A Verified Generate-Test-Aggregate Coq Library for Parallel Programs Extraction

Kento Emoto[1], Frédéric Loulergue[2], and Julien Tesson[3]

[1] Kyushu Institute of Technology, Japan, emoto@ai.kyutech.ac.jp
[2] Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, France, Frederic.Loulergue@univ-orleans.fr
[3] Université Paris Est, LACL, UPEC, France, Julien.Tesson@univ-paris-est.fr

Abstract. The integration of the generate-and-test paradigm and semi-rings for the aggregation of results provides a parallel programming framework for large scale data-intensive applications. The so-called GTA framework allows a user to define an inefficient specification of his/her problem as a composition of a generator of all the candidate solutions, a tester of valid solutions, and an aggregator to combine the solutions. Through two calculation theorems a GTA specification is transformed into a divide-and-conquer efficient program that can be implemented in parallel. In this paper we present a verified implementation of this framework in the Coq proof assistant: efficient bulk synchronous parallel functional programs can be extracted from naive GTA specifications. We show how to apply this framework on an example, including performance experiments on parallel machines.
Keywords: List homomorphism, functional programming, automatic program calculation, semi-ring computation, bulk synchronous parallelism, Coq

1 Introduction

Nowadays parallel architectures are everywhere. However parallel programming is still reserved to experienced programmers. There is an urgent need of programming abstractions, programming methodologies as well as support for the verification of parallel applications, in particular for distributed memory models. Our goal is to provide a framework to *ease* the *systematic* development of *correct* parallel programs. We are particularly interested in large scale data-intensive applications.

Such a framework should provide programming building blocks whose semantics is easy to understand for users. These building blocks should come with an equational theory that allows to transform programs towards more efficient versions. These more efficient versions should be parallelisable in a transparent way for the user.

GTA (generate-test-aggregate) [4,3] provides such a framework, integrating the generate-and-test paradigm and semi-rings for the aggregation of results. Through two calculation theorems a GTA specification is transformed into an

efficient program. It can then be implemented in parallel as a composition of algorithmic skeletons [1] which can be seen as higher-order functions implemented in parallel.

In this paper we present a verified implementation of the framework in the Coq proof assistant. The contributions of this paper are:

- the formalisation of the GTA paradigm in the Coq proof assistant,
- the proof of correctness of the program transformations,
- an application of the framework to produce a parallel program for the knapsack problem and experiments with its execution on parallel machines.

We first present the generate-test-aggregate paradigm (Section 2), and its formalisation in Coq including the proof of the calculation theorems (Section 3). We then explain how to obtain a parallel code from the result of a program calculation using Coq extraction mechanism (Section 4). In Section 5 we describe experiments performed on parallel machines. Comparison with related work (Section 6) and conclusion (Section 7) follow.

2 GTA: Generate, Test and Aggregate

We briefly review the Generate-Test-and-Aggregate (GTA) paradigm [4,3]. It has been proposed as an algorithmic way to synthesise efficient programs from naive specifications (executable naive programs) in the following GTA form.

$$GTAspec = aggregate \circ test \circ generate$$

In this section we first give a simple example of how to specify naive algorithms in the GTA form. We give a clear but inefficient specification of the knapsack problem following this structure.

The knapsack problem is to fill a knapsack with items, each of certain non-negative value and weight, such that the total value of packed items is maximal while adhering to a weight restriction of the knapsack. For example, if the maximum total weight of our knapsack is 3kg and there are three items ($10, 1kg), ($20, 2kg), and ($30, 2kg) then the best we can do is to pick the selection ($10, 1kg), ($30, 2kg) with total value $40 and weight 3kg because all selections with larger value exceed the weight restriction.

The function *knapsack*, which takes as input a list of value-weight pairs (both positive integers) and computes the maximum total value of a selection of items not heavier than a total weight W, can be written in the GTA form:

$$knapsack\ W = maxvalue \circ validWeight\ W \circ subs$$

The function *subs* is the generator. From the given list of pairs it computes all possible selections of items, that is, all 2^n sublists if the input list has length n. The function $validWeight\ W = \mathsf{filter_B}((<= W) \circ weight)$ is the tester. It discards all generated sublists whose total weight exceeds W and keeps the rest. The

function *maxvalue* is the aggregator. From the remaining sublists adhering to the weight restriction it computes the maximum of all total values.

The function *subs* can be defined as follows:

$$subs = \mathsf{fold\_right}\ (\lambda\ x\ s.(\wr[]\int \uplus \wr[x]\int) \times_{+\!+} s)\ (\wr[]\int)$$

The result of the generator *subs* is a bag of lists which we denote using $\wr$ and $\int$. The symbol $\uplus$ denotes bag union, e.g., $\wr[], [x]\int = \wr[]\int \uplus \wr[x]\int$, and $\times_{+\!+}$ the lifting of list concatenation to bags, concatenating every list in one bag with every list in the other. Here is an example application of *subs*: $subs\ [1,3,3] = \wr[], [1]\int \times_{+\!+} \wr[], [3]\int \times_{+\!+} \wr[], [3]\int = \wr[], [1], [1,3], [1,3], [1,3,3], [3], [3], [3,3]\int$. We took the liberty to reorder bag elements lexicographically because bags are unordered collections. Note that elements may occur more than once as witnessed by $[1,3]$.

The function *validWeight* is filter operation $\mathsf{filter_B}$ with a predicate to check the weight restriction in which we use $weight = \mathsf{fold\_right}\ (\lambda(v,w)s.w+s)\ 0$ to find the total weight of a given item list:

$$validWeight = \mathsf{filter_B}\ ((<= w) \circ weight)$$

Here, it is easily seen that *weight* satisfies the follwoing three equations:

$$\begin{aligned} weight\ [] &= 0 \\ weight\ [(v,w)] &= w \\ weight\ (xs +\!+ ys) &= weight\ xs + weight\ ys \end{aligned}$$

It simply replaces list constructors [] and $+\!+$ with 0 and $+$. A function satisfying three equations of this form is called monoid homomorphism. Note that a monoid is a mathematical structure made of an associative binary operator with its identity element: The constructors and operators in the equations describe monoids.

