BSPlib library [14] and can easily express patterns [13,17] (or algorithmic skeletons [4]) of frameworks such as MapReduce or Spark.

Why3 [2,3] is a framework for the deductive verification of programs. It provides a specification and programming language named WhyML which can be used directly or as an intermediate language for other tools to verify C [15], Java [9], Ada or Rust [7] programs. The framework itself also provides mini-C and mini-Python front-ends. Why3 generates verification conditions to be verified by external provers. A strong point of Why3 is that it targets a large variety of provers including Alt-Ergo [5], Z3 [21] and CVC5. Correct-by-construction OCaml code can be extracted from WhyML.

Our contributions are the formalization of BSML and its standard library in WhyML and its use in the specification and verification of a scalable parallel function for the maximum prefix sum problem, using map and reduce skeletons.

[1] https://ocaml.org.

C. Ferreira and T. A. C. Willemse (Eds.): SEFM 2023, LNCS 14323, pp. 246–262, 2023.
https://doi.org/10.1007/978-3-031-47115-5_14

The remainder of the paper is organized as follows. In Sect. 2, we give an overview of Why3 and WhyML, including its limitations when dealing with higher-order functions. We introduce functional bulk synchronous parallel programming with BSML in Sect. 3. Section 4 is devoted to the formalization of the primitives of BSML and its application to the specification and verification of the BSML standard library. We consider the specification, development and verification of a small application: a parallel function that solves the maximum prefix sum problem in Sect. 5. We discuss related work in Sect. 6 and conclude in Sect. 7.

The set of Why3 modules is called WhyBSML and is available at:

https://doi.org/10.5281/zenodo.8166092.

2 An Overview of Why3

2.1 Specifying and Verifying Functional Programs with Why3

Why3 is often use in the verification of imperative programs. As BSML is purely functional and BSML applications mostly used the functional features of OCaml, we focus here on the verification of functional programs. This focus is also a necessity as we will explain in the next subsection.

In addition to its core features, Why3 provides a standard library with data structures such as lists and arrays, as well as basic arithmetic logic with integers and reals. We illustrate this short introduction with the example of Fig. 1. Note that this figure presents a pretty-printed version of the actual code, for example /\ is rendered as $\wedge$, `->` as $\rightarrow$, `'a` as α, *etc.*

WhyML developments are organized in *modules*. The example defines two modules: `Max` (lines 1–8) and `MaxList` (lines 10-32). Defined modules can be used in other modules with the `use` keyword. We use some modules of the Why3 standard library: `int.Int` about integer arithmetic (lines 2 and 11) and `list.List`, `list.Length`, `list.NthNoOpt` for basic definitions and facts about lists (lines 12–14).

The module `Max` is devoted to the specification and definition of a function `max` which returns the larger of two integers. This function does not have any pre-condition but its post-conditions are introduced by the keyword `ensures`.

Assuming the file `maximum.mlw` contains only the module `Max`, verifying that `max` satisfies its preconditions using the prover Alt-Ergo can be done with the following command:

```
why3 prove --prover alt-ergo maximum.mlw
```

and the tool answers `max` indeed satisfies its contract:

```
File maximum.mlw:
Goal max'vc.
Prover result is: Valid (0.00s, 8 steps).
```

```
1  module Max
2      use int.Int
3
4      let max (x : int) (y : int) : int
5        ensures { result = x ∨ result = y }
6        ensures { result ≥ x ∧ result ≥ y }
7      = if x < y then y else x
8  end
9
10 module MaxList
11   use int.Int
12   use list.List
13   use list.Length
14   use list.NthNoOpt
15   use Max
16
17   function ([]) (l : list α) (i : int) : α= nth i l
18
19   let rec maximum (l : list int) : int
20     requires { length l > 0 }
21     ensures { ∀ i:int. 0 ≤ i < length l →result ≥ l[i] }
22     ensures { ∃ i:int. 0 ≤ i < length l ∧ l[i] = result }
23     variant { l }
24   = match l with
25     | Nil → absurd
26     | Cons h Nil →h
27     | Cons h t →
28       let _ = assert { ∀ i:int. 0 ≤ i < length t →
29                                  l[i+1] = t[i] } in
30       max h (maximum t)
31     end
32 end
```

Fig. 1. A WhyML Example

In our example, most of the functions to verify are recursive and often manipulate lists. Lines 19–31 are an example of a recursive function that takes a list of integers and returns the highest value the list contains.

To write the contract of function `maximum`, we use the notation `l[i]` to access the i^{th} element of list `l`. This notation is defined as a binary function in line 17 and is actually an alias for the `nth` function of the standard library. Note that this definition is introduced by the keyword `function` instead of the keyword `let` (as in line 4). The purpose of `([])` is to be used only in specifications while `max` is code that is meant to be executed. Pure functions may be used in both roles if they are defined using both keywords. In this example, `max` cannot be used in assertions while the bracket notation cannot be used in programs.

