L'Atelier FoCaL

Renaud Rioboo

http://www-spi.lip6.fr/~rr/

UNIVERSITE PIERRE & MARIE CURIE
LA SCIENCE A PARIS

LIP6

http://www.upmc.fr/ http://www.lip6.fr/

Grenoble, Décembre 2005

Le Projet FoCaL

Collaboration:

- Équipe SPI: T. Hardin (coordinatrice), M. Jaume, R. Rioboo
- INRIA: D. Doligez, P. Weis
- CPR, CEDRIC: V. Donzeau, C. Dubois, D. Delahaye, O. Pons

Sponsors

- INRIA CFC, CNRS Math-STIC, Modulogic, MKM Network
- Alliance project (Univ Kent)
- MKM and Calculemus Interest Groups

Un survol de FoCaL

Le système manipule des entités qui appartiennent à des collections (`e in c`) qui sont décrites par des espèces (`c is s`)

Les espèces (`species`)

- spécifient des opérations (`sig`) et leur propriétés (`property`)
- implémentent des algorithmes (`let`) et montrent leur correction (`theorem`)
- Instancient des structures de données (`rep`)
- Héritent les composants d'autres espèces (`inherits`)

Les collections (`collection`) instancient des espèces concretes (`implements`) et donnent une interface utilisateur abstraite.

Les entités sont manipulées avec des appels de méthode (`c!m`)

L'utilisateur final de FoCaL

manipule des entités à l'aide d'appels de méthode:

```
let rec fib(n in integer) in integer =
  if integer!is_zero(n)
  then integer!zero
  else
    if integer!is_one(n)
    then integer!one
    else
      let pred n = integer!moins(n,integer!one) in
      let m = pred(n) in
      integer!plus(#fib(m), #fib(pred(m)));;
```

Terminaison? Langage de colle sans peuve!

L'apprenti programmeur FoCaL

Crée de nouvelles collections

```
collection naturals implements petits_entiers;;
let rec fib(n in naturals) in naturals =
  if naturals!is_zero(n)
  then naturals!zero
  else if naturals!is_one(n)
       then naturals!one
       else let pred n = #non_failed(naturals!moins(n,naturals!one)) in
            let m = pred(n) in naturals!plus(#fib(#m), #fib(pred(m)));;
let fib_32 =
  let mult = integer!mult in
  let two = integer!plus(integer!one,integer!one) in
  let four = mult(two,two) in let sixteen = mult(four,four) in
  fib(mult(sixteen,two));;
```

Le programmeur FoCaL

Encapsule ