Our Goal

To ease the development of **correct** and **verified parallel** programs with **predictable performances** using theories and tools to allow a user to develop an application by using building blocks and implementing short programs satisfying conditions easily or automatically proved.
Parallel Programming

Structured Parallelism
- Automatic Parallelization
  - Algorithmic Skeletons
  - Bridging Models (BSP, ...)
  - Declarative Parallel Programming
  - ...

Concurrent & Distributed Programming

Bridging Models (BSP, ...)

Leslie Valiant in his 1990 CACM paper
'A Bridging Model for Parallel Computation'

Valiant's proposal: Bulk Synchronous Parallelism (BSP)
Other models: LogP and variants, BSP variants, ...

Algorithmic Skeletons

Coined by Murray Cole in

http://homepages.inf.ed.ac.uk/mic/Pubs/skeletonbook.ps.gz

Popular skeletons: Google's MapReduce

Skeletal Parallelism
- Skeleton = pattern of a parallel algorithm familiar sequential semantics
- Program = composition of skeletons

Libraries of Algorithmic Skeletons
- For C++: SkeTo\textsuperscript{a}, OSL\textsuperscript{b}, Muesli, QUAFF, ...
- For C: eSkel, SKElib
- For Java: Lithium, Muskel, Calcium, ...
- For functional languages:
  - OCaml: OCamP3L, Parmap
  - Erlang: Skel
  - Haskell: HaskSkel, Edenskeletons

Algorithmic Skeletons Theory
- List homomorphisms for parallel programming (Cole 1993)
- Many further developments in particular in Tokyo

*http://sketo.ipl-lab.org
http://trueclift.univ-orleans.fr/OSL
**Program Correctness**

**A posteriori verification**
Write the program then try to prove its correctness:
- verification condition generator + provers
- interactive theorem provers
- software model checking
- ...

**Correctness by construction**

- B method
- Bird Meertens Formalism (theory of lists, ...)
- ...

**Overview of our approach**

**Programs correct by construction**

- Write naive correct programs (specification)
- Apply program transformation techniques
- To obtain equivalent efficient programs
- That are automatically parallelised

**Increasing Confidence**

**Usage of the Coq proof assistant**

**Coq proof assistant**

- OCaml
- The Bulk Synchronous Parallel ML library

**Specifications Transformations Automatic parallelisation**

**Extraction to OCaml + BSML**

**Increasing Confidence**

**Usual Usage**

- Pen-and-paper program transformation
- From the last form: hand-written C++ code for a skeleton library (mostly in C, C++, Java)

**Potential Problems**

- Pen-and-paper transformation may be erroneous
- The C++ code may not be equivalent to the last form
- The skeleton library may contain bugs

→ Software assistance and verification needed
ACM SIGPLAN Software Award 2013

The Coq proof assistant provides a rich environment for interactive development of machine-checked formal reasoning. Coq is having a profound impact on research on programming languages and systems [...] It has been widely adopted as a research tool by the programming language research community [...] Last but not least, these successes have helped to spark a wave of widespread interest in dependent type theory, the richly expressive core logic on which Coq is based. [...] The Coq team continues to develop the system, bringing significant improvements in expressiveness and usability with each new release. In short, Coq is playing an essential role in our transition to a new era of formal assurance in mathematics, semantics, and program verification.

Curry-Howard Correspondance

Natural Deduction

1. \( \forall A : \Gamma \vdash A \)
2. \( \Gamma, A \vdash B \rightarrow C \)
3. \( A \rightarrow B \rightarrow C \)
4. \( \Gamma \vdash B \rightarrow C \)
5. \( \Gamma \vdash \text{let } e \text{ in } A \rightarrow B \rightarrow C \)
6. \( A \rightarrow B \rightarrow C \rightarrow D \)

Simply Typed \( \lambda \)-Calculus

1. \( \forall A : \Gamma \vdash A \)
2. \( \Gamma \vdash x : A \)
3. \( \Gamma, x : A \vdash e : B \)
4. \( \forall A \rightarrow B \rightarrow C \rightarrow D \)
5. \( \Gamma \vdash (\lambda x. e) : A \rightarrow B \rightarrow C \rightarrow D \)
6. \( \Gamma \vdash (\text{let } e \text{ in } A \rightarrow B \rightarrow C \rightarrow D) \)
Curry-Howard Correspondance

