





Sous la direction de Pr. Jean-Paul Bodeveix

# Générateur de code Coq

Joas Kinouani
Abakar Mahamat Sougui

# Plan

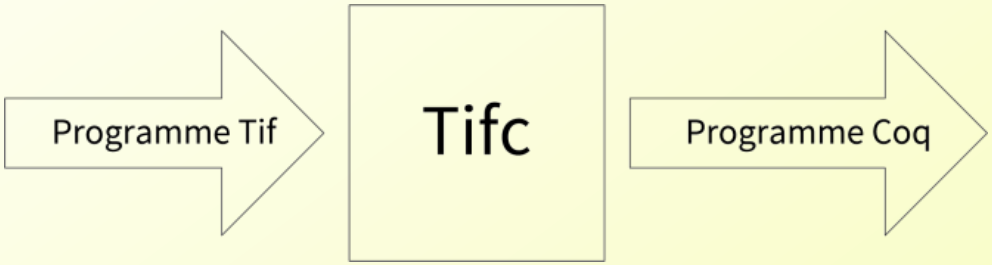
- Contexte
  - L'IRIT
  - Equipe ACADIE
  - Sujet du TER
- Analyseur syntaxique pour types inductifs
  - Syntaxe
  - Représentation d'un type inductif
  - Analyse syntaxique avec Camlp5
- Générateur de code Coq
  - Introduction à Coq
  - Types (co)inductifs
  - Observateurs
  - Accesseurs
  - Sous-termes
  - Preuve de transitivité
- Conclusion

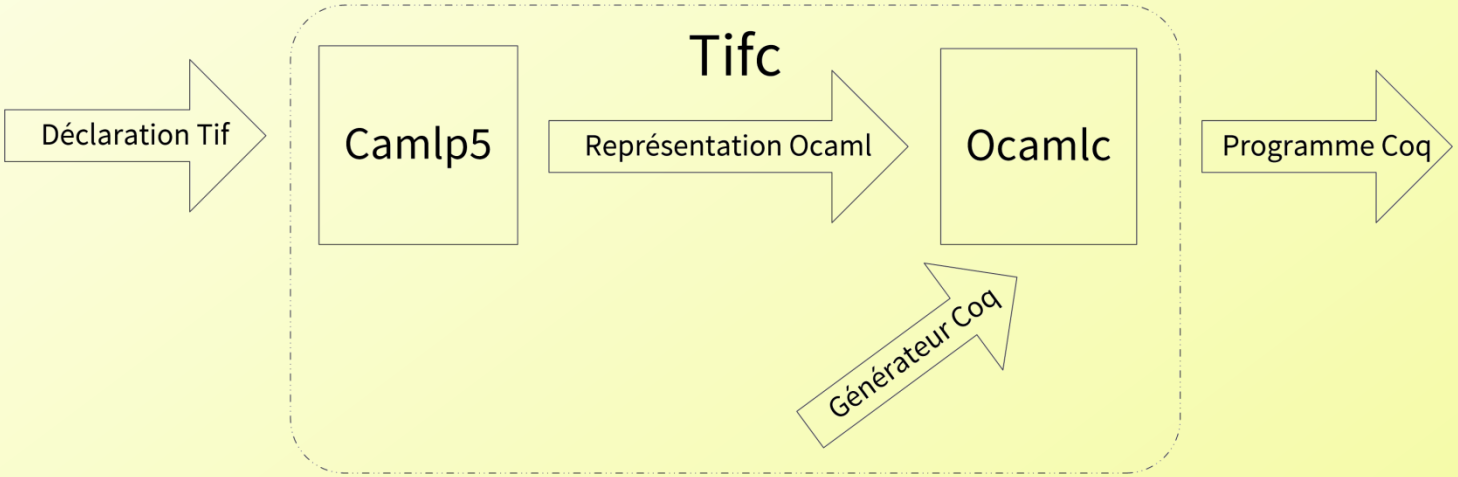
# Contexte


250 (enseignant-)chercheurs
244 doctorants
14 post-doctorants
43 ingénieurs
19 équipes

