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

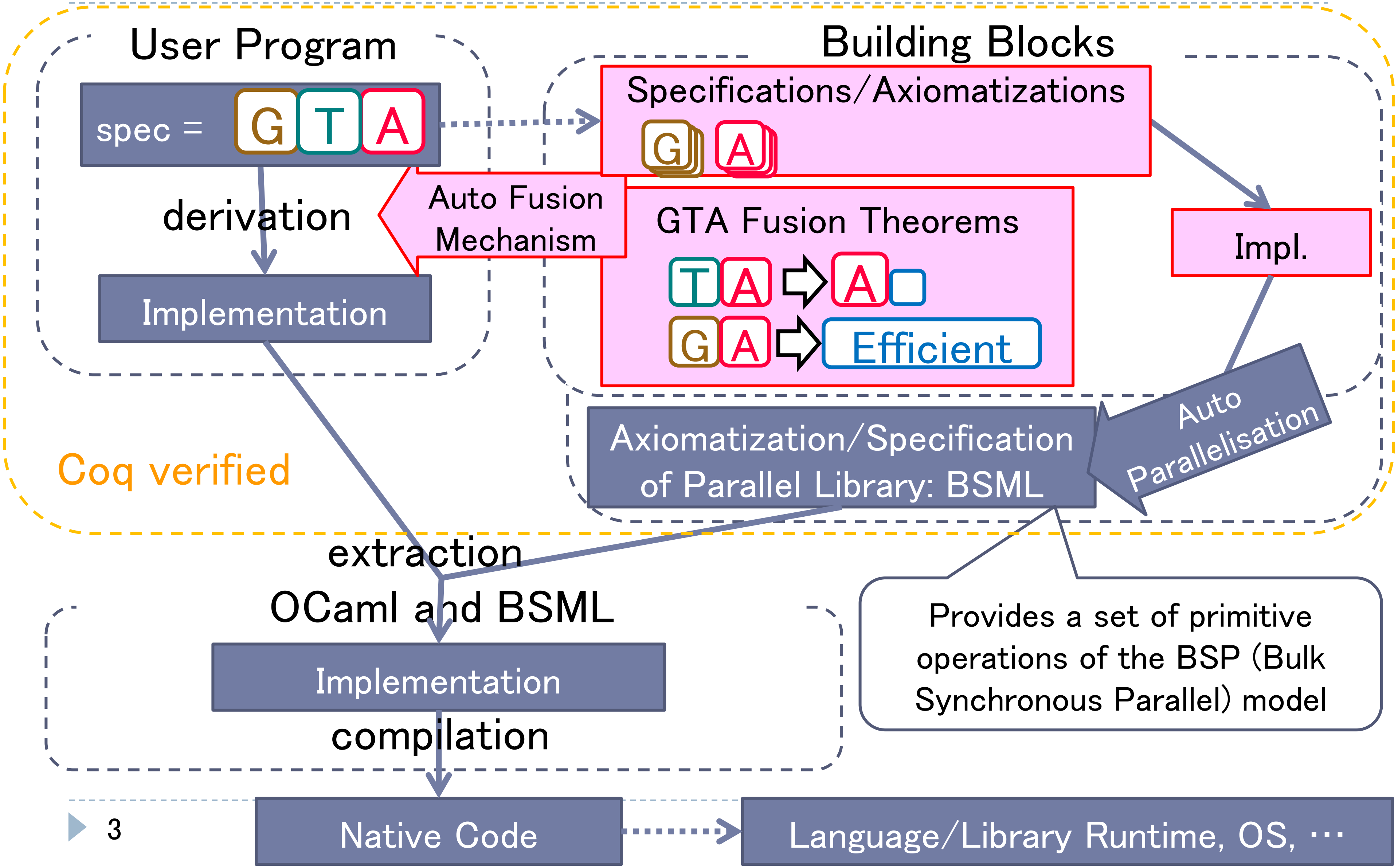
Kento EMOTO
Kyushu Institute of Technology, Japan

Joint work with Frédéric Loulergue and Julien Tesson
Université d'Orléans and Université Paris-Est, France

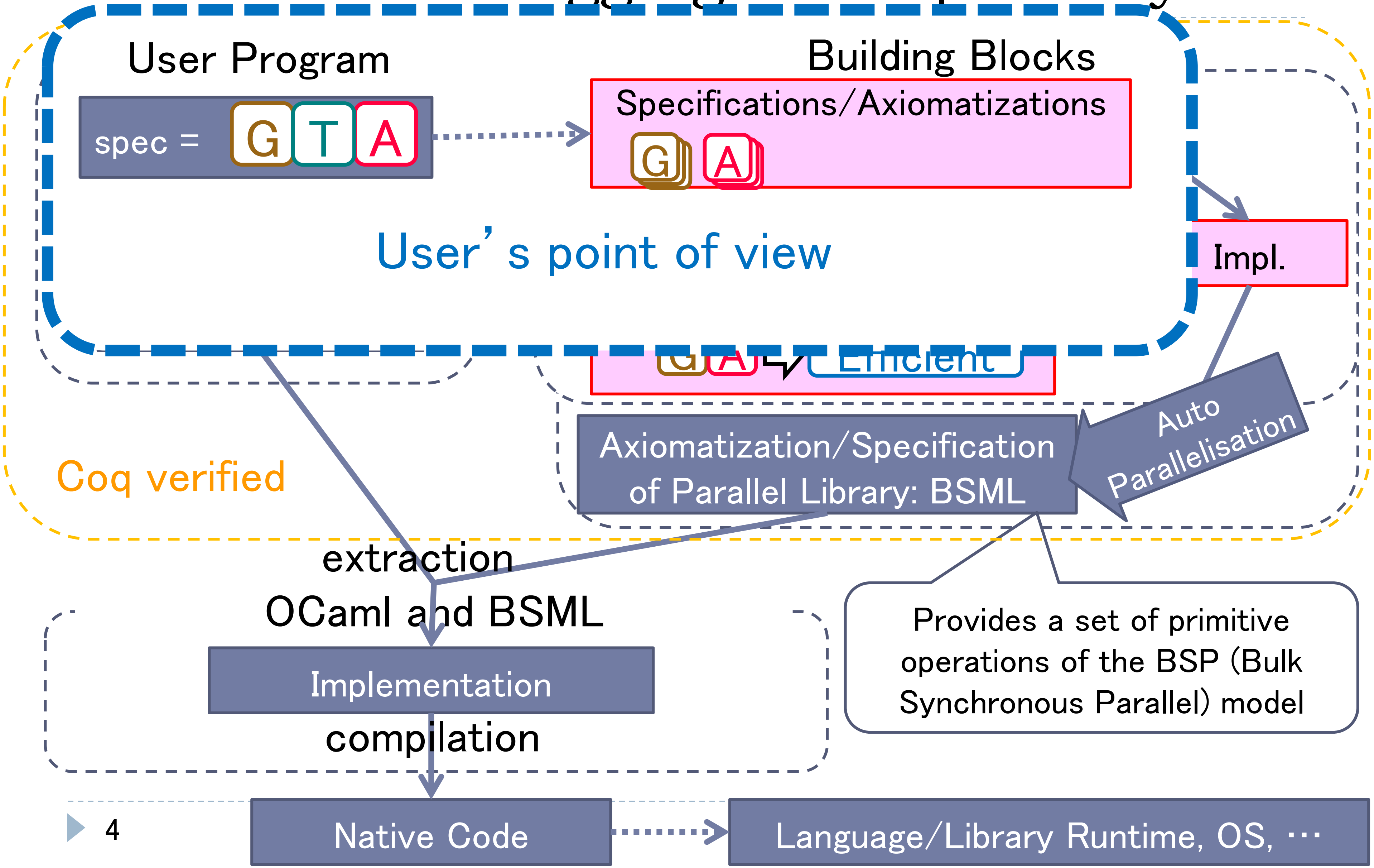
Background & Motivation

- Parallel programming is necessary, but not easy
  - Parallelism is the only way to gain performance
  - Writing/maintaining code with low-level parallelism is difficult
- High-level parallel programming has been proposed (e.g., skeletal parallel programming [Cole 89])
  - Writing code by composing building blocks hiding low-level parallelism
  - Easy to write/maintain parallel programs
- Generate-test-aggregate programming [Emoto et al. ESOP '12]
  - Naïve program = composition of generator, tester and aggregator
  - Theory to derive efficient implementation from a naïve program
  - Prototype Scala library with automatic derivation [Liu et al. PMAM'13]
- Question: Is such a library correctly implemented?
- This study: Verified generate-test-aggregate library on Coq

Generate-Test-Aggregate Coq Library

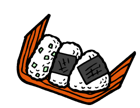
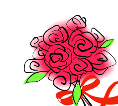


Generate-Test-Aggregate Coq Library



Running Example: 0-1 Knapsack Problem

- Given a knapsack and a set of items, find the most valuable selection of items adhering to the knapsack's weight restriction

- The best total value is $120 by choosing ,  and 

Writing Your Naïve Code in GTA Form

```
Definition naive_prog := aggregate :o: test :o: generate.
```

- Generate all candidate substructures of the input
- Test and discard unnecessary candidates
- Aggregate the valid candidates to make the final result

Writing Your Naïve Code in GTA Form

Definition knapsack w := maxValue :o: validWeight w :o: allSelections.

- allSelects generates all item selections
- validWeight filters out selections with total weight heavier than w
- maxValue takes the maximum total value (for simplicity, value only)

Given a knapsack and a set of items, find the most valuable selection of items adhering to the knapsack’s weight restriction

User's Point of View

---Writing Your Naïve Code---

Generator: Generating All Candidates

```
Definition knapsack w := maxValue :o: validWeight w :o: allSelections.
```

- generate : [A] -> { [A] }
  - { X } is the type of bags (multi-sets) of X
- You may design your generators, but it is not easy
- The library provides a set of ready-made generators
  - subs for all sublists
  - segs/inits/tails for all contiguous sublists/prefixes/suffixes
  - …
- For the knapsack problem, we choose the subs generator:

```
Definition allSelections := subs .
```

---Writing Your Naïve Code---

Tester: Discarding Invalid Candidates

```
Definition knapsack w := maxValue :o: validWeight w :o: allSelections.
```

- test : { [A] } -> { [A] }
  - A filter operation of a bag with predicate p of a specific kind:
    Definition p := ok :o: fold_right ($\odot$) $i_\odot$:o: map f
    - ok : a lightweight judgment
    - $\odot$: a monoid operator with the identity element $i_\odot$

      (Monoid: an associative binary operator with its identity element)
- For the knapsack problem, p checks the total weight:

```
Definition totalWeight := fold_right (+) 0 :o: map getWeight .
Definition p w := (fun a => a <= w) :o: totalWeight .
Definition validWeight w := filter (p w).
```

---Writing Your Naïve Code---

Aggregator: Making the Final Result

```
Definition knapsack w := maxValue :o: validWeight w :o: allSelections.
```

- aggregate :: { [A] } -> S
  - S is a type of the final result
- You may design your aggregators, but it is not easy
- The library provides a set of ready-made aggregators
  - maxsum f for finding the maximum f-weighted sum
  - sumprod f, count, maxsumSolution f, longest, top-k variants, …
- For the knapsack problem, we can use the maxsum aggregator:

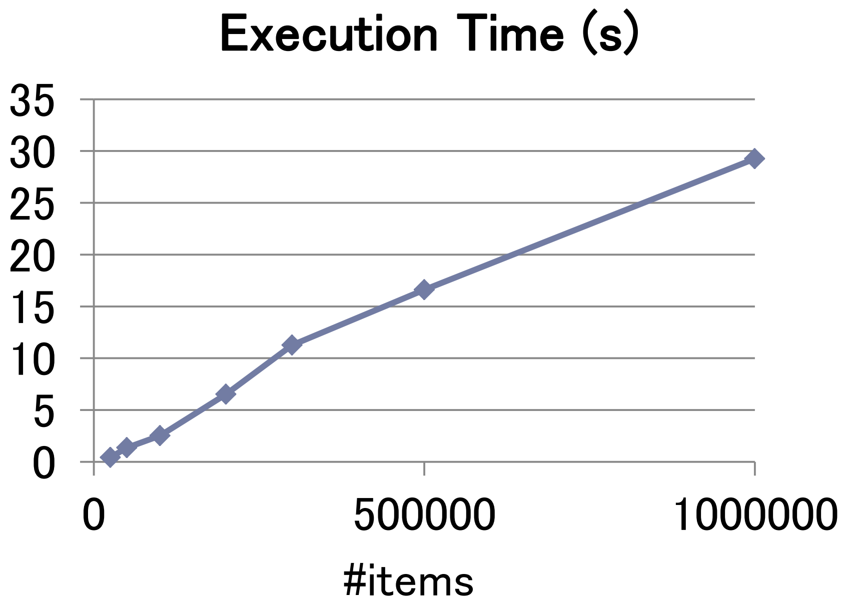
```
Definition maxValue := maxsum getValue .
```

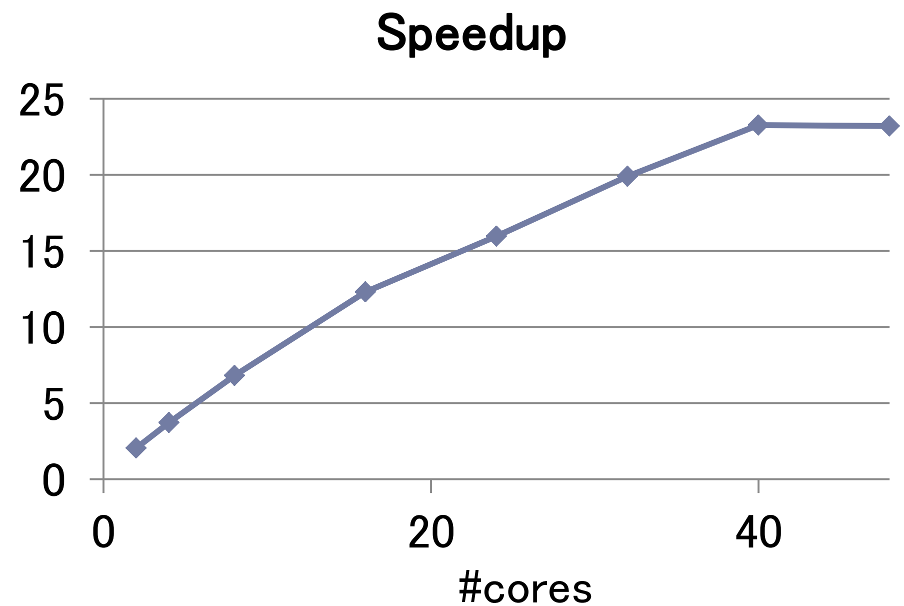
All You Need to Do

```
Definition allSelections  := subs .
Definition totalWeight := fold_right (+) 0 :o: map getWeight .
Definition p w := (fun a => a <= w) :o: totalWeight .
Definition validWeight w := filter (p w).
Definition maxValue := maxsum getValue .
Definition knapsack w := maxValue :o: validWeight w :o: allSelections.
(* check the naïve program *)
Eval compute in (knapsack 3 [item 2 1; item 2 2; item 3 2 ]).
(* small proofs related to the naïve program *)
Program Instance totalWeight_monoidOp :
  isUsingMonoidOp totalWeight getWeight plus 0 := fold_right_monoid.
Program Instance proper_getWeight : Proper (eq_item ==> eq) getWeight.
Next Obligation. (* omit *) Defined.
Definition knapsack_opt w := fused (tgt := knapsack w ). (* auto derivation*)
```