Natural Deduction – Example 2

\[
\begin{align*}
\text{(i)} & \quad \Gamma \vdash (A \rightarrow B) \rightarrow C \quad \text{B} \in \Gamma \\
\text{(v)} & \quad (\forall \text{x}) \quad \Gamma \vdash A \rightarrow C \quad \text{B} \in \Gamma \\
\text{(ii)} & \quad \Gamma \vdash \text{x} : A, \text{y} : B, \text{z} : C \Rightarrow \text{x} : A, \text{y} : B, \text{z} : C \\
\text{(v)} & \quad \Gamma \vdash \text{x} : A, \text{y} : B, \text{z} : C \\
\end{align*}
\]

\(\lambda x : A, \lambda y : B. \lambda z : C. (f \text{x})\) is a way to encode the proof tree of

\(A \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C\)

Curry-Howard Isomorphism

For all formula there exists a proof of this formula in natural deduction if and only if there exists a \(\lambda\)-term that has this formula as type.

- Theorem statement \(\Rightarrow\) Type
- Proof \(\Rightarrow\) Program

Coq in practice

- Functional programming language
- Rich type system: allow to express logical properties
- Language for building proofs (ie proof terms)
- Program extraction
Previous examples in Coq

The Proof General mode for Emacs ...

...or the CoqIDE

We open the file `Introduction.v`:

1 available at http://frederic.loulergue.eu/hpcs2017

We start to feed Coq with the commands:
We state a lemma and enter the interactive proof mode:

The tactic `intro` "apply" the (i) rule:

The context is now similar to Γ:

We apply rule (a) by naming the implication part:

and so now we have only to deal with A...
...that is an assumption, we use rule ($\forall$):

```
No more subgoals ⇒ proof done ⇒ $\lambda$-term built
```

Qed typechecks the term against the lemma statement:

```
Second version, we do multiple intro:

and apply HBC instead of apply HAC:
```
Previous examples in Coq

\[
\text{Print } t. \text{ prints the term } t:
\]

It is the \(\lambda\)-term we constructed "by hand".

---

Previous examples in Coq

We could give directly the proof as a \(\lambda\)-term:

...or use Coq more powerful tactics:
Interactive Sessions

Remaining of the Tutorial
- Demo-like using an interactive session of Coq
- The VM provides all the tools:
  - to have the same interactive session on your machine,
  - to do the proposed exercises.

The Virtual Machine
- http://frederic.loulergue.eu
- USB Flashdrive

Outline
1. Introduction and Motivation
2. Functional Programming in Coq
3. Proofs in Coq
4. Theory of Lists and Parallelisation in Coq
   An Overview of Bird Meertens Formalism
   Theory of Lists in Coq
   Applications
   Automatic Parallelization
5. Conclusion
6. Bibliography

Functional Programming in Coq

Data Structures, Values and Functions
- Inductive Types
- Pattern Matching
- Polymorphism
- Recursive Functions
- How to Deal with Partial Functions

Modularity
- Modules, Parametric Modules
- Type Classes

Extraction

Interactive Session
- Programming.v
Bird Meertens Formalism

John Backus  
Turing Award 1977

Associated with the functional style of programming is an algebra of programs whose variables range over programs and whose operations are combining forms. This algebra can be used to transform programs and to solve equations whose “unknowns” are programs in much the same way one transforms equations in high school algebra.

Bird and Meertens  
[3, 14]

- put this approach into practice with the formalism that took their names
- also called Squiggol
Join-lists

Finite sequence of values having the same type
- empty list $[]$
- singleton list $[a]$ (for an element $a$)
- concatenation $x ++ y$ of two lists $x$ and $y$.

Properties of ‘constructors’
- $[]$ unit for $++$
- $++$ associative operation.

map
\[
\begin{align*}
\text{map } f [] &= [] \\
\text{map } f [a] &= [f a] \\
\text{map } f (x ++ y) &= (\text{map } f x) ++ (\text{map } f y)
\end{align*}
\]

reduce
\[
\begin{align*}
\text{reduce } \odot [a] &= [a] \\
\text{reduce } \odot (x ++ y) &= (\text{reduce } \odot x) \odot (\text{reduce } \odot y)
\end{align*}
\]

Homomorphic Functions

A function $h$ is $\odot$-homomorphic
if for all lists $x$ and $y$,
\[
h(x ++ y) = (h x) \odot (h y)
\]
for a binary operation $\odot$.