Finally, the aggregator *maxvalue* computes the maximum of summing up the values of each list in a bag using the maximum operator $\uparrow$, which is defined as a recursive function (natural fold operation) on bags:

$$\begin{aligned} maxvalue\ \wr\int &= -\infty \\ maxvalue\ \wr l\int &= \mathsf{fold\_right}\ (\lambda(v,w)s.v+s)\ 0\ l \\ maxvalue\ (b \uplus b') &= maxvalue\ b \uparrow maxvalue\ b' \end{aligned}$$

It is easily seen that *maxvalue* satisfies the following five equations; it simply replaces constructors $\uplus$, $\times_{+\!+}$, $\wr[]\int$ and $\wr\int$ with $\uparrow$, $+$, 0 and $-\infty$, respectively. A function satisfying five equations of this form is called semiring homomorphism. Note that a semiring is a mathematical structure of a monoid; a commutative monoid and a distributivity law over the operators of those monoids. The constructors and operators in the following equations form a semiring.

$$\begin{aligned} maxvalue\ \wr\int &= -\infty \\ maxvalue\ \wr[]\int &= 0 \\ maxvalue\ \wr[(v,w)]\int &= v \\ maxvalue\ (b \uplus b') &= maxvalue\ b \uparrow maxvalue\ b' \\ maxvalue\ (b \times_{+\!+} b') &= maxvalue\ b + maxvalue\ b' \end{aligned}$$

Now, we have defined a naive program, i.e., a GTA specification of the problem.

Next, let us consider an efficient algorithm $knapsack'$ to solve the problem. The GTA provides two theorems to derive mechanically such an efficient program from a GTA specification, but we start with the derived efficient program to understand what the theorems do. We will see the theorems later.

We may use linear dynamic programming of the following form that repeatedly updates a map denoted by using { and }, in which each entry $w \mapsto v$ means that there is a selection of items with the best total value v with the total weight w:

$$knapsack'\ W = \pi \circ \mathsf{fold\_right}\ (\lambda(v,w)m.(\{0 \mapsto 0\} \oplus_W \{w \mapsto v\}) \otimes_W m)\ \{0 \mapsto 0\}$$

Here, $\oplus_W$ merges two maps by taking the maximum value of two entries of the same weight, while $\otimes_W$ makes every possible combination of entries of two maps and merges the results by $\oplus_W$. The post-process function π extracts the final result from the final map.

For example, we have $knapsack'\ 3\ [] = \pi(\{0 \mapsto 0\}) = 0$. Here, the map $\{0 \mapsto 0\}$ means that we have one selection with the best total value 0 and the total weight 0 since we have no item, and π extracts the only value 0 as the final answer. Similarly, we have $knapsack'\ 3\ [(30,2)] = \pi(\{0 \mapsto 0, 2 \mapsto 30\}) = 30$. The map $\{0 \mapsto 0, 2 \mapsto 30\}$ represents the selection possibilities about the item $(30,2)$: either it is selected (entry $2 \mapsto 30$ that results in the best value 30 with total weight 2), or not ($0 \mapsto 0$). The work of $\otimes_3$ can be seen clearly in the following sub-computation in $knapsack'\ 3\ [(10,1),(20,2),(30,2)] = \pi(\{0 \mapsto 0, 1 \mapsto 10, 2 \mapsto 30, 3 \mapsto 40, \_ \mapsto 60\}) = 40$. Here, $\_$ represents entries with "more than 3" to be ignored by π, and thus a map has at most 5 entries.

$$\begin{aligned}
&\{0 \mapsto 0, 1 \mapsto 10\} \otimes_3 (\{0 \mapsto 0, 2 \mapsto 20\} \otimes_3 \{0 \mapsto 0, 2 \mapsto 30\}) \\
&= \{0 \mapsto 0, 1 \mapsto 10\} \otimes_3 (\{(0+0) \mapsto (0+0)\} \oplus_3 \{(2+0) \mapsto (20+0)\} \\
&\qquad\qquad \oplus_3 \{(0+2) \mapsto (0+30)\} \oplus_3 \{(2+2) \mapsto (20+30))\} \\
&= \{0 \mapsto 0, 1 \mapsto 10\} \otimes_3 (\{0 \mapsto 0\} \oplus_3 \{2 \mapsto 20\} \oplus_3 \{2 \mapsto 30\} \oplus_3 \{_ \mapsto 50\}) \\
&= \{0 \mapsto 0, 1 \mapsto 10\} \otimes_3 \{0 \mapsto 0, 2 \mapsto 30, _ \mapsto 50\} \\
&= \{(0{+}0) \mapsto (0{+}0)\} \oplus_3 \{(1{+}0) \mapsto (10{+}0)\} \oplus_3 \{(0{+}2) \mapsto (0{+}30)\} \oplus_3 \cdots \\
&= \{0 \mapsto 0, 1 \mapsto 10, 2 \mapsto 30, 3 \mapsto 40, _ \mapsto 60\}
\end{aligned}$$

It should be noted that $\mathsf{fold\_right}$ used in $knapsack'$ is parallelisable because of the associativity of the operator $\otimes_W$.

We have got two programs $knapsack$ and $knapsack'$ to solve the knapsack problem. What is the relationship between these two? Comparing $subs$ and $knapsack'$ we can find that both $subs$ and $knapsack'$ can be written with a new function $poly\_subs\ f\ (\oplus)\ (\otimes)\ \imath_\oplus\ \imath_\otimes = \mathsf{fold\_right}\ (\lambda xm.(\imath_\otimes \oplus f\ x) \otimes m)\ \imath_\otimes$:

$$\begin{aligned}
subs &= poly_subs\ (\lambda x.\wr[x]\int)\ (\uplus)\ (\times_{+\!+})\ (\wr\int)\ (\wr[]\int) \\
knapsack'\ W &= \pi \circ poly_subs\ (\lambda(v,w).\{w \mapsto v\})\ (\oplus_W)\ (\otimes_W)\ (\{\})\ (\{0 \mapsto 0\})
\end{aligned}$$

Here, $poly\_subs$ is polymorphic over the result type of semiring operators $\oplus$ and $\otimes$, and $subs$ is an instantiation of this polymorphic function with the constructors $\uplus$ and $\times_{+\!+}$. Such a generator is called a polymorphic semiring generator.

Now, we are ready to see the fusion theorems for mechanical derivation of $knapsack'$ from $knapsack$. The GTA provides two theorems, namely, semiring-fusion and filter-embedding. The filter-embedding gives a way to derive the operators into $\oplus_W$ and $\otimes_W$ on maps, while the semiring-fusion gives a way to substitute these operators into the generator.

Theorem 1 (Filter Embedding [4]). *Given a monoid homomorphism mhom, a semiring homomorphism agg, and a function ok, there exist a function π and a semiring homomorphism agg' and the following equation holds:*

$$agg \circ \mathsf{filter_B}\ (ok \circ mhom) = \pi \circ agg'$$

Theorem 2 (Semiring Fusion [4]). *Given a semiring homomorphism agg, which replaces the constructors with f, $\oplus$, $\otimes$, $\imath_\oplus$, and $\imath_\otimes$, and a polymorphic semiring generator gen, the following equation holds:*

$$agg \circ gen\ (\lambda x \to \wr[x]\int)\ (\uplus)\ (\times_{+\!\!+})\ (\wr\int)\ (\wr[]\int) = gen\ f\ (\oplus)\ (\otimes)\ \imath_\oplus\ \imath_\otimes$$

These two theorems derive $knapsack'$ from $knapsack$ as follows. Here, $maxvalue'$ is a semiring homomorphism that replaces the constructors with $\lambda(v, w).\{w \mapsto v\}$, $\oplus_W$, $\otimes_W$, $\{\}$ and $\{0 \mapsto 0\}$, and is mechanically derived from $maxvalue$ and $validWeight\ W$ by the filter-embedding.

$$\begin{aligned}
knapsack\ W &= maxvalue \circ validWeight\ W \circ subs \\
&= \{\quad \text{Filter-embedding} \quad\} \\
&\quad\ maxvalue' \circ subs \\
&= \{\quad \text{Semiring-fusion} \quad\} \\
&\quad\ \pi \circ poly_subs\ (\lambda(v, w).\{w \mapsto v\})\ (\oplus_W)\ (\otimes_W)\ (\{\})\ (\{0 \mapsto 0\}) \\
&= knapsack'\ W
\end{aligned}$$

It should be noted that by using GTA one can easily develop an efficient parallel programs to solve many variants of problems such as a statistics probelm to find a most likely sequence of hidden events [7], a combinatorial problem to find the best period (contiguous subsequence) in a time series [2], and so on [3], by simply defining testers to specify variant problems with additional conditions.

3 Verified GTA Library

In this section we introduce our verified GTA library with an automatic fusion mechanism that allows a user to get an efficient Coq code freely from his/her naive Coq code in the GTA form. The library mainly consists of three parts: user interface, proof of the fusion theorems, and the automatic fusion mechanism.

3.1 User Interface: Writing Your Naive Code

This part defines a variety of items used to define a GTA specification as a Coq program, which includes axiomatization of the bag (i.e., multi-set) data structure and mathematical properties of components in a GTA specification.

Bag Axiomatisation. Since in a GTA specification a generator produces a bag of lists, we need a bag data structure in Coq to define a GTA specification. We axiomatised the bag data structure as a module type, and implemented a module of this type by using the list data structure as its underlying structure. A bag module has three constructors: `empty` for an empty bag, `singleton` to make a singleton bag of the given element, and `union` ($\uplus$ in the mathematical notation) to merge two bags. It is also equipped with a decidable equivalence relation, under which the usual semantics of bags (multi-sets) holds, and the natural fold operation `homB` respecting the equivalence relation. Interested readers may refer to the source code [15]. We also defined a module that, using the constructors and fold operation, implements computations on bags, such as map, filter, operator $\times_{+\!+}$ (`mapB`, `filterB`, and `cross` in Coq code), and function `singleBag` to make a bag of a singleton list of the given element. In addition, we showed properties of these operators, such as the semiring properties of the operators `union` and `cross` with their identities `empty` and `nilBag` (a bag of nil).

Generators. The first component of a GTA specification is a generator that produces a bag of lists and has to be an instance with $\uplus$ and $\times_{+\!+}$ of a polymorphic function over the result type of semiring operators $\oplus$ and $\otimes$. Figure 1 shows Coq code related to generators. The polymorphism condition for each generator is captured by an instance of typeclass `isSemiringPolymorphicGenerator`, which connects a generator `gen` and its polymorphic function `pgen`. The typeclass also has a field to show fusable property based on the polymorphism. We omit the details here, but the concept is that a polymorphic function determines computation structure independent of given arguments. This could be shown by the free theorem [18] about Coq, but Coq cannot prove such a property about himself. Thus, the library asks a user to show the property by hand. We may provide tactics to support this part because such a proof can be systematic. It is planed in our future work.