For `maximum`, we have a larger contract with new clause types. We add a pre-condition (following the keyword `requires`) to this contract, due to the fact

that our function is not defined on empty lists. To ensure termination, we define a `variant`, which must be decreasing with each recursive call. The recursive call in line 30 is indeed call on the tail of the input list, thus this call is made on a strictly smaller argument than `l`. The variant can be a term of any type as long as this type comes with a well-founded order relation. It can even be a sequence of terms: in this case, lexicographic ordering is used.

We need quantifiers to express our post-conditions. The maximum value must be contained in the list (line 22 using $\exists$), and must be greater than or equal to all the values in the list (line 21 using $\forall$).

The definition of the function follows in lines 24–30. It proceeds by pattern matching on the input list. The case of the empty list (constructor `Nil`) is `absurd` as the pre-condition specifies the input list should not be empty (expressed as a fact on its length in line 20). If the list is a singleton (case `(Cons h Nil)`), the result is of course the only element of the list. Otherwise — and let us ignore lines 28–29 for the moment — the result is the maximum of the head and the recursive call on the tail (line 30). Without lines 28–29, the execution of the tool now answers:

```
File maximum.mlw:
Goal max'vc.
Prover result is: Valid (0.00s, 8 steps).
File maximum.mlw:
Goal maximum'vc.
Prover result is: Timeout (5.00s).
```

Using Z3 or CVC5, or increasing the timeout, or changing the proof strategy does not change the outcome. It is possible to apply transformations to the goals. Using the Why3 IDE, just splitting the verification condition for `maximum` gives five verification conditions: one for verifying the empty case is indeed absurd, one to check that the recursive call is indeed decreasing, one to check the pre-condition of the recursive call and one for each post-conditions. All these sub-goals are valid except for the one corresponding to the post-condition in line 22 remain unknown. To help the provers, we added lines 28–29 which relate elements of `l` with elements of its tail via `nth`. This assertion is easily verified and then eases the verification of the post-condition. The answer of the tool changes to:

```
Prover result is: Valid (0.09s, 749 steps).
```

2.2 Limitations with Higher-Order Functions

To show the limitations of Why3 in handling higher-order functions, let us consider the example of Fig. 2. Intuitively, `option` α extends the type α with a value `None` and all the other values are encapsulated in the constructor `Some`.

In lines 1–10, we define a module `Concrete` containing the definition of a function `remove_option` that extracts the value encapsulated in an optional value assuming this value is not `None`. In the module `Failure`, we apply this function but through a higher-order function `apply` that just applies a function

```
1  module Concrete
2    use export option.Option
3
4    let remove_option (opt : option α) : α
5      requires { opt ≠None } ensures { (Some result) = opt }
6    = match opt with
7      | Some x → x
8      | None → absurd
9      end
10 end
11 module Failure
12   use Concrete
13   let apply (f:α→β)(a:α) : β= f a
14
15   let test_KO (c: option α) : α(* CANNOT BE VERIFIED *)
16     requires { c ≠None } ensures { Some(result) = c }
17   = apply remove_option c
18 end
19 module Abstract
20   use export option.Option
21
22   val function remove_option(opt: option α) : α
23   axiom remove_option: ∀ x: α. remove_option(Some x) = x
24 end
25 module Success
26   use Abstract
27   let apply (f:α→β)(a:α) : β= f a
28
29   let test_OK (c: option α) : α
30     requires { c ≠None } ensures { Some(result) = c }
31    = apply remove_option c
32 end
```

Fig. 2. Limitations with Higher-Order Functions

to a value. The tool fails to verify the function `test_KO` which intuitively does exactly the same as `remove_option`. Note that if `remove_option` was performing side effects or was partial because it may raise exceptions, Why3 would reject the program with an error. Here the problem is less visible. Indeed, the arguments of a higher-order function must be purely functional and *total* functions. In our case `remove_option` is not total as its pre-condition excludes `None`. The manifestation of the problem can be seen in a sub-verification condition generated by Why3: ∀ `opt:option α. opt` ≠`None`, which is impossible to prove.

Still, as most BSML primitives are higher-order functions, and we need to use functions such as `remove_option`, a work-around was needed. Our solution is shown in module `Abstract` (lines 19–24). Instead of writing a concrete implementation of `remove_option`, we *declare* a function `remove_option` without defining it, and we only give its semantics (with an `axiom`) when the pre-condition is met.

It looks like a total function but if its application does not satisfy the precondition then it is impossible to reason about the result of the application. If the overall verification of a client code works despite an incorrect application of `remove_option`, it means the result of the incorrect application was not used. In module `Success`, the same client code as module `Failure` uses module `Abstract` instead of module `Concrete` and the verification succeeds.