$(\text{img}(h), \odot, [a])$ is a monoid
\[
a = h x = h([] ++ x) = ([]) \odot (h x) = i_0 \odot a
\]
\[
a = h x = h(x ++ []) = (h x) \odot ([]) = a \odot i_0
\]
$\odot$ is associative ...

Variant with an additional condition
$h : A \to B$
$(B, \odot, i_B)$ est un monoïde

Lists and Parallelism (distributed memory)

Usual List
\[
[a_0; \ldots; a_{n-1}] + + + \ldots + + [a_0; \ldots; a_{n-1}]
\]

Distributed List
Processor $1 \ldots$ Processor $i \ldots$ Processor $p$
\[
[a_0; \ldots; a_{n-1}] + + + \ldots + + [a_0; \ldots; a_{n-1}]
\]

Homomorphisms and Data Parallelism

Theorem (First homomorphism theorem)

If $h$ is $\odot$-homomorphic then $h = (\text{reduce } \odot) \circ (\text{map } f)$.

Processor$1 \ldots$ Processor$1 \ldots$ Processor$p$
\[
h([a_0; \ldots; a_{n-1}] + + + \ldots + + [a_0; \ldots; a_{n-1}])
\]
\[
= (\text{map phase})
\]
\[
\text{reduce } \odot ([a_0; \ldots; a_{n-1}] + + + \ldots + + [a_0; \ldots; a_{n-1}])
\]
\[
= (\text{local reduce phase})
\]
\[
= (\text{global reduce phase})
\]
\[
\odot \ldots \odot f a_0
\]
**Third Homomorphism Theorem (1)**

**Definition (⌥-leftwards and ⌥-rightwards functions)**

A function \( h \) is ⌥-leftwards for an operation ⌥, if for every list \( x \) and every element \( a \),

\[
h ([a] + + x) = a \oplus h(x).
\]

A function \( h \) is ⌥-rightwards for an operation ⌥, if for every list \( x \) and every element \( a \),

\[
h (x + + [a]) = (h(x)) \oplus a.
\]

**foldr and foldl**

The unique function ⌥-leftwards (resp. ⌥-rightwards), is usually written \( \text{foldr} \oplus e \) (resp. \( \text{foldl} \oplus e \)) where \( e = h[] \).

**Properties of foldl and foldl**

\[
\begin{align*}
\text{foldr} \oplus e (x ++ y) &= \text{foldr} \oplus e (y) + (x) & \text{[fr]} \\
\text{foldl} \oplus e (x ++ y) &= \text{foldl} \oplus e (y) + (x) & \text{[fl]}
\end{align*}
\]

---

**Third Homomorphism Theorem (2)**

**Theorem (Third homomorphism theorem)**

Let be \( h \) a function, ⌥ and ⌥ binary operations. If \( h \) is ⌥-leftwards and ⌥-rightwards, then \( h \) is ⌥-homomorphic.

**Approach to parallelisation**

- Third theorem ⇒ homomorphic
- First theorem ⇒ reduce ⌥ map
- Then replace by parallel versions of map and reduce

**Problem**

The third homomorphism theorem is not constructive

---

**Third Homomorphism Theorem (3)**

**Definition (Weak right inverse)**

Let be \( h \) a function on lists.

\( h' \) is a weak right inverse of \( h \) iff for every list \( x \),

\[
h x = h (h'(h x)).
\]

**Lemma (Existence of a weak right inverse)**

For a computable function \( h \) whose domain is countable, there exists a function \( h' \) such that: for all \( x \), \( h (h'(h x)) = h x \).

**Proof.**

\( h' \) may be partial. For compute \( h' a \), let’s enumerate the elements of the domain of \( h \) and stop when we meet a \( x \) such that \( h x = a \) and return \( x \). This process terminates for all elements of the image of \( h \), by may not terminate otherwise.