For example, the generator `subs` in Section 2 for the knapsack problem can be defined as an instance of the polymorphic function `poly_subs` as follows. We can show its fusable condition by a simple induction.

```
Fixpoint poly_subs
         (V:Type) (f:T→V) (oplus otimes:V→V→V) (ep et :V) (l:list T) :=
  match l with
   | nil ⇒ et
   | a::l' ⇒ otimes (oplus (f a) et) (poly_subs f oplus otimes ep et l')
  end.
Definition subs := poly_subs singleBag union cross empty nilBag.
Global Program Instance subs_is_polymorphic_generator
  : isSemiringPolymorphicGenerator subs (@poly_subs).
Next Obligation.
  apply Build_isSemiringPolymorphicFunction.
  induction l.
  - unfold poly_subs. simpl. reflexivity.
  - simpl. intros. rewrite IHl. reflexivity.
Defined.
```

```
Definition semiringPolymorphicType (B : Type) : Type
  := ∀{V:Type} (f : T →V) (oplus : V→V→V) (otimes : V→V→V) (ep et : V), list B →V.

Class isSemiringPolymorphicGenerator `{A : Type}
 (gen : list A →bag (list T)) (pgen : semiringPolymorphicType A) := {
  SemiringGenEquiv : ∀l, gen l = pgen (bag (list T)) singleBag union cross empty nilBag l;
  isSemiringPolymorphicGenerator_Polymorphism : (* snip *)
}.
```

Fig. 1. Formalization of polymorphic semiring generators.

Testers. The second component is a tester to discard invalid lists in the bag produced by a generator. Figure 2 shows Coq code related to testers. To be successfully fused by the theorems, a tester has to be a filter and its predicate is a composition (denoted by `:o:` in the code) of a simple decidable predicate and a monoid homomorphism. These conditions are straightforwardly captured by typeclasses `isHomomorphicFilter` and `isMonoidHomomorphism`.

For example, the tester `validWeight` in Section 2 for the knapsack problem can be defined by using the filter `filterB` on bags as follows. Here, decidability of its predicate (named `weightOk`) is also defined to be used by `filterB`.

```
Inductive Item := item : nat →nat →Item.
Definition getVal i := match i with item v _ ⇒ v end.
Definition getWeight i := match i with item _ w ⇒ w end.
Definition atmost (w:nat) := fun a ⇒ a <= w.
Definition weight (l:list Item):= fold_right (fun a w⇒ getWeight a+w) 0 l.
Definition weightOk (w:nat) := atmost w :o: weight.
Lemma atmost_dec (w:nat): ∀(a:nat), {atmost w a}+{∼atmost w a}.(* snip *)
Lemma weightOk_dec (w:nat):
                ∀(l:list Item), {weightOk w l}+{∼weightOk w l}.(* snip *)
Definition validWeight (w:nat) := filterB (weightOk w) (weightOk_dec w).
```

For successful fusion we also need an instance of `isMonoidHomomorphism` as well as the `Proper` instance of `atmost`, while we do not need an instance of `isHomomorphicFilter` because `validWeight` is directly defined by `fitlerB`:

```
Program Instance proper_atmost: Proper (eq⟹ eq⟹ iff) atmost.(* snip *)
Program Instance weightOk_monoidHom:
  isMonoidHomomorphism (T:=Item) weight getWeight plus 0. (* snip *)
```

Note that for the performance of the final fused program it is better to finitise the domain of the monoid homomorphism (e.g., to use the finite set $\{n : \mathbb{N} \mid n \leq w\}$ as the domain) before applying fusions. This can be done by hand or even by an automatic mechanism in some cases. We omit this here for the space limitation, but the experiment was conducted on the finitised version. Interested readers may refer to papers [4,3] and the source code [15].

Aggregators. The final component is an aggregator to make a summary using semiring operators from lists passing testers. Figure 3 shows Coq code related to aggregators. To be fused by the semiring-fusion, an aggregator has to be a semiring homomorphism. This is captured by typeclass `isSemiringHomomorphism`

```
Context `{eqDecT : @EqDec T eqT equivT} `{eqDecM : @EqDec M eqM equivM}.
 (* Monoid is a class to hold a monoid operator with its identity *)
Class isMonoidHomomorphism (h : list T →M) (f : T →M) (oplus : M →M →M) (e : M) := {
 isMH_Monoid : Monoid oplus e;
 isMH_CondAppend : ∀x y, h (app x y) === oplus (h x) (h y);
 isMH_CondSingle: ∀a, h (a::nil) === f a;
 isMH_CondNil: h nil === e;
 isMH_proper_oplus: Proper (eqM ⟹ eqM ⟹ eqM) oplus;
 isMH_proper_f: Proper (eqT ⟹ eqM) f
}.
 (* "dec_comp f g dec_f" derives decidability of (f :o: g) from that of f, namely, dec_f . *)
Class isHomomorphicFilter (tes : bag (list T) →bag (list T)) (mhom : list T →M) (h : T →M)
 (odot : M →M →M) (e : M) (ok : M →Prop) (dec : ∀m : M, {ok m} + {∼ ok m}) := {
 isHF_isMonoidHomomorphism: isMonoidHomomorphism (eqT:=eqT) mhom h odot e;
 isHF_spec: ∀b:bag(list T),tes b === filterB(ok:o:mhom)(dec_comp mhom ok dec)b;
 isHF_proper_ok: Proper (eqM ⟹ iff) ok
}.
```

Fig. 2. Formalization of monoid homomorphisms and homomorphic filters.

saying that the given function `agg` is a semiring homomorphism made of the function f, the operators `oplus` and `otimes` with `ep` as zero and `et` as one. The library provides a simple way to make a semiring homomorphism: Given semiring operators, nested fold operation `semiringHom` is the semiring homomorphism.

For example, the aggregator `maxvalue` in Section 2 to find the maximum sum value can be defined by using the fold operation `semiringHom` with predefined semiring `semiring_max'_plus'` of the max and plus operators with the minus infinity (i.e., the zero of the semiring):

```
Definition maxvalue
        := semiringHom (fun a:Item⇒ Num(getVal a)) semiring_max'_plus'.
```

Now, we have got a GTA specification as a naive Coq program, and we can check its correctness by running it with a small example:

```
Definition knapsack (w : nat) := maxvalue :o: validWeight w :o: subs.
Definition items := [item 10 1; item 20 2; item 30 2 ].
Eval compute in (knapsack 3 items).
(* = Num 40 : nat_minf *)
```

This is basically all that a user has to do in the GTA paradigm. The rest is to call an interface function (a field of a specific class) to trigger automatic fusion, which is shown later.

3.2 The Core: Proof of the GTA Fusion Theorems

The core of the library is proof of two fusion theorems, namely, the semiring-fusion and filter-embedding (Theorems 2 and 1). These fusion theorems give mechanical rules to transform a GTA specification (naive program) into an efficient program, whose automatic mechanism will be shown later.

The filter-embedding fusion transforms a composition of a tester and an aggregator into a new aggregator followed by a simple projection function. This eliminates the tester from a GTA specification. Since the new aggregator uses a

```
Context `{S : Type} `{eqDecS : @EqDec S eqS equivS} `{eqDecT : @EqDec T eqT equivT}.
(* eqBag is the equivalence relation given in a Bag module, taking an equivalence relation on elements.*)
(* Semiring is a typeclass to hold semiring operators with identities. *)
Class isSemiringHomomorphism (agg:bag(list T)→S)(f:T→S)(oplus otimes:S→S→S)(ep et:S):={
  isSH_Semiring : Semiring oplus otimes ep et;
  isSH_CondUnion : ∀x y, agg (union x y) === oplus (agg x) (agg y);
  isSH_CondCross : ∀x y, agg (cross x y) === otimes (agg x) (agg y);
  isSH_CondSingle : ∀a, agg (singleBag a) === f a;
  isSH_CondEmpty : agg (empty) === ep;
  isSH_CondNil : agg (nilBag) === et;
  isSH_proper_f : Proper (eqT ==> eqS) f;
  isSH_proper_oplus : Proper (eqS ==> eqS ==> eqS) oplus;
  isSH_proper_otimes : Proper (eqS ==> eqS ==> eqS) otimes;
  isSH_proper : Proper (eqBag (list T) ==> eqS) agg
}.

 (* short−hand to make a semiring homomorphism, i.e., nested fold operations *)
Definition semiringHom (f : T →S) `(semiring : Semiring oplus otimes ep et)
  := homB oplus (fold_right (fun a r ⇒ otimes (f a) r) et) ep.

Global Program Instance semiringHom_is_semiringHomomorphism
  `{semiring : Semiring oplus otimes ep et}
  `{proper_oplus : Proper (eqS ==> eqS ==> eqS) oplus }
  `{proper_otimes : Proper (eqS ==> eqS ==> eqS) otimes }
  `{proper_f : Proper (eqT ==> eqS) f}
  : isSemiringHomomorphism (semiringHom f oplus otimes ep et semiring) f oplus otimes ep et.
```

Fig. 3. Formalization of semiring homomorphisms.

semiring (so-called *monoid semiring* [12]) on finite maps, we need formalization of finite maps to formalise this theorem. Coq's standard library has formalization of maps, but it requires a module for each key type to show its decidability, which prevents us from an automatic optimisation mechanism that may change the key type. Therefore, we reformalised maps as a module type, in which the decidability of keys is given as an instance of a specific typeclass while functions and properties are the same as the standard library except for additional induction principles. The library also provides a list-based implementation of the module type. Interested readers may refer to the source code [15].

On top of the map formalisation we proved properties of the semiring on maps, which is the most crucial and difficult part of proving the filter-embedding. We also introduced disjoint-sum version of maps whose semiring properties are easily shown, and used an equivalence correspondence between the original and disjoint maps to prove the difficult part clearly. Interested readers may find a formalisation of the disjoint-sum maps in paper [12].

Once the properties of the semiring are shown, the filter-embedding theorem can be shown straightforwardly as the previous papers [4,3] did. The proof of semiring fusion is also straightforward once we are given an instance of `isSemiringPolymorphicGenerator`. Figure 4 shows the theorems proved in Coq, in which `postproc` is a simple projection function to extract the final result from a map and `monoid_semiring_of` is the semiring on maps built from the semiring in the aggregator `agg` and the monoid in the tester `tes`.

```
Context `{S : Type} `{eqDecS : @EqDec S eqS equivS}.

Theorem filterEmbeddingFusion
   `(shomAgg : isSemiringHomomorphism agg f oplus otimes ep et)
   `(homFilter : isHomomorphicFilter tes mhom h odot e ok dec)
 :∀ l,(agg:o:tes) l === (postproc ok dec ep oplus :o:
                         semiringHom(embed oplus h f)(monoid_semiring_of shomAgg homFilter))l.
Theorem semiringFusion
  `(polyGen : isSemiringPolymorphicGenerator A gen pgen)
  `(shomAgg : isSemiringHomomorphism agg f oplus otimes ep et)
  : ∀l, (agg :o: gen) l === (pgen S f oplus otimes ep et) l.
```

Fig. 4. Two fusion theorems of the GTA.

3.3 Automatic Fusion Mechanism

To allow a user to get an efficient Coq code freely from his/her naive GTA specification, the library implements an automatic fusion mechanism based on the typeclass mechanism. He/she can get efficient code by calling the function `fuse` on his/her specification as follows.

```
Definition knapsack' (w : nat) := fused (f := knapsack w).
```

The automatic fusion mechanism is implemented by instances of two classes `Fusion` and `Fuser` shown in Fig. 5. Instances of `Fusion` form knowledge database of fusion, i.e., a set of function triples $(consumer, producer, fused)$ such that $consumer \circ producer$ is equivalent to $fused$, while the instance `fuser` of `Fuser` triggers a search to find an instance of `Fusion` that gives the result $fused$ of fusing the given function composition $f = consumer \circ producer$. Figure 5 shows some of these instances, including fusion knowledge of the theorems in Fig. 4.

For example, the above call of `fused` on `knapsack` eventually finds the instance `comp_l_fuser`. It first calls `fused` on `maxvalue :o: validWeight` to get their fused result, and then calls another `fused` on the composition of the result and `subs`. The first call finds the instance `filterEmbeddingFusionInstance` to fuse `maxvalue` and `validWeight`, and returns a composition of `postproc` with the new aggregator, namely, `semiringHom` with the semiring operators on maps. The latter call finds the instance `semiringFusionInstanceWithPP` to get the final fused result equivalent to the efficient program shown in Section 2. It is worth noting that the mechanism works for multiple testers in a GTA specification, although it may take longer time to finish the fusion.

4 Extraction of Certified Efficient Parallel Code

The result of the fusion mechanism described in the previous section, is an instance of `poly_subs`. This function could actually be proven equivalent to a composition of a `map` and a `reduce` on lists. These two combinators could themselves be proven to correspond (in a sense we will describe later) to parallel implementations of `map` and `reduce` on *distributed* lists. In this way we obtain a parallel version of `poly_subs` and therefore of the `knapsack'` function. In order to do so, we need to be able to write (data) parallel programs in Coq.

```
Class Fusion `{eqDec : EqDec D} `(producer : B →C) `(consumer : C →D) (fused : B →D)
    := { fusion_spec : ∀b, consumer (producer b) === fused b }.

Class Fuser `{eqDec : EqDec D} `(f : B →D) := { fused : B →D; fuser_spec : ∀
b, f b === fused b }.

 (* An instance of Fuser to trigger a search for Fusion instances *)
Global Program Instance fuser `{fusion : Fusion producer consumer _fused}
  : Fuser (consumer :o: producer) := { fused := _fused }.

 (* A fuser for multiple compositions: fused ( (f . g) . h ) = fused ( fused (f . g ) . h) *)
Global Program Instance comp_l_fuser
  `{fusion_gf : @Fusion D R equiv0 eqDec B C g f fg}
  `{fusion_hgf : @Fusion D R equiv0 eqDec A B h fg fgh}
  : Fusion h (f :o: g) fgh. (* snip proof. *)

Global Program Instance filterEmbeddingFusionInstance (* Knowledge of the filter−embedding *)
  `{shomAgg : isSemiringHomomorphism agg f oplus otimes ep et}
  `{homFilter : isHomomorphicFilter tes mhom h odot e ok dec}
  : Fusion (bag (list T)) (bag (list T)) tes agg
   ((postproc ok dec ep oplus)
    :o: (semiringHom (embed oplus h f) (monoid_semiring_of shomAgg homFilter))).

Global Program Instance semiringFusionInstanceWithPP (* A variant of the semiring−fusion *)
  `{polyGen : isSemiringPolymorphicGenerator A gen pgen}
  `{shomAgg : isSemiringHomomorphism agg f oplus otimes ep et}
  `{pp : S →X} (* projection function after an aggregator *)
  : Fusion gen (pp :o: agg) (pp :o: pgen S f oplus otimes ep et).
```

Fig. 5. Instances for automatic fusion mechanism.

Bulk Synchronous Parallel ML or BSML is a purely functional programming language [8]. It is currently implemented as a library for OCaml, on top of any C MPI [13] implementation. It thus allows execution on a wide variety of parallel architectures and is especially well suited as an extraction target for Coq development.

In BSML, the underlying architecture is supposed to be a Bulk Synchronous Parallel (BSP) [17] computer. It is an abstract architecture as, with the help of a software layer, any general purpose parallel computer can be seen as a BSP computer. Such a computer is a *distributed* memory architecture: a set of p processor-memory pairs, connected through a network that allows point-to-point communications, together with a global synchronisation unit. A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjointed phases: (a) Each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes; (b) the network delivers the requested data transfers; (c) a global synchronisation barrier occurs, making the transferred data available for the next super-step.

BSML primitives are shallowly embedded in Coq. Their specifications are given in a module type `PRIMITIVES`. In this module, a partial description of a BSP architecture is as follows:

```
Section Processors.
  Parameter bsp_p : nat. (* the number p of processors *)
```

```
  Axiom bsp_pLtZero: 0 < bsp_p. (* we have at least one processor *)
  Definition processor : Type := { pid: nat | pid < bsp_p }.
End Processors.
```

BSML is based on a distributed data structure, named *parallel vector*. In OCaml its abstract type is `'a par`, and in Coq we have `par: Type →Type`. We write informally $\langle\, v_0\ , \ldots,\ v_{p-1} \,\rangle$ a value of this type. There is *one* value per processor of the BSP computer, and nesting is not allowed: These values should be "sequential", i.e. they cannot be or contain parallel vectors. BSML offers a global view of a parallel program: It looks like a sequential program but manipulates parallel vectors. The implementation however is a parallel composition of sequential communicating programs.

There are four primitives to deal with parallel vectors. Their signatures and informal semantics follows:

```
mkpar:(processor→A)→par A   apply:par(A→B)→ par A→par
proj:par A→processor→A      put:par(processor→A)→par(processor→A)
```

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

`mkpar` is used to create parallel vectors whose values are given by a function f. The function f should be a sequential function, i.e. it should not call any of the BSML primitives. `proj` is its inverse. `apply` denotes the pointwise application of a parallel vector of functions to a parallel vector of values. `put` is used to exchange data between processors. In the input parallel vector of functions, each function encodes the messages to be sent to other processors. $(f_i\ j)$ is the message to be sent by processor i to processor j. In BSML OCaml implementation the first constant constructor of a type is considered as the "empty" message and incurs no communication cost. The functions in the output vector encode received messages. If the input and output vectors are thought as matrices (each of the p functions can produce p values when applied to the p processor names), `put` is matrix transposition. Note that at a given processor there is no way to directly access the value held by another processor: `put` is needed. Both `proj` and `put` require a full BSP super-step to be evaluated. `mkpar` and `apply` are evaluated during the computation phase of a super-step.

In order to formalise this semantics in Coq, we need an "observer" function that is able to get the values held in a parallel vector, for which extensional equality implies equality:

```
Parameter get:∀ A: Type, par A→processor→A.
Axiom par_eq:∀(A:Type)(v w:par A),(∀(i:processor),get v i=get w i)→v=w.
```

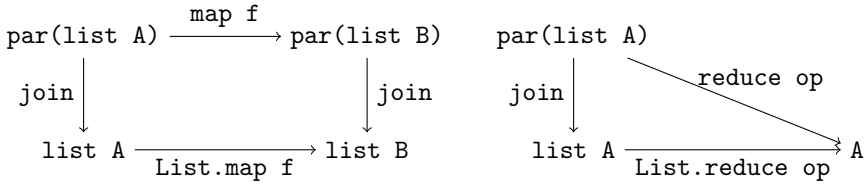
It is then straightforward to formalise the semantics. For example:

```
Parameter put: ∀vf:par(processor→A),
  { X: par(processor→A) | ∀i j: processor, get X i j = get vf j i }.
```

```
Module List.
  Include Coq.Lists.List.
  Definition reduce `(op:A →A →A) `{m: Monoid A op e} := fun l ⇒ fold_left op l e.
End List.
Module MR (Import Bsml : PRIMITIVES).
  Program Definition map `(f:A→B)`(v:par(list A)) : par(list B) :=
    apply (mkpar (fun _ ⇒ List.map f)) v.
  Program Definition reduce `(op:A →A →A)`{m: Monoid A op e}(v:par(list A)) : A :=
    List.reduce op (List.map (proj (apply (mkpar (fun _ ⇒ List.reduce op)) v)) processors).
End MR.
```

Fig. 6. Parallel `map` and `reduce`

Verified algorithmic skeletons. If we think of a parallel vector of lists `par(list A)` as a *distributed* list, a parallel BSML implementation of `map` and `reduce` is shown in Figure 6, where `Monoid` is a type class for monoids, and `processors` is the list of all the processors. We proved that these parallel implementations of `map` and `reduce` are correct with respect to their sequential counter-parts, i.e:



where `join` transforms a distributed list to the sequential list it represents:

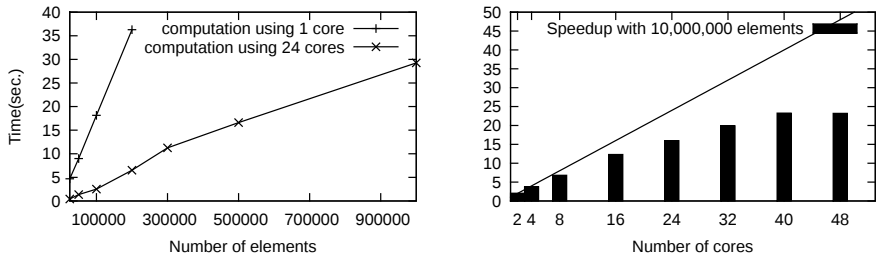
```
Definition join `(v:par(list A)):list A:= List.flat_map (get v) processors.
```

These correspondences are stored in instances of type classes, so that when the user requires the parallelisation of a composition of sequential functions for which there exist corresponding parallel implementations, it is done automatically. We use this feature of our framework to parallelise the outcome of the fusion mechanism.

Code extraction. Using the extraction mechanism of Coq, for example on the module `MR` of Figure 6, we obtain an OCaml functor that need to be applied to an implementation of the (extraction of the) module type `PRIMITIVES`. Actually, the BSML library for OCaml provides several implementations of a module type named `BSML`, including a MPI-based parallel one.

This module type almost contains the extracted module type `PRIMITIVES`. The only difference is that in the extracted module, the type `processor` is `nat` but in `BSML` it is `int`. We thus provide a wrapper module `BsmlNat` that performs conversions between processor representations. This conversion is correct as far as the processor name remains below $2^{30} - 1$ (1073741823) or $2^{62} - 1$ (4611686018427387903), depending on the architecture.

BSML implementations in OCaml and C are not formally verified yet. However we implemented a sequential version of the module type `PRIMITIVES` in Coq



(a) Execution Time (b) Speedup

Fig. 7. Execution Time and Speedup for the parallel execution of the extracted program

(using Coq vectors for implementing `par`). After extraction this provides a verified reference sequential implementation. It could be used to test the unverified implementations of BSML in OCaml.

5 Experiment results

The experiments were conducted on a shared memory machine containing 4 processors with 12 computer cores each (thus a total of 48 cores) and 64GB of memory. On each processors, there is two NUMA nodes, each node connecting 6 cores. On this particular architecture, we noticed in other experiments that there is a performance loss when there is active communication at the same time on the two NUMA nodes of one processor. The operating system is Ubuntu, the used languages and libraries are: Open MPI 1.5, BSML 0.5, and OCaml 3.12.1.