3 Functional Bulk Synchronous Parallelism

The OCaml language is a versatile programming language that combines functional, imperative and object-oriented paradigms. BSML [19] (Bulk Synchronous Parallel ML) is an OCaml-based library that embodies the principles of the BSP [25] (Bulk Synchronous Parallel) model. It provides a range of constants and functions to facilitate BSP programming. The BSP machine, viewed as a homogeneous distributed memory system with a point-to-point communication network and a global synchronization unit, serves as the underlying architecture for BSML. BSP programs, composed of consecutive super-steps, run on this kind of machine. The execution of each super-step follows a distinct pattern, starting with the computation phase where each processor-memory pair performs local computations using data available locally. This phase is followed by the communication phase, during which processors can request and exchange data with other processors. Finally, the synchronization phase concludes the super-step, synchronizing all processors globally.

With its collection of four expressive functions and constants like `bsp_p` representing the number of processors in the BSP machine, BSML empowers developers to create BSP algorithms. While OCaml supports imperative programming and BSML can exploit it [16], in this paper we only consider the pure functional aspects of OCaml and BSML. Indeed, the four BSML functions are higher-order functions but Why3 does not handle non-pure function arguments. This deliberate focus differentiates it from the imperative counterparts provided by libraries such BSPlib for C [14]. The types and informal semantics of BSML primitives are listed in Fig. 3.

Let us consider a function `f` that maps integers to values of type α (denoted as `f: int→`α in OCaml). The BSML primitive `mkpar f` produces a *parallel vector* of type α `par` when applied to function `f`. Within this parallel vector, each processor, identified by the index value `i` within the range $0 \leq i <$ `bsp_p` ,stores the computed value of `f i` . For instance, employing the expression `mkpar(fun i→i)` yields a parallel vector denoted as $\langle 0, \ldots,$ `bsp_p` $- 1\rangle$ of type `int par` . Throughout subsequent discussions, we shall refer to this parallel vector as `this`. Additionally, the function `replicate` possesses the type $\alpha \rightarrow \alpha$ `par` and can be defined as follows: `let replicate = fun x → mkpar(fun i → x)` . By employing the expression `replicate x,` the value `x` becomes uniformly available across all processors within the parallel vector. Parallel vectors always have size `bsp_p` .

To apply a parallel vector of functions (which is not a function) to a parallel vector of values, one has to use the primitive `apply` . Both `mkpar` and `apply`

$$\texttt{bsp\_p} : \texttt{int}$$
$$\texttt{bsp\_p} = p$$

$$\texttt{mkpar} : (\texttt{int} \to \alpha) \to \alpha\,\texttt{par}$$
$$\texttt{mkpar}\ f = \langle f\,0,\ \ldots,\ f\,(p-1)\rangle$$

$$\texttt{proj} : \alpha\,\texttt{par} \to (\texttt{int} \to \alpha)$$
$$\texttt{proj}\ \langle v_0,\ \ldots,\ v_{p-1}\rangle = \texttt{function}\,0 \to v_0 \mid \ldots \mid p-1 \to v_{p-1}$$

$$\texttt{apply} : (\alpha \to \beta)\texttt{par} \to \alpha\,\texttt{par} \to \beta\,\texttt{par}$$
$$\texttt{apply}\ \langle f_0,\ \ldots,\ f_{p-1}\rangle\ \langle v_0,\ \ldots,\ v_{p-1}\rangle = \langle f_0\ v_0,\ \ldots,\ f_{p-1}\ v_{p-1}\rangle$$

$$\texttt{put} : (\texttt{int} \to \alpha)\texttt{par} \to (\texttt{int} \to \alpha)\texttt{par}$$
$$\texttt{put}\ \langle tosend_0,\ \ldots,\ tosend_{p-1}\rangle = \langle received_0,\ \ldots,\ received_{p-1}\rangle$$
$$\text{where for all } src,\ dst$$
$$0 \leq src, dst < p \Rightarrow received_{dst}\ src = tosend_{src}\ dst$$

Fig. 3. BSML primitives

are executed within the pure computation phase of a super-step. For communications and an implicit synchronization barrier, the last two primitives `proj` and `put` should be applied. `proj` is essentially an inverse of `mkpar` but the resulting function is partial and only defined on the domain `[0, p-1]` . As the first constant constructor of any inductively defined type is considered as the empty message, `put` allows to program any communication pattern of a BSP super-step. In the input vector of `put` , each function encodes the message to be sent to other processors by the processor holding it. In the result vector, each function represents the message received from other processors by the processor holding the function.

Figure 4 presents a small BSML example using its primitives and `parfun` which is part of its standard library. `List.map` and `List.fold_left` are part of the OCaml standard library and are sequential map and reduce functions.

At lines 4–5, we define a function `list_of_par` which converts a parallel vector into a list. This function requires a full super-step for its execution because it needs data exchanges. Also part of the BSML standard library, `procs` has type `int list` and is the list `[0;...;bsp_p-1]` .

At lines 7–8, we define an algorithmic skeleton: a parallel map that operates on a distributed list (represented here as a value of type α `list par`). This function also requires the computation phase of a super-step and does not need any data exchange or synchronization.

From line 10 to 13, we define the `reduce` algorithmic skeleton, using a binary associative operation `op` and a neutral element `e`, it "sums" a distributed list into a single value. It proceeds in two steps. First, each processor compute a partial "sum" of the list it holds locally. Second, this vector of partial sums is transformed into a list which is finally summed up. As we call `list_of_par` , a full super-step is required.

Finally, in lines 15–18, we implement a parallel function to solve the maximum prefix sum problem. The goal is to compute the maximum value among

the sums of the prefixes of a list. Computing at the same time the maximum prefix sum and the sum of a list (in a pair) can be implemented using `map` and `reduce` . For example, on a machine with at least 4 processors, the value of `mps (mkpar(function|0→[1;2]|1→[-1;2]|2→[-1;3]|3→[-4]|_→[]))` is `6`. Indeed, the argument of `mps` is a distributed version (on 4 processors) of the list `[1;2;-1;2;-1;3;-4]` and its prefix with the largest sum is the list without its last element. We specify and prove the correctness of `mps` in Sect. 5.

```
1  open Bsml
2  open Stdlib.Base
3
4  let list_of_par (v: α par) : α list =
5    List.map (proj v) procs
6
7  let map : (α→β) → α list par → β list par =
8    fun f v → parfun (List.map f) v
9
10 let reduce : (α→α→α) →α → α list par → α =
11   fun op e v →
12   let locally_reduced =parfun (List.fold_left op e) v in
13   List.fold_left op e (list_of_par locally_reduced)
14
15 let mps : int list par → int =
16   let op (xm, xs) (ym, ys) =(max 0 (max xm (xs+ym)), xs+ys) in
17   let f x =(max 0 x, x) in
18   fun v → fst (reduce op (0, 0) (map f v))
```

Fig. 4. A BSML Example

4 Formalization of BSML Core and Standard Library

To be able to specify and write BSML programs, we need BSML primitives in WhyML. BSML primitives are implemented in parallel on top of MPI [23] called through OCaml's Foreign Function Interface (FFI). Therefore, we cannot provide BSML in WhyML as an implementation. We need to give a BSML *theory*: a set of constant, axioms and function declarations. The axiomatization of BSML primitives can be found in Fig. 5. The semantics of functions `mkpar`, `apply`, `proj` and `put` are expressed in their contract (lines 12–24) while the strict positivity condition on `bsp_p` is given as an axiom on line 4. The type of parallel vector is abstract. Still we need to be able to observe parallel vectors. That is the role of logic function `get` which is a `ghost` function: it can only be used in specifications. A parallel vector is fully defined by the values all the processors hold as expressed by the axiom `extensionality` in lines 9-10. The axiomatization is very close to the informal semantics of Fig. 3. Instead of

```
theory BSML
  use int.Int
  val constant bsp_p : int
  axiom at_least_one_processor : bsp_p > 0

  type par α
  val ghost function get (_ : par α) (_ : int) : α
  axiom extensionality:
    ∀ v v': par α.
    (∀ i: int. 0≤i<bsp_p →get v i = get v' i) →v = v'

  val mkpar (f : int →α ) : par α
    ensures { ∀ i:int. 0 ≤ i < bsp_p →get result i = f i }

  val apply (f : par (α →β)) (v : par α) : par β
    ensures { ∀ i:int. 0 ≤ i < bsp_p →
                       get result i = (get f i) (get v i) }

  val proj (v : par α) (x : int) : α
    ensures { result = get v x }

  val put (v : par (int →α)) : par (int →α)
    ensures { ∀ s d:int. 0 ≤ s < bsp_p →0 ≤ d < bsp_p →
                              (get result d) s = (get v s) d }
end
```

Fig. 5. BSML Theory in WhyML

considering the parallel vectors globally with the notation $\langle v_0, \ldots, v_{p-1} \rangle$, we consider each value v_i denoted by `get v i`.

It is possible to realize this theory by a sequential implementation, for example implementing parallel vectors with sequential lists or arrays. This ensures the consistency of this theory.

To illustrate the use of this theory, we now specify, implement and verify several of the functions provided in the BSML standard library. The first one is `replicate`:

```
let replicate (x: α) : par α
  ensures { ∀ i:int. 0 ≤ i < bsp_p → get result i = x }
= mkpar(fun _ → x)
```

This verified function has only one post-condition: the result of replication is parallel vector which contains the same value everywhere.