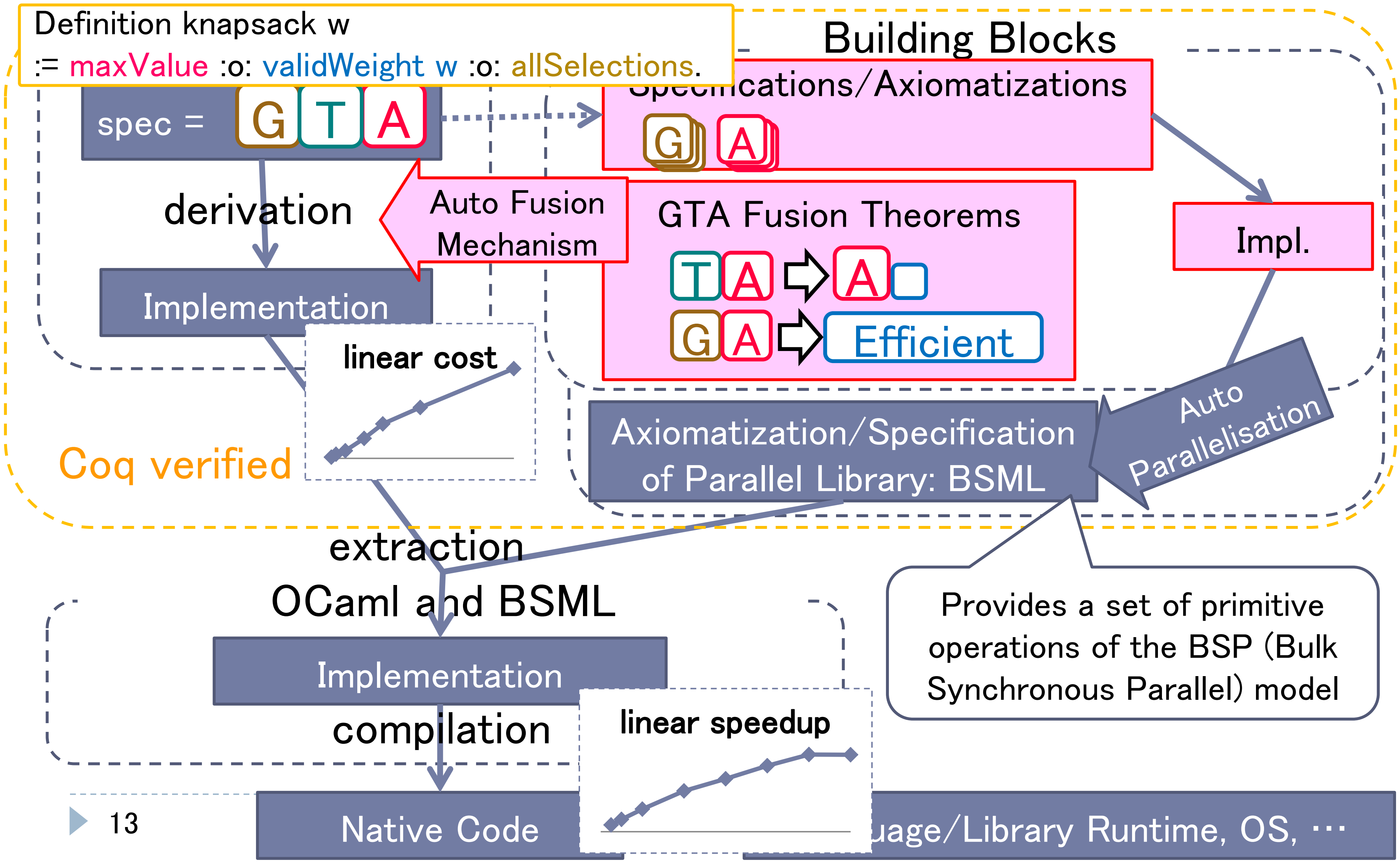
Experiment Results on Extracted Code

- knapsack_opt (auto optimized, parallelized knapsack) has been extracted to OCaml + BSML (BSP primitives)
- Cost is linear in #items, although the naïve program looks an exponential cost program
- Good speedup (except for the fully busy case)
  - 64GB shared memory, 48 cores = 12 cores x 4 processors

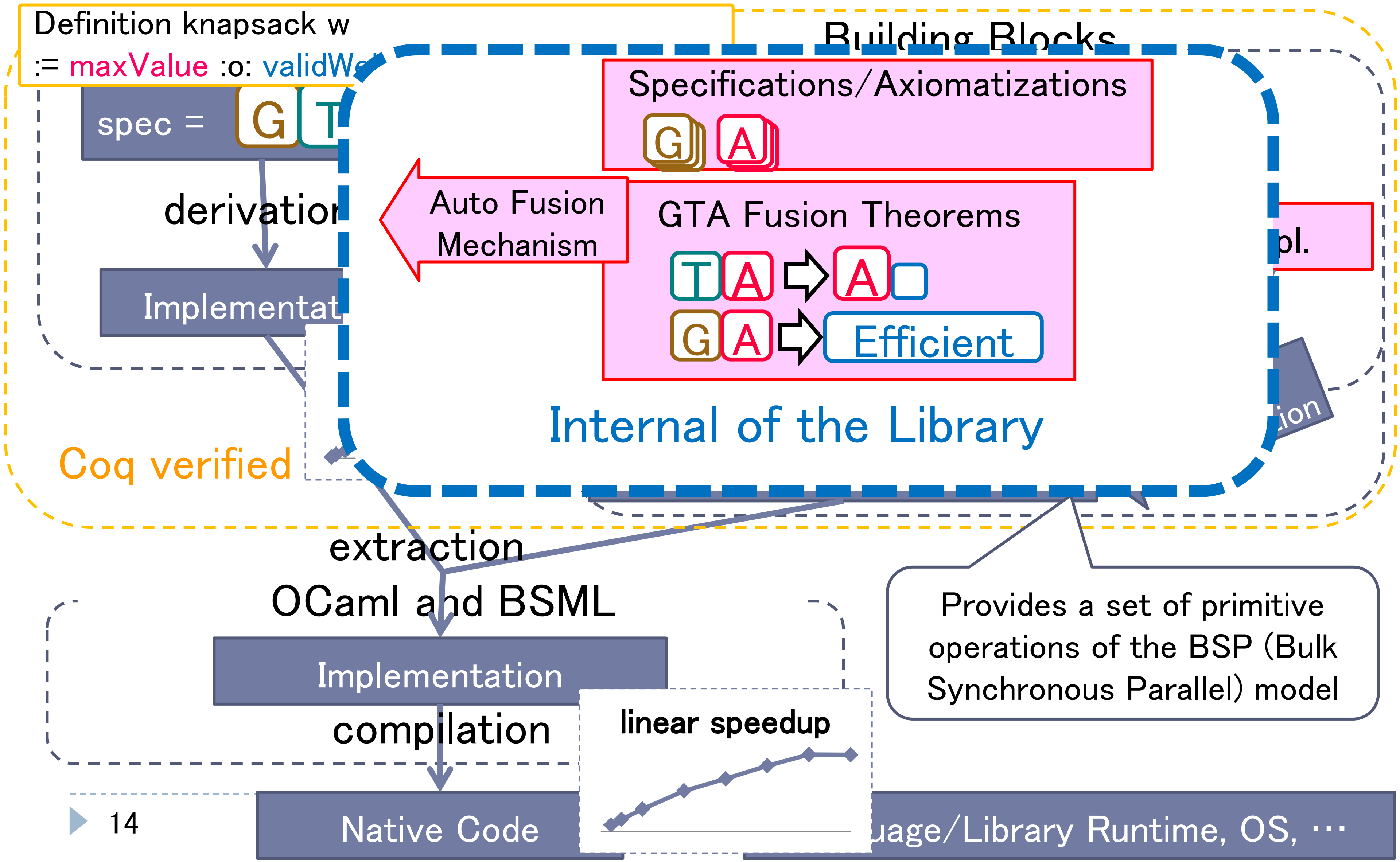




Generate-Test-Aggregate Coq Library



Generate-Test-Aggregate Coq Library



Derived Implementation of knapsack

- E.g., knapsack_opt 2kg [(1kg, $10), (1kg, $20), (2kg, $20)]

= postproc (

0kg	$0
1kg	$10

⊗

0kg	$0
1kg	$20

⊗

0kg	$0
2kg	$20

)

= postproc (

0kg	$0
1kg	$20
2kg	$30

⊗

0kg	$0
2kg	$20

) = postproc (

d	
0kg	$0
1kg	$20
2kg	$30
3kg+	$50

)

= $30

Parallel time complexity: O(wn/p + w^2 log p) (n = #items, p = #cores)

- Auto-derivation mechanism derives this by using two verified transformation theorems

Automatic Fusion

- Fusion: eliminating intermediate data structures between two funcs:
  - E.g., map f (map g x) = map (f :o: g) x
- Basic idea: Use the typeclass resolver for an automatic search
  - Auto-parallelization has been implemented by the same tech. [Tesson 11]
- Two typeclasses: Fusion for a rule DB and Fuser for a trigger

```
Class Fusion `(producer : B -> C) `(consumer : C -> D) (_fused : B -> D) := {
  _spec : forall b, consumer (producer b) === _fused b }.
```

```
Class Fuser `(tgt : B -> D) := {
 fused : B -> D;     spec : forall b, tgt b === fused b }.
Global Program Instance fuser `{fusion : Fusion producer consumer _fused}
: Fuser (consumer :o: producer) := { fused := _fused; spec := _spec }.
```

Automatic Fusion Mechanism

- Definition opt := fused (tgt := f :o: h).