17 membres permanents
23 membres non-permanents

# Contexte


Programme Tif
Tifc
Programme Coq


Déclaration Tif
Tifc
Camlp5
Représentation Ocaml
Ocamlc
Programme Coq
Générateur Coq

# Plan

- Contexte
  - L'IRIT
  - Equipe ACADIE
  - Sujet du TER
- Analyseur syntaxique pour types inductifs
  - Syntaxe
  - Représentation d'un type inductif
  - Analyse syntaxique avec Camlp5
- Générateur de code Coq
  - Introduction à Coq
  - Types (co)inductifs
  - Observateurs
  - Accesseurs
  - Sous-termes
  - Preuve de transitivité
- Conclusion

# Syntaxe du langage

```
type bit =
      Zero
    | One

type tree (t : Type) =
      Leaf
    | Node (left : tree t) (root : t) (right : tree t)
```

*déclaration* → '**type**' *identificateurMin paramètresOpt* '=' *constructeurs*
*paramètresOpt* → $\epsilon$ | *paramètre paramètresOpt*
*paramètre* → '(' *identificateurMin* ':' *typage* ')'
*constructeurs* → *constructeur* | *constructeur* '|' *constructeurs*

# Syntaxe du langage

```
type bit =
      Zero
    | One

type tree (t : Type) =
      Leaf
    | Node (left : tree t) (root : t) (right : tree t)
```

*constructeur* → *identificateurMaj arguments*
*arguments* → $\epsilon$ | *argument arguments*
*argument* → '(' *identificateurMin* ':' *typage* ')'

# Syntaxe du langage

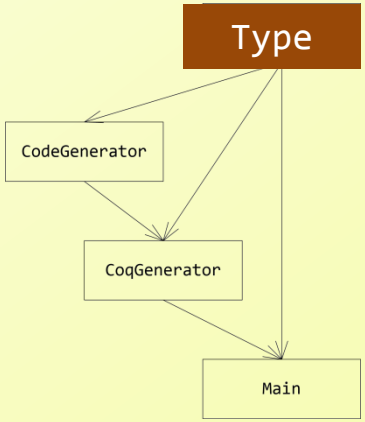
```
type tree (t : Type) =
      Leaf
    | Node (left : tree t) (root : t) (right : tree t)

type tree (t : Type) =
      Leaf
    | Node (node : tree t * t * tree t)

type comparator (t : Type) =
      Comparator (comparison : t -> t -> nat)
```

*typage* → *type* | *parenthésé* | *produit* | *signatureDeFonction*
*type* → *identificateur généricté*
*généricité* → $\epsilon$ | *identificateur généricité*
*parenthésé* → '(' *typage* ')'
*produit* → *typage* '*' *typage* | *typage* '*' *produit*
*signatureDeFonction* → *typage* '->' *typage*

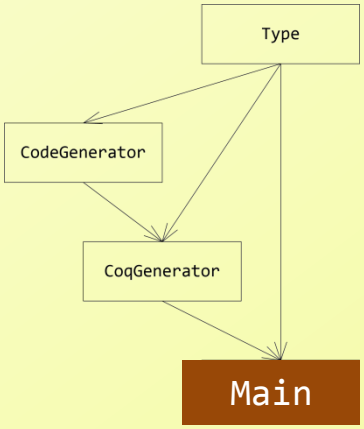
# Représentation d'un type inductif




```
type declaration_t = Declaration of string * parameter_t list * constructor_t list
type parameter_t = Parameter of string * expression_t
type constructor_t = Constructor of string * argument_t list
type argument_t = Argument of string * expression_t
type expression_t =
      Type of string * string list
    | Parenthesized of expression_t
    | Product of expression_t list
    | FunctionSignature of expression_t * expression_t
```