In Sect. 3, we mentioned the function `parfun` without defining it. Its implementation and specification follows, as well as the definition of function `parfun2`:

```
let parfun (f : α → β) (v: par α) : par β
  ensures { ∀ i:int. 0 ≤ i < bsp_p →
                     get result i = f (get v i) }
```

```
= apply (replicate f) v
let parfun2 (f : α → β → γ) (u : par α) (v : par β) : par γ
  ensures { ∀ i:int. 0 ≤ i < bsp_p →
                     get result i = f (get u i) (get v i) }
= apply (parfun f u) v
```

It shows how to use the `apply` primitive. There is also a `parfun3` function omitted here.

Next, we use the communication primitive `proj`. As we wrote in Sect. 3, `proj` is essentially the inverse of `mkpar`. This function allows us to obtain the value of a vector `v` at a given processor `i`. However, it should not be used for such individual vector access, otherwise the performances would be extremely poor. Indeed, a call to `proj` requires a communication phase that is a total exchange and a global synchronization barrier. The use of `proj` should thus be thought as a collective operation. Note that `proj` and `get` have the same semantics. However, the intent is very different: `get` is written only in specifications, can be thought as an indexed array access, and is used for *local* reasoning, while `proj` is used only in programs and requires a full super-step to execute. `proj` should rather be thought as a *global* (i.e. concerning and involving all the processors) conversion of a parallel vector into a function.

To illustrate `proj`, we define `list_of_par`. As we mentioned before this function requires a complete super-step to run. Again it should be seen as a *global* conversion from parallel vectors to lists:

```
let function list_of_par (v : par α) : list α
  ensures { ∀ i:int. 0 ≤ i < bsp_p → result[i] = get v i }
  ensures { length result = bsp_p }
= map (proj v) (procs())
```

As in the BSML/OCaml version we call `procs` – which needs to be a function for Why3 to accept the code. `procs` returns the list of all processor identifiers. The definition of `procs` relies on a function `from_to` itself implemented using a `init` function. Our contribution does also contain a library of sequential functions, mostly on lists, as well as verified lemmas stating their properties. These functions can in most cases be used both in programs and in specifications.

Finally, the `put` primitive is illustrated to implement a broadcast function. This data exchange (and implicit global synchronization) function is more precise than `proj`, in the sense that with `proj` each processor sends the same value to all processors, while with `put` each processor can send a different value to each destination processor. Also, as some values are considered as empty messages, this makes possible to reduce exchange costs. We remind the reader that after a `put`, for all processors `d` and `s`, the result function at destination processor `d`, applied to the identifier of source processor `s` returns the value of the input function at source processor `s` applied to destination processor `d`.

The definition of the `bcast_direct` function of the standard library follows. This function is used to broadcast a value from a `root` processor to all other processors. To do so, first, we prepare a function vector for the processors to make

the messages to send to each other (local definitions `make_msg` and `to_send`). It is clear that only the `root` processor will send data. The other processors' message is `None` which is interpreted by the BSML/OCaml implementation as an empty message. Second, the local definition `received` proceeds with the data exchange and ends the super-step. `received` is a parallel vector of functions. What we are interested in is the value sent by processor `root`. That is why the local definition `optional_result` then applies this parallel vector of functions to the replicated value `root`. Of course, the obtained message is encapsulated in a `Some` constructor. Therefore, all the processors finally apply `remove_option` to yield the final result. The broadcast is meaningless if `root` is not a valid processor identifier. In this case, the exception `Bcast` is raised:

```
let bcast_direct (root : int) (v : par α)
  ensures { 0 ≤ root < bsp_p →
            ∀ i:int. 0 ≤ i < bsp_p →
                     get result i = get v root }
  raises { Bcast }
= if (0 ≤ root) && (root < bsp_p )
  then
    let make_msg src load = fun _ →
        if src = root then Some load else None in
    let to_send  = apply (mkpar make_msg) v in
    let received = put to_send in
    let optional_result = apply received (replicate root) in
    parfun remove_option optional_result
  else raise Bcast
```

Our BSML theory allows us to write BSML programs and their specifications and is expressive enough for the Why3 framework to verify that they indeed satisfy their specifications.

We only presented a sub-set of the functions of the BSML standard library we implemented, and we refer to the companion artifact for the complete set of functions. For example, we also provide the `shift` , `shift_right` and `shift_left` communication functions, which offer a different communication pattern than `bcast_direct`: Each data item is shifted by a certain number of processors.

5 Verified Scalable Maximum Prefix Sum

To exercise the formalization presented in the previous section, we specify and verify an implementation of the maximum prefix sum informally presented in Sect. 3. As in the BSML implementation, the implementation with WhyML relies on algorithmic skeletons. The skeleton `par_map` is defined in lines 1–6 of Fig. 6. The only difference with its BSML/OCaml counterpart is the post-conditions including one expressed as a correspondence with the sequential `map`. Given a distributed list `dl` (of type `par(list α)`), one obtains the same result by either applying `map_par` then transforming the obtained distributed list into a list

```
1  let map_par (f: α→β) (dl: par (list α)) : par (list β)
2    ensures { ∀ i:int. 0 ≤ i < bsp_p →
3                       get result i = map f (get dl i) }
4    ensures { to_list result = map f (to_list dl) }
5  = let ghost _ = flatten_map f (list_of_par dl) in
6    parfun (map f) dl
7
8  let reduce_par (ghost inv: α→bool)
9    (op: α→α→α) (e: α) (dl: par(list α)) : α
10   requires { associative inv op }
11   requires { neutral inv op e }
12   requires { preserves inv op }
13   requires { inv e }
14   requires { ∀ i:int. 0 ≤ i < bsp_p →
15                       satisfies inv (get dl i) }
16   ensures { result = fold_left op e (to_list dl) }
17 = let ghost _ = fold_left_flatten inv op e (list_of_par dl) in
18   let reduce_seq l = fold_left op e l in
19   let partial_reductions = parfun reduce_seq dl in
20   reduce_seq (list_of_par partial_reductions)
```

Fig. 6. Verified Algorithmic Skeletons in WhyML

```
1   predicate associative (inv: α→bool) (op: α→α→α) =
2     ∀ x y z:α. inv x →inv y →inv z →
3       op x (op y z) = op (op x y) z
4
5   predicate neutral (inv: α→bool) (op: α→α→α) (e: α) =
6     (∀ x:α. inv x →op x e = x) ∧
7     (∀ x:α. inv x →op e x = x)
8
9   predicate preserves (inv: α→bool) (op: α→α→α) =
10    ∀ x y : α. inv x →inv y →inv (op x y)
11
12  predicate satisfies (inv: α→bool) (l: list α) =
13    ∀ i:int. 0 ≤ i < length l →inv (nth i l)
```

Fig. 7. Algebra Concepts

with `to_list`, or applying the sequential `map` to the sequentialization of the distributed list. Line 5 is just a hint for the provers: an application of lemma `flatten_map` that basically commutes `map` and `flatten`.

The implementation (lines 8-20) of the parallel reduction `reduce_par` is also very close to its BSML/OCaml counterpart of Fig. 4. As expected, the post-condition on line 16 is expressed with respect to the sequential reduction here implemented with the usual `fold_left` function. As the result is already a sequential value there is no need to sequentialize it. However, this correspondence is true only if `op` is associative and `e` is its neutral element which are

two pre-conditions stated lines 10–11. There are two additional pre-conditions and a ghost argument, i.e. an argument only used in the contract (and possible annotations) of the function. The reason is again to deal with a form of partial functions. `op` is a total function, but it may not have the desired properties (associativity, neutral element) on all the values of its input type. Indeed, the OCaml version of `op` for `mps` that we will also use in the WhyML version of `mps`, is not associative if we consider all pairs of integers. In the maximum prefix sum problem, the first component of such a pair represents the maximum prefix sum, it is therefore positive, and the second component the sum of the list, thus it is lower or equal to the first component. The ghost argument `inv` expresses such properties on the values manipulated during the reduction. This is an invariant: `op` should preserve the property (line 12) and the input values `e` and `dl` should satisfy this property (line 13). The predicates `associative`, `neutral`, `preserves` and `satisfies` are defined in Fig. 7. Such definitions work also well when there is no need for an invariant: in this case we simply use the constant boolean function always returning `true`.

With these skeletons, it is possible to implement a parallel function to compute the maximum prefix sum of a distributed list as we did in Sect. 3. First, we define a *specification* as an inefficient function but direct translation of the informal specification: the `mps_spec` function on lines 1–2 of Fig. 8. We also define `op` (lines 7–8) and `f` (line 10) which are the arguments to `map` and `reduce` as in the BSML/OCaml example of Fig. 4. This time they are not local definitions because we need to state and verify some lemmas about them and because we have two versions of `mps`: `mps_seq` and `map_par`. The invariant explained above is defined lines 12–13. We need an auxiliary function to verify the correctness of our functions with respect to the specification: `ms` (line 4–5) is the tupling of `mps_spec` and `sum`. The rest of the code in Fig. 8 is the definitions of the sequential and parallel versions of the maximum prefix sum computation. Both of them are expressed as a composition of map and reduce.

The proof that `mps_seq` indeed implements the specification `mps_spec` proceeds by using the first homomorphism theorem. This theorem states that a homomorphic function can be implemented as a composition of map and reduce. A function `f` is homomorphic when there exists a binary operation $\odot$ such that: $\forall$ `l1 l2: list` α`. f(l1++l2) = (f l1)` $\odot$ `(f l2)` where `++` denotes list concatenation. `mps_spec` is not homomorphic but `ms` is. Two lines of annotations are necessary to guide the provers in the sequential case (lines 17–18). The parallel case does need any annotation: basically the contracts of `map_par` and `reduce_par` state their correspondence with their sequential counterpart thus the correspondence of the parallel `mps_par` with the sequential `mps_seq`, and `mps_seq` satisfies `mps_spec`.

The full development is about 600 lines of WhyML with about 45% of specifications and 55% of code. It generates 74 goals, 100% of which are proved. Their verification produces 37 sub-goals. The strategy `Auto level 2` is used: it tries the provers CVC4, Alt-Ergo, CVC5 and Z3 with a short timeout (1 s). If the goal is not proved then it splits the goal and tries on the sub-goals with the

```
let function mps_spec (l: list int) : int
= maximum (map sum (prefix l))

function ms (l: list int) : (int,int)
= (mps_spec l, sum l)

let function op (x: (int,int)) (y: (int,int)) : (int, int)
= (max (max 0 (fst x)) ((snd x)+(fst y)), (snd x)+(snd y))

let function f (x : int) = (max 0 x, x)

predicate mps_sum_inv (x: (int,int))
= 0 ≤ fst x ∧ fst x ≥ snd x

let function mps_seq (l: list int) : int
  ensures { result = mps_spec l }
= let ghost _ = first_homomorphism_theorem ms op l in
  assert { fold_left op (0, 0) (map f l) = ms l };
  fst (fold_left op (0, 0) (map f l))

let mps_par (dl: par(list int)) : int
  ensures { result = mps_spec (to_list dl) }
= let mapped = map_par f dl in
  fst (reduce_par mps_sum_inv op (0, 0) mapped)
```

Fig. 8. Verified Maximum Prefix Sum in WhyML

same timeout and finally if necessary tries with a larger timeout (10 s). Alt-Ergo version 2.4.3 proved 11 goals taking between 0.02 s and 0.56 s (when successful) and CVC4 version 1.6 proved 91 goals taking between 0.04 s and 2.45 s. Several sub-goals can contribute to a goal to be proved. For example the verification condition of `mps_seq` is split in 3 sub-goals. In the number of the goals proved by CVC4 and Alt-Ergo the root goals verified because their sub-goals are proved are not counted. In our case, only 9 goals needed to be split to achieve their proofs.

All the parts of the WhyML development that need to use BSML functions include `use bsml.BSML`. When we extract the code however, the module `BSML` cannot be extracted: there is no implementation for this module. For compiling the OCaml code obtained by extraction of the other modules of our WhyML development, we simply use the handwritten implementation of BSML in OCaml (which uses OCaml's FFI to call MPI C functions). This is done via a very simple Why3 custom extraction driver: each BSML type or value is written using the OCaml qualified identifier notation. For example, if the WhyML development contains `mkpar` then the extracted code will contain `Bsml.mkpar` where `Bsml` is the module containing the handwritten BSML parallel library.

6 Related Work

BSP-WHY [10,11] also uses (a previous version of) Why to verify bulk synchronous parallel programs. However, the two approaches are very different. BSP-WHY considers BSP programs written in an imperative style close to BSPlib [14]. The verification proceeds by transforming well-formed programs — a sub-class of what has been formally defined later by Dabrowski as textually aligned programs [6] — into sequential simulating programs that are then verified using Why. The BSP-WHY code cannot be run on parallel machines.

The work closest to ours is the specification, verification and extraction of BSML programs using the Coq proof assistant. Early contributions started with the work of Gava [12]. A formalization of BSML primitives in a style very close to the Why3 formalization presented in this paper was proposed by Tesson and Loulergue [24] and used in a framework, named SyDPaCC, for the verification of BSP functional programs [8,20]. The two main differences with our work is that: (1) proofs are much less automated in Coq than in Why3 but (2) the framework leverages the type-class resolution mechanism of Coq to automatically parallelize programs. For example in this framework, the user does not need to write the code for `mps_seq` and `mps_par`, but only needs to write `mps_spec` and to prove that its tupling with `sum` is leftwards and rightwards (i.e. can be written as calls to `fold_left` and `fold_right`) and exhibits a weak right inverse. The framework would then use transformation theorems to automatically obtain `mps_seq` and then verified correspondences as expressed in the post-conditions of `map_par` and `reduce_par` to automatically produce `mps_par` [18].

Ono et al. [22] employed Coq to verify Hadoop MapReduce programs and extract Haskell code for Hadoop Streaming or directly write Java programs annotated with JML, utilizing Krakatoa [9] to generate Coq lemmas. The first part of their work is functional and therefore closest to our work. However, it is limited to MapReduce which is more general than the `map_par` and `reduce_par` skeletons but is less expressive than BSML. The second part of their work is more imperative.