---

**Third Homomorphism Theorem (4)**

**Theorem (Weak third homomorphism theorem)**

Let be \( h \) a function, \( h' \) a weak right inverse of \( h \), ⌥ and ⌥ binary operations. If \( h \) is ⌥-leftwards and ⌥-rightwards, then \( h \) is ⌥-homomorphic where \( a \oplus b = h((h' a) + + (h' b)).\)
Type Classes (1)

**Monoid**

- **Class LeftNeutral** `(op : B → A → A) e` :=
  
  ```coq
  left_neutral : \forall a, op e a = a.
  ```

- **Class RightNeutral** `(op : A → B → A) e` :=
  
  ```coq
  right_neutral : \forall a, op a e = a.
  ```

- **Class Neutral** `(op : A → A → A) e` :=
  
  ```coq
  neutral_left Neutral := LeftNeutral op e;
  neutral_right Neutral := RightNeutral op e.
  ```

- **Class Associative** `(op : A → A → A) e` :=
  
  ```coq
  associative : \forall x y z, op (op x y) z = op x (op y z).
  ```

**Class Monoid** `(op : A → A → A) e` :=

```coq
monoid_assoc : Associative op;
monoid_neutral : Neutral op e.
```

Type Classes (2)

**Instances**

- **Program Instance** `plus_O_monoid : Monoid plus 0`.

**Next Obligation**

```coq
constructor. intros. now rewrite plus_assoc.
```

**Qed.**

**Next Obligation**

```coq
constructor. trivial.
```

**Next Obligation**

```coq
constructor. intros. now rewrite plus_n_O.
```

**Qed.**

- **Instance** `app_nil_monoid (A : Type) : Monoid (@app A)`.

**Admitted.**

Type Classes (3)

**Instance Resolution**

- **Definition** `reduce (op : A → A → A) (Monoid A op e) (list A) :=
  
  fold_left op l e`.

- **Definition** `result1 := reduce plus [0;1;2]`.

**Eval compute in result1.**

```coq
(= 3 : nat)
```

- **Definition** `result2 := reduce (@app Set) [nat];[bool];[list nat]`.

**Eval compute in result2.**

```coq
(= [nat; bool; list nat]; list Set)
```

- **Require Import** `ZArith`.

- **Fail Definition** `result2 := reduce Zplus ([0;1;2];[nat;bool;list nat]`.

**(= Error: Cannot infer the implicit parameter e of reduce. )**
### Homomorphic Function in Coq

A function $h$ is $\oplus$-homomorphic

\[
\text{if for all lists } x \text{ and } y,
\]

\[
h (x \oplus y) = (h x) \oplus (h y)
\]

(2)

for a binary operation $\oplus$.

### Coq

**Class** Homomorphic `{h:list A -> B} `{op:B -> B -> B} :=

`{ homomorphic : \forall x y, h (x++y) = op (h x) (h y) }.`

### List Data Structure

- Usual definition of Coq standard library
- foldr and foldl are List.fold_left and List.fold_right.

### Homomorphisms in Coq

#### Modelled as Type Classes

- One class per equation

**Class** Homomorphism `{f'h : list A -> B} `{f:A -> B} :=

`{ homomorphism_f : \forall (a:A), h [a] = f a }.`

**Class** Homomorphism `{h:list A -> B} `{op:B -> B -> B} `{f:A -> B} `{LMonoid B op e} `{Homomorphic A B h op} `{Homomorphism_f A B h f} :=

`{ homomorphism_nil : h [] = e }.`

### Image of $h$ in Coq

**Definition** img `{h:list A -> B} := { b:B | \exists l, h l = b }.**

- For P:A -> Prop, expression `{ a:A | P a } is a notation of sig P.

**Inductive** sig `{A:Type} `{P:A -> Prop} : Type :=

exist : \forall x : A, P x -> sig P.

- ... but we then prove equality of some proof terms.
First Homomorphism Theorem in Coq

**Definition** hom_to_map_reduce \[\{A B : \text{Type}\} \{h : \text{list} A \rightarrow B\} \{\text{H:Homomorphic} A B h \text{ op}\} : \text{list} A \rightarrow \text{img} h := (\text{reduce op}) \circ (\text{List.map} (\text{fun} x \rightarrow h(x))).