# Analyse syntaxique avec Camlp5

```
type tree (t : Type) =
      Leaf
    | Node (left : tree t) (root : t) (right : tree t)
```


```
Declaration ("tree",
    [Parameter ("t", Type ("Type", []))],
    [Constructor ("Leaf", []);
     Constructor ("Node",
        [Argument ("left", Type ("tree", ["t"]));
         Argument ("root", Type ("t", []));
         Argument ("right", Type ("tree", ["t"]))])])
```

# Plan

- Contexte
  - L'IRIT
  - Equipe ACADIE
  - Sujet du TER
- Analyseur syntaxique pour types inductifs
  - Syntaxe
  - Représentation d'un type inductif
  - Analyse syntaxique avec Camlp5
- Générateur de code Coq
  - Introduction à Coq
  - Types (co)inductifs
  - Observateurs
  - Accesseurs
  - Sous-termes
  - Preuve de transitivité
- Conclusion

# Introduction à Coq

- Assistant de preuve
  - Certification de logiciels
    - Protocoles réseaux
    - Systèmes critiques
    - Compilateur C

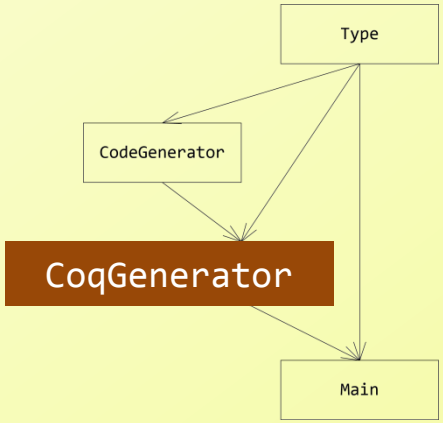
# Ocaml & Coq

| Ocaml | Coq |
|---|---|
| `type` | `(Co)Inductive` |
| `let` | `Definition` |
| `let rec` | `Fixpoint` (plus puissant et plus rigoureux) |
| Absent | Logique |
| Absent | Preuve |

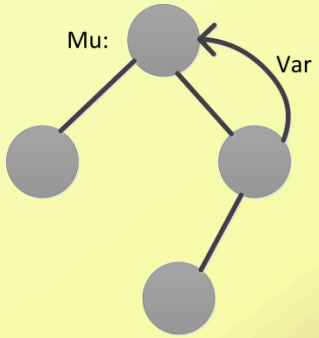
# Types (co)inductifs

```
Declaration ("tree",
    [Parameter ("t", Type ("Type", []))],
    [Constructor ("Leaf", []);
     Constructor ("Node",
        [Argument ("left", Type ("tree", ["t"]));
         Argument ("root", Type ("t", []));
         Argument ("right", Type ("tree", ["t"]))])])
```

CoqGenerator



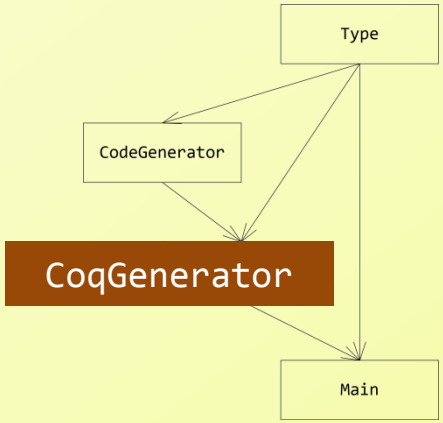

```
Parameter Label : Type.
Inductive tree (t : Type) :=
      Leaf
    | Node (left : tree t) (root : t) (right : tree t)
    | Mu (tag : Label) (data : tree t)
    | Var (tag : Label).
```



# Observateurs

```
Declaration ("tree",
    [Parameter ("t", Type ("Type", []))],
    [Constructor ("Leaf", []);
     Constructor ("Node",
        [Argument ("left", Type ("tree", ["t"]));
         Argument ("root", Type ("t", []));
         Argument ("right", Type ("tree", ["t"]))])])
```