7 Conclusion and Future Work

We were able to formalize the primitives of the parallel programming library BSML with WhyML and leverage Why3 for verifying a large part of the BSML standard library as well as an application written in BSML. We plan to experiment the extracted code more thoroughly and on larger parallel machines with a few thousand cores.

WhyML offers exceptions and references, thus allows to write imperative programs. However, such programs cannot be passed as arguments to higher-order functions. It therefore limits the usage of imperative features with BSML as all primitives are higher-order functions. The code outside BSML primitives can be imperative thus the sequencing of BSP super-steps could be imperative. It is also possible to use imperative features to implement pure functions passed as

arguments to BSML primitives. Also, it is possible to deal with partial functions as we did with `remove_some` . We plan to explore all these possibilities in the future.

Acknowledgment. The authors thank the anonymous reviewers for their helpful comments.

References

1. Armbrust, M., et al.: Scaling spark in the real world: performance and usability. PVLDB **8**(12), 1840–1851 (2015). https://www.vldb.org/pvldb/vol8/p1840-armbrust.pdf
2. Bobot, F., Filliâtre, J.C., Claude, M., Melquiond, G., Paskevich, A.: The Why3 platform (2023). https://why3.lri.fr
3. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Let's verify this with Why3. Int. J. Softw. Tools Technol. Transfer **17**(6), 709–727 (2014). https://doi.org/10.1007/s10009-014-0314-5
4. Cole, M.: Algorithmic skeletons: structured management of parallel computation. MIT Press (1989)
5. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories. Oxford, United Kingdom (2018). https://inria.hal.science/hal-01960203
6. Dabrowski, F.: A denotational semantics of textually aligned SPMD programs. J. Log. Algebraic Methods Program. **108**, 90–104 (2019). https://doi.org/10.1016/j.jlamp.2019.02.010
7. Denis, X., Jourdan, J., Marché, C.: CREUSOT: a foundry for the deductive verification of rust programs. In: Riesco, A., Zhang, M. (eds.) Formal Methods and Software Engineering. ICFEM 2022. LNCS, vol. 13478. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17244-1_6
8. Emoto, K., Loulergue, F., Tesson, J.: A verified generate-test-aggregate Coq library for parallel programs extraction. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 258–274. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_17
9. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_21
10. Fortin, J., Gava, F.: BSP-WHY: an intermediate language for deductive verification of BSP programs. In: 4th workshop on High-Level Parallel Programming and applications (HLPP), pp. 35–44. ACM (2010). https://doi.org/10.1145/1863482.1863491
11. Fortin, J., Gava, F.: BSP-Why: a tool for deductive verification of BSP algorithms with subgroup synchronisation. Int. J. Parallel Prog. **44**(3), 574–597 (2015). https://doi.org/10.1007/s10766-015-0360-y
12. Gava, F.: Formal proofs of functional BSP programs. Parall. Process. Lett. **13**(3), 365–376 (2003)
13. Gava, F., Garnier, I.: New implementation of a BSP composition primitive with application to the implementation of algorithmic skeletons. In: 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009), APDCM workshop, pp. 1–8. IEEE (2009). https://doi.org/10.1109/IPDPS.2009.5160876

14. Hill, J.M.D., et al.: BSPlib: the BSP programming library. Parallel Comput. **24**, 1947–1980 (1998)
15. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
16. Loulergue, F.: A BSPlib-style API for bulk synchronous parallel ML. Scalable Comput.: Pract. Exp. **18**, 261–274 (2017). https://doi.org/10.12694/scpe.v18i3.1306
17. Loulergue, F.: Implementing algorithmic skeletons with bulk synchronous parallel ML. In: Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 461–468. IEEE (2017). https://doi.org/10.1109/PDCAT.2017.00079
18. Loulergue, F., Bousdira, W., Tesson, J.: Calculating parallel programs in Coq using list homomorphisms. Int. J. Parallel Prog. **45**(2), 300–319 (2016). https://doi.org/10.1007/s10766-016-0415-8
19. Loulergue, F., Gava, F., Billiet, D.: Bulk synchronous parallel ML: modular implementation and performance prediction. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2005. LNCS, vol. 3515, pp. 1046–1054. Springer, Heidelberg (2005). https://doi.org/10.1007/11428848_132
20. 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). https://doi.org/10.1145/2554850.2554912
21. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
22. Ono, K., Hirai, Y., Tanabe, Y., Noda, N., Hagiya, M.: Using Coq in specification and program extraction of Hadoop MapReduce applications. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 350–365. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24690-6_24
23. Snir, M., Gropp, W.: MPI the complete reference. MIT Press (1998)
24. 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). https://doi.org/10.1016/j.procs.2011.04.005
25. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103 (1990). https://doi.org/10.1145/79173.79181
26. White, T.: Hadoop - The Definitive Guide. O'Reilly, 2nd edn. (2010)