**Theorem** first_homomorphism_theorem \[\{\text{H:Homomorphic} A B h \text{ op}\} : \forall L, h L = \text{of_img} (\text{hom_to_map_reduce} h L).

reduce

**Definition** reduce \[\{\text{op:A} \rightarrow A \rightarrow A\} \{\text{m: Monoid} A \text{ op e}\} := \text{fun} l \rightarrow \text{fold_left} \text{ op} l \text{ e}.

Third Homomorphism Theorem in Coq

**Problem**

- In Coq only terminating functions,
- Lemma 5 cannot be proved.

**Weak third homomorphism theorem**

**Instance** third_homomorphism_theorem_right_inverse \[\{\text{h:list} A \rightarrow B\} \{\text{inv:Right_inverse} A B h h'\} \{\text{Hl:Leftwards} A B \text{ h pl e}\} \{\text{Hr:Rightwards} A B \text{ h opr e}\} : \text{Homomorphic} h (\text{fun} l r \rightarrow h'(h'(l)++(h' r))).

with

**Class** Right_inverse \[\{\text{h:list} A \rightarrow B\} \{\text{h':B} \rightarrow \text{list} A\} := \{\text{right_inverse:} \forall l, h l = h(h'(l))\}.

Maximum Prefix Sum

**mps**

\[
\text{mps} [1, 2, -1, 2, -1, -1, 3, -4] = 5
\]

**Specification**

\[
\text{mps} = \text{maximum} \circ (\text{map sum}) \circ \text{prefix}
\]

**Code and Paper**

- Applications/Mps.v
- Bsml/Applications/BsmlMps.v
- Paper:[13]
**Count**

### Specification

**Definition** `count_spec` (A: Type) (p: A \rightarrow bool) (l: list A) : nat :=

\[ \text{List.length (List.filter p l)}. \]

### Goal

- A slightly more efficient version
- A parallel version

### File

- Applications/Count.v

---

**Automatic Parallelization**

**Section** Count.

**Variable** A : Type.

 Variable predicate : \{ pred : A \rightarrow bool & \{ a : A | pred a = true \} \}.

(* * * * * Version where the result is a scalar *)

**Definition** `par_count_img` : par(list A) \rightarrow img (count_spec predicate) :=

Eval sydpacc in

parallel (hom_to_map_reduce (count_spec predicate)).

**Definition** `par_count` : par(list A) \rightarrow nat :=

Eval sydpacc in of_img (Stdlib.parfun of_img (@proj1_sig _ _)) (par_count_img).

(* * * * * Version where the result is a parallel vector *)

**Definition** `par_count_img'` :=

Eval sydpacc in

parallel (hom_to_map_reduce (count_spec predicate)).

**Definition** `par_count'` : par(list A) \rightarrow par nat :=

Eval sydpacc in

(parfun (@compose f int of nat)) (of_img(parfun (@compose f nat of int)@proj1_sig _ _)) (par_count_img').

End Count.

---

**Outline**

1. Introduction and Motivation
2. Functional Programming in Coq
3. Proofs in Coq
4. Theory of Lists and Parallelisation in Coq
5. Conclusion

---

**Extraction and Execution**

Module `Primitives` with type `par` = Bsml.par = struct

let `bsp p` = nat of int then

type `par` = Bsml.par = struct

let `mkpar f` = Bsml.mkpar (fun i! f (nat of int i)) then

let `apply` = Bsml.apply then

let `put` = Stdlib.parfun of_img then

let `proj` = compose (Stdlib.proj) int of nat then

End `Primitives`.
**Bulk Synchronous Parallel ML (BSML) in Coq**

**Parallel vectors**
- In Coq: `Parameter par : Type -> Type.`
- Informally: \( (a_0, \ldots, a_{p-1}) \)

**Primitives**
- `Parameter mkpar : (processor -> A) -> par A`
- `Parameter apply : par(A -> B) -> par A -> par B`
- `Parameter proj : par A -> processor -> A`
- `Parameter put : par(processor -> A) -> par(processor -> A)`

\[
\begin{align*}
\text{mkpar } f &= (f 0, \ldots, f (p - 1)) \\
\text{apply } (f_0, \ldots, f_{p-1}) (v_0, \ldots, v_{p-1}) &= (f_0 v_0, \ldots, f_{p-1} v_{p-1}) \\
\text{proj } (v_0, \ldots, v_{p-1}) &= (\lambda i. v_i) \\
\text{put } (f_0, \ldots, f_{p-1}) &= (\lambda j. f_j 0, \ldots, f_j (p - 1))
\end{align*}
\]