CoqGenerator




```
Definition isNode {t: Type} (x : tree t) : Prop :=
    match x with
        Node _ _ _ => True
      | _ => False
    end.
```

# Observateurs

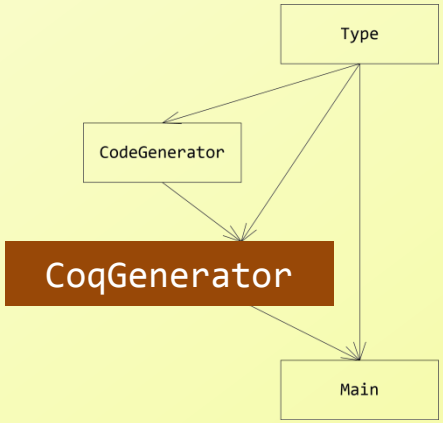
- Exemples

```
Coq < Compute isNode (Leaf nat).
 = False : Prop
Coq < Definition node := Node nat (Leaf nat) 53 (Leaf nat).
Node is defined
Coq < Compute isNode node.
 = True : Prop
```

# Accesseurs

```
Declaration ("tree",
    [Parameter ("t", Type ("Type", []))],
    [Constructor ("Leaf", []);
     Constructor ("Node",
        [Argument ("left", Type ("tree", ["t"]));
         Argument ("root", Type ("t", []));
         Argument ("right", Type ("tree", ["t"]))])])
```

CoqGenerator




```
Definition getNodeLeft {t: Type} (x : tree t) : isNode x -> tree t :=
    match x with
        Node leftSubtree _ _ => fun pre => leftSubtree
      | _ => fun pre => match pre with end
    end.
```

# Accesseurs

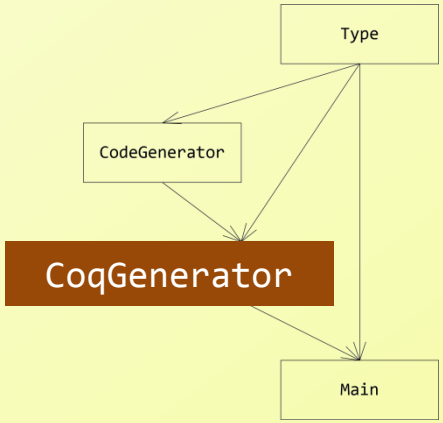
- Exemples

```
Coq < Definition node := Node nat (Leaf nat) 53 (Leaf nat).
Node is defined
Check I.
I : True
Coq < Compute getNodeLeft node I.
= Leaf nat : tree nat
Coq < Compute getNodeLeft (Leaf nat) I.
Error
```

# Sous-termes

```
Declaration ("tree",
    [Parameter ("t", Type ("Type", []))],
    [Constructor ("Leaf", []);
     Constructor ("Node",
        [Argument ("left", Type ("tree", ["t"]));
         Argument ("root", Type ("t", []));
         Argument ("right", Type ("tree", ["t"]))])])
```

CoqGenerator




```
Inductive subterm {t: Type} (x : tree t) : tree t -> Type :=
      SubSelf : subterm x x
    | SubMu : forall tag data, subterm x (Mu t tag data) -> subterm x data
    | SubLeft : forall lt r rt, subterm x (Node t lt r rt) -> subterm x lt
    | SubRight : forall lt r rt, subterm x (Node t lt r rt) -> subterm x rt.
```

# Sous-termes

$$\frac{x : tree\ t}{subterm\ x\ x}\ SubSelf$$

$$\frac{x : tree\ t \ \wedge subterm\ x\ (Node\ t\ lt\ r\ rt)}{subterm\ x\ lt}\ SubLeft$$

$$\frac{x : tree\ t \ \wedge subterm\ x\ (Node\ t\ lt\ r\ rt)}{subterm\ x\ rt}\ SubRight$$

# Conclusion

# Questions ?