We extracted the parallel version of the `knapsack'` program in Section 3.3. We then measured the scalabity of this programs run in parallel. Figure 7(a) shows timings for computations on different number of elements for a knapsack with a capacity of 30. The list contains elements of random weight and value (always lesser than 10). The computation time grows linearly with the number of elements. In the sequential case (`poly_subs` implementation), for lists over 200,000 the program fails due to stack overflow, as `poly_subs` is non tail recursive. For the parallel version however, the `map` we use is a tail recursive one. It is therefore possible to consider much bigger input lists. Figure 7(b) shows the mean (over 30 measures) *relative* (i.e. the reference implementation is the same program but executed with only one core) speedup for a computation over a list of 10,000,000 elements.

VerifiedGTA is one of the component of the SyDPaCC system. The full development (version "ITP2014") is available at the web [15].

6 Related Work

To our knowledge SyDPaCC [6], on which VerifiedGTA is based, is the only framework, that makes possible the extraction of compilable and scalable parallel programs from a development in a proof assistant.

Among the work on formal semantics for BSP computations, the only one used to generate actual programs is LOGS [19]. In this approach, parallel programs are built by composing sequential programs in parallel whereas we adopt a global view. The GTA specifications required from the user are much simpler than LOGS specifications, but less general. To our knowledge no part of LOGS is formally verified. BSP-Why [5] is an extension of Why2 for the verification of imperative BSP programs but it does not support program derivation.

The style of programming we follow is polymorphic. Mu et al. provides a framework for polytypic programming and program transformation in Coq [11]. Expressiveness is very interesting but it would be a challenge to provide correspondence between polytypic sequential functions and parallel ones.

Lupinski et al. [9] formalised the semantics of a skeletal parallel programming language. This work is based on a deep embedding: one formalisation provides the high-level semantics of the skeletons, another one provides a model of the implementation in Join-calculus. From this model an implementation of the skeletons in JoCaml is designed. The semantics of BSML being purely functional, it is possible to have a shallow embedding in Coq, and then to write BSML programs in Coq and finally extract OCaml/BSML programs. Neither BSML, nor JoCaml implementations have been proved correct with respect to their calculi.

Swierstra [14] formalised mutable arrays in Agda, and added explicit distributions to these arrays. He can then write and reason on algorithms on these distributed arrays. The two main examples are a distributed map, and a distributed sum. In BSML the distribution of parallel vectors is fixed. On the other hand, it is possible to defined a higher-level data structure on top of parallel vector and consider various distributions of the data structure in parallel vectors. BSML in Coq remains purely functional. It would be possible to consider parallel vectors of mutable arrays, and even extract such BSML imperative programs to OCaml as it was done for imperative programs by Malecha et al. [10].

7 Conclusion and Future Work

The verified GTA library in Coq allows to derive and extract an efficient Bulk Synchronous Parallel ML program from a naive program definition (a specification) as a composition of a generator, a tester and an aggregator. We experimented an extracted application on two parallel architectures.

Future work includes a specialisation of the framework for the case where the monoid of map keys is finite. This allows to replace the finite maps by tuples (or arrays) for which we have direct access to elements. We are also interested in extending the framework to handle GTA programs on other structures than lists, such as trees and graphs.

Acknowledgements This work was partially supported by JSPS (KAKENHI Grant Number 24700025), ANR (ANR-2010-INTB-0205-02) and JST (10102704).

References

1. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989), available at `http://homepages.inf.ed.ac.uk/mic/Pubs`
2. Corra, R.C., Farias, P.M., de Souza, C.P.: Insertion and sorting in a sequence of numbers minimizing the maximum sum of a contiguous subsequence. Journal of Discrete Algorithms 21, 1 – 10 (2013)
3. Emoto, K., Fischer, S., Hu, Z.: Filter-embedding semiring fusion for programming with MapReduce. Formal Aspects of Computing 24(4-6), 623–645 (2012)
4. Emoto, K., Fischer, S., Hu, Z.: Generate, Test, and Aggregate – A Calculation-based Framework for Systematic Parallel Programming with MapReduce. In: ESOP. LNCS, vol. 7211, pp. 254–273. Springer (2012)
5. Gava, F., Fortin, J., Guedj, M.: Deductive Verification of State-Space Algorithms. In: IFM. LNCS, vol. 7940, pp. 124–138. Springer (2013)
6. Gesbert, L., Hu, Z., Loulergue, F., Matsuzaki, K., Tesson, J.: Systematic Development of Correct Bulk Synchronous Parallel Programs. In: International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). pp. 334–340. IEEE (2010)
7. Ho, T.J., Chen, B.S.: Novel extended viterbi-based multiple-model algorithms for state estimation of discrete-time systems with markov jump parameters. IEEE Transactions on Signal Processing 54(2), 393–404 (2006)
8. Loulergue, F., Hains, G., Foisy, C.: A Calculus of Functional BSP Programs. Science of Computer Programming 37(1-3), 253–277 (2000)
9. Lupinski, N., Falcou, J., Paulin-Mohring, C.: Sémantique d'une langage de squelettes. `http://www.lri.fr/~paulin/Skel/article.pdf` (2012)
10. Malecha, G., Morrisett, G., Wisnesky, R.: Trace-based verification of imperative programs with i/o. J. Symb. Comput. 46(2), 95–118 (2011)
11. 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 Berlin / Heidelberg (2008)
12. Otto, F., Sokratova, O.: Reduction relations for monoid semirings. Journal of Symbolic Computation 37(3), 343 – 376 (2004)
13. Snir, M., Gropp, W.: MPI the Complete Reference. MIT Press (1998)
14. Swierstra, W.: More dependent types for distributed arrays. Higher-Order and Symbolic Computation 23(4), 489–506 (2010)
15. SyDPaCC Home Page. `http://traclifo.univ-orleans.fr/SyDPaCC`
16. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM 33(8), 103 (1990)
17. Wadler, P.: Theorems for free! In: Proceedings of the fourth international conference on Functional programming languages and computer architecture. pp. 347–359. ACM (1989)
18. 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)