---

**An Overview of the Parallelization Mechanism (1)**

**Type correspondence**
- `join` with `join` is surjective

---

**An Overview of the Parallelization Mechanism (2)**

**Function correspondence**
- `join f` with `join`

**Class FunCorr**
- `{ACorr : TypeCorr A Ap join_A}`
- `{BCorr : TypeCorr B Bp join_B}`
- `{fCorr : @FunCorr A Ap join_A ACorr B Bp join_B BCorr f fp}`

**Variants:**
- sequential input types
- sequential output types

**Instances:**
- compositions
- "algorithmic skeletons"

---

**An Overview of the Parallelization Mechanism (3)**

**map and reduce skeletons**
- `Program Definition par_map '(f:A -> B)'(v:par(list A)) : par(list B) := parfun (List.map f) v.`
- `Program Definition par_reduce 'op:A -> A'}(m:LMonoid A op e)(v:par(list A)) : A := reduce op (List.map (proj (parfun (reduce op) v)) processors).`

**Parallelization**
Outline

1. Introduction and Motivation
2. Functional Programming in Coq
3. Proofs in Coq
4. Theory of Lists and Parallelisation in Coq
   - An Overview of Bird Meertens Formalism
   - Theory of Lists in Coq
   - Applications
   - Automatic Parallelization
5. Conclusion
6. Bibliography

Summary

Summary

Coq for mechanising the design of programs in the Bird Meertens Formalism tradition
- Automatic parallelisation with type classes
- extraction of parametric modules appliqued to parallel implementations of BSML in OCaml, C and MPI
- Small: 1900 LoC Coq, 600 LoC OCaml, 120 LoC C

To Learn More about Coq

Online
- Ilya Sergey, Programs and Proofs, http://ilyasergey.net/pnp/

Books
- A. Chlipala, Certified Programming with Dependent Types, MIT Press, 2013

To Learn More about SyDPaCC

Parallelization of Homomorphism in Coq
- Application: Maximum Prefix Sum
- Paper: [13]

Generate-Test-and-Aggregate
- Specifications: generator + tester + aggregator
- Interactive Theorem Proving 2014: [7]

Bulk Synchronous Parallel Homomorphisms
- Capture a larger class of BSP algorithms
- Papers: [8, 12, 11]
To Learn More about SyDPaCC: Poster Session!

Formalization of a Big Graph API in Coq
  Presenter: Jolan Philippe

A Verified Parallel Implementation of Frequent Itemset Mining
  Presenter: Chris Whitney

Collaborative Work with avec

- Dr. Wadoud Bousdira (Université d'Orléans)
- Dr. Sylvain Dailler (KUT & Université d'Orléans)
- Dr. Kent Emoto (Kyushu University of Technology)
- Pr. Zhenjiang Hu (National Institute of Informatics)
- Dr. Sylvain Jubertie (Université d'Orléans)
- Dr. Louis Gesbert (OCamlPro)
- Hideki Hashimoto (The University of Tokyo)
- Dr. Joerrey Légaux (Université d'Orléans)
- Dr. Kento Matsuizaki (Kochi University of Technology)
- Dr. Virginia Niculescu (Babes-Bolyai University of Cluj-Napoca)
- Simon Robillard (Chalmers)
- Pr. Masato Takeichi (The University of Tokyo)
- Dr. Julien Tesson (Université Paris-Est Créteil)

Outline

1. Introduction and Motivation
2. Functional Programming in Coq
3. Proofs in Coq
4. Theory of Lists and Parallelisation in Coq
   - An Overview of Bird Meertens Formalism
   - Theory of Lists in Coq
   - Applications
   - Automatic Parallelization
5. Conclusion
6. Bibliography

Bibliographie I