3rd, replaced with fh_fused

1st, looking for an instance of Fuser (f :o: h)

2nd, looking for an instance of Fusion h f _

Fusion p c _fused => Fuser (c :o: p)

Fusion h f fh_fused
(* f :o: h === fh_fused *)

Fusion g f fg_fused
(* f :o: g === fg_fused *)

Fusion f h hf_fused
(* h :o: f === hf_fused *)

Fusion f' f ff_fused
(* f :o: f' === ff_fused *)

Fusion g h hg_fused
(* h :o: g === hg_fused *)

Instance pool

Verified Fusion Theorems

- Filter-embedding Fusion T A ⇒ A pp
  - New aggregator does computation on tables

```
Theorem filterEmbeddingFusion
`(c1 : isNestedFoldsWithSemiring  aggregate  f oplus otimes ep et)
`(c2 : isFilterWithFoldWithMonoid  test  h odot e ok dec)
: forall x,
 (aggregate :o: test) x === (postproc :o: nestedFolds mkTable semiringOnTables) x.
```

- Semiring Fusion G A ⇒ Efficient
  - A kind of shortcut fusion (substitution of consumer's operators)

```
Theorem semiringFusion
`(c1 : isNestedFoldsWithSemiring        aggregate   f oplus otimes ep et)
`(c2 : isSemiringPolymorphicGenerator generate      polygen)
: forall x, (aggregate :o: generate) x === (polygen f (oplus, otimes, ep, et)) x.
```

Other Applications Include…

- More restriction on selections in the Knapsack Problem
  - E.g., "Item B must be contained if item C is contained",
    "The number of items with value > $100 is at most 5",
    "Select an even number of items", etc.
  - Your GTA program can have multiple testers
- Finding the most likely sequence of hidden events from a sequence of observed events (Viterbi and its variants)
- Finding the longest (most valuable) segment (region) satisfying a set of conditions
- etc

Conclusion

- A Verified Generate-Test-Aggregate Coq Library
  - Equipped with an automatic fusion mechanism
  - Proofs of two fusion theorems
  - You can write an easy-to-design/verify/modify naïve program, but get an efficient parallel program
  - Extracted code runs on BSML/OCaml on parallel machines
  - Axiomatization/Implementation of Bags, typeclass-based Maps, Monoid semiring (algebra of tables), …
- Subjects in future studies
  - Extension of the theory to trees and graphs
  - Use of efficient implementation of 'tables'
  - Code extraction for execution on Hadoop/MapReduce

Thank you for listening.

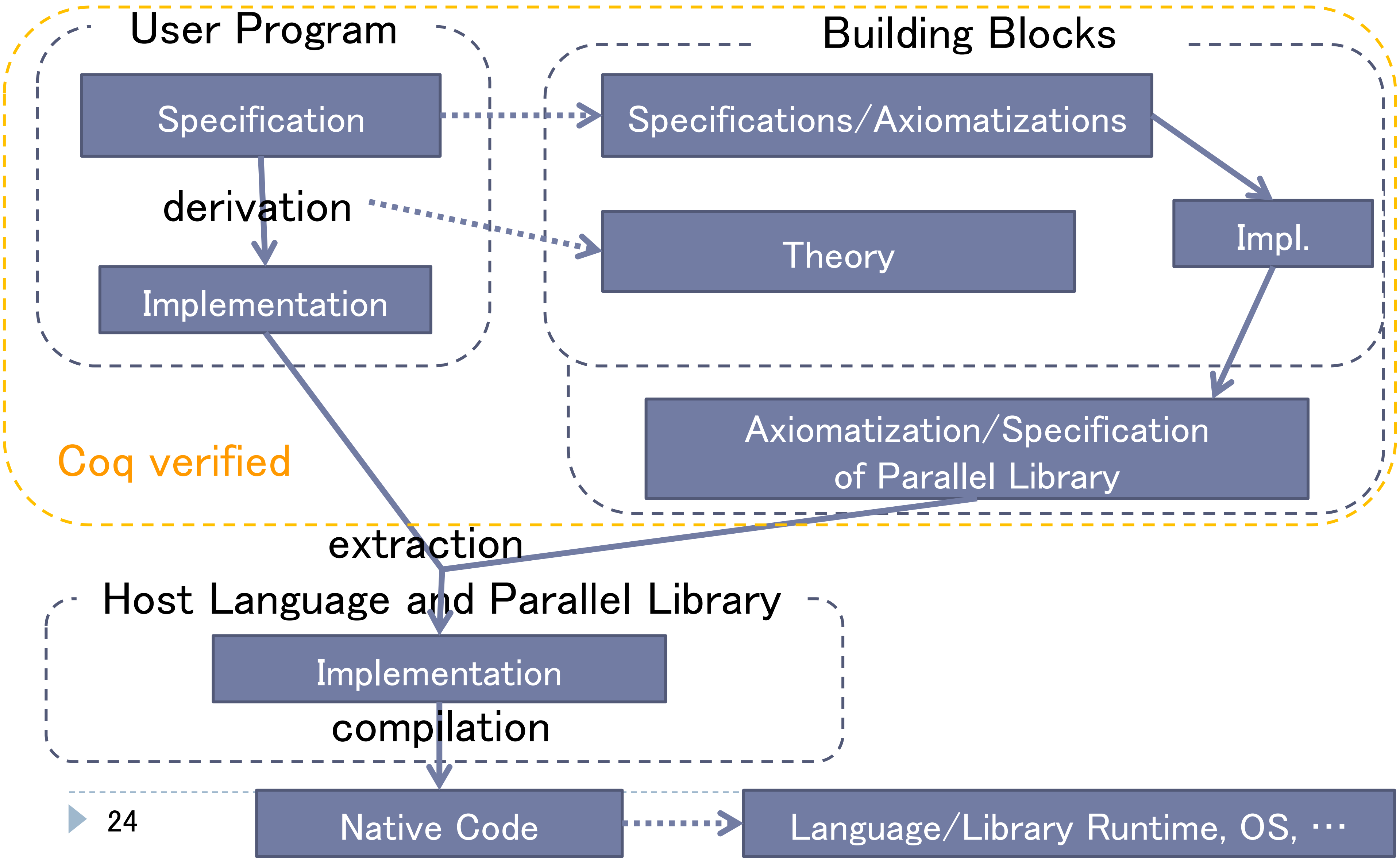
Visit the following URL for the library code:

http://traclifo.univ-orleans.fr/SyDPaCC

SyDPaCC

- Systematic Development of Programs for Parallel and Cloud Computing

SyDPaCC: Systematic Development of Programs for Parallel and Cloud Computing



Finitization and Automatic Finitization

- Making the range R of the homomorphism in a filter is important to the performance of the derived program
  - The cost of the multiplication operator on tables: $O(|R|^2)$
- We can use { x : nat | x <= w + 1 } as R, instead of nat, for Definition p := comparison_with w :o: sum_of_nats
  - The comparison may be (<= w), (==w), (>=w), (==)

```
(** automatic finitization of the predicate *)
Definition weightLimit' (w : nat) := rewrite_p (p := weightLimit w).
Definition validWeight' (w : nat) := filterB (weightLimit' w) dec_spec.
Definition knapsack' (w : nat) := maxvalue :o: validWeight' w :o: subs.

(** The linear cost program. *)
Definition knapsack'_opt (w : nat) := Eval simpl in fused (f := knapsack' w).
```

Internal of the Library

1st Fusion Theorem:

Filter-embedding Fusion

T A ⇒ A pp

```
Theorem filterEmbeddingFusion
`(c1 : isNestedFoldsWithSemiring  aggregate  f oplus otimes ep et)
`(c2 : isFilterWithFoldWithMonoid  test  h odot e ok dec)
: forall x,
 (aggregate :o: test) x === (postproc :o: nestedFolds mkTable semiringOnTables) x.
```

- The first condition says
  an aggregator is a nested folds with semiring operators:

  Definition nestedFolds f $(\oplus, \otimes, i_\oplus, i_\otimes)$
  $:=$ fold_{bag} $(\oplus)$ $i_\oplus$:o: map_{bag} (fold_right $(\otimes)$ $i_\otimes$:o: map f).

  - Semiring: monoid op. $\otimes$ distributes over commutative monoid op. $\oplus$,
    and $i_\oplus$ is the absorbing element of $\otimes$.

- New aggregator does computation on tables
  - The structure of tables is derived from the tester & aggregator

Internal of the Library

T A ⇒ A pp

1st Fusion Theorem:

Filter-embedding Fusion

```
Theorem filterEmbeddingFusion
`(c1 : isNestedFoldsWithSemiring  aggregate  f oplus otimes ep et)
`(c2 : isFilterWithFoldWithMonoid  test  h odot e ok dec)
: forall x,
 (aggregate :o: test) x === (postproc :o: nestedFolds mkTable semiringOnTables) x.
```

- New aggregator does computation on tables
- For the knapsack problem with w = 2kg,
  - mkTable ("1kg, $10") =

Total weight	Max. total value
1kg	$10

  - postproc (T) = $30, where T is the table:

T.W.	M. T. V.
0kg	$0
1kg	$30
2kg	$20
3kg+	$50

```
Definition totalWeight
  := foldr (+) 0 :o: map getWeight.
Definition p w
  := (fun a => a<=w) :o: totalWeight .
```

Internal of the Library

T A ⇒ A_{pp}

1^{st} Fusion Theorem:
Filter-embedding Fusion

- Two table merge operations ⊕ and ⊗ (w = 2kg)

T.W.	M. T. V.
0kg	$0
1kg	$30
2kg	$10

⊕

T.W.	M. T. V.
0kg	$0
2kg	$20

=

T.W.	M. T. V.
0kg	$0
1kg	$30
2kg	$20

(row−wise maximum)

T.W.	M. T. V.
0kg	$0
1kg	$30
2kg	$10

⊗

T.W.	M. T. V.
0kg	$0
2kg	$20

=

T.W.	M. T. V.
0kg	$0
1kg	$30
2kg	$20
$3kg^+$	$50

(all possible combination)

Note: since the weight limit w = 2kg, entries greater than 3kg are unnecessary.
This finitization of tables can be done automatically in a similar way to the fusion

2nd Fusion Theorem:

Semiring Fusion

G A ⇒ Efficient

```
Theorem semiringFusion
`(c1 : isNestedFoldsWithSemiring        aggregate   f oplus otimes ep et)
`(c2 : isSemiringPolymorphicGenerator generate      polygen)
: forall x, (aggregate :o: generate) x === (polygen f oplus otimes ep et) x.
```

- The second condition (instance) says
  - generate = polygen + "constructors of bags of lists", and
  - polygen accepts any semiring operators (i.e., polymorphic)
- Constructors of basg of lists:
  - Cross-concatenation: $\{x, y\} \times_{++} \{z, w\} = \{ x{+}{+}z, x{+}{+}w, y{+}{+}z, y{+}{+}w\}$
  - Union: $\{x, y\} \cup \{z, w\} = \{x, y, z, w\}$
- Definition poly_subs f (op, ot, ep, et)
  := fold_right ot et :o: map (fun x => op (f x) et)
- Definition subs := poly_subs (fun x => { [x] }) $(\times_{++})$ $(\cup)$ { [ ] } { }
  - E.g., subs [1, 2] = $(\{[1]\} \cup \{[\,]\}) \times_{++} (\{[1]\} \cup \{[\,]\})$
    $= \{[1], [\,]\} \times_{++} \{[2], [\,]\} = \{[1,2], [1], [2], [\,]\}$

Property of Polymorphic Functions

```
Class isSemiringPolymorphicFunction
  (pgen : forall {V:Type}, (T -> V) -> (V->V->V) -> (V->V->V) -> (V) -> (V) -> V)
  :={semiringPolymorphism :
     forall {V:Type} (f : T -> V) (oplus : V->V->V) (otimes : V->V->V)  (ep et : V),
      FSHom f oplus otimes ep et (pgen FS_F FS_OPLUS FS_OTIMES FS_EP FS_ET)
        = pgen f oplus otimes ep et
}.
```

- All instances of a polymorphic function act in the same way.
  - Evaluation of a computation tree constructed by a polymoprhic function produces the same result as computing the result directly by the polymorphic function

FreeSemiring and its Homomorphism

```
Inductive FreeSemiring :=
      | FS_F : T -> FreeSemiring
      | FS_OPLUS : FreeSemiring -> FreeSemiring -> FreeSemiring
      | FS_OTIMES : FreeSemiring -> FreeSemiring -> FreeSemiring
      | FS_EP : FreeSemiring
      | FS_ET : FreeSemiring.

Fixpoint FSHom {V:Type} (f : T -> V) (oplus otimes : V->V->V)  (ep et : V) (x)
:= match x with
   | FS_F a => f a
   | FS_OPLUS l r =>
            oplus (FSHom f oplus otimes ep et l) (FSHom f oplus otimes ep et r)
   | FS_OTIMES l r =>
            otimes (FSHom f oplus otimes ep et l) (FSHom f oplus otimes ep et r)
   | FS_EP => ep
   | FS_ET => et
  end.
```

Semiring $(\oplus, \otimes, o, i)$

- Associativity: $x \oplus (y \oplus z) = (x \oplus y) \oplus z$
  $x \otimes (y \otimes z) = (x \otimes y) \otimes z$
- Commutativity: $x \oplus y = y \oplus x$
- Distributivity: $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$
- Identities: $x \oplus o = o \oplus x = x$
  $x \otimes i = i \otimes x = x$
- Absorbing: $x \otimes o = o \otimes x = o$

  - Semiring $(\oplus, \otimes, o, i)$
    = Monoid $(\otimes, i)$ + Commuataive Monoid $(\oplus, o)$
    + Distributivity + Absorbing

Monoid Semiring

- Given a semiring $(\oplus, \otimes, \mathtt{o}, \mathtt{i})$ on S and monoid $(\odot, \mathtt{e})$ on M, we can make a new semiring on linear combinations (tables).
  - Linear combination: $s_1\ m_1 + \cdots + s_k\ m_k$ (table view:)

m_1	s_1
$\cdots$	$\cdots$
m_k	s_k

  - Addition: $s_1\ m + s_2\ m = (s_1 \oplus s_2)\ m$
    (otherwise no effect)
  - Multiplication:
    $$(s_1\ m_1 + \cdots + s_k\ m_k) \times (t_1\ n_1 + \cdots + t_j\ n_j)$$
    $$= (s_1 \otimes t_1)\ (m_1 \odot n_1) + \cdots + (s_k \oplus t_1)\ (m_k \odot n_1)$$
    $$+ \cdots$$
    $$+ (s_1 \otimes t_j)\ (m_1 \odot n_j) + \cdots + (s_k \oplus t_j)\ (m_k \odot n_j)$$

All Assignments Generator

- assign [T,F] [a, b, c]
  = {   [(a, T), (b, T), (c, T)],
  [(a, T), (b, T), (c, F)],
  [(a, T), (b, F), (c, T)],
  [(a, T), (b, F), (c, F)],
  [(a, F), (b, T), (c, T)],
  [(a, F), (b, T), (c, F)],
  [(a, F), (b, F), (c, T)],
  [(a, F), (b, F), (c, F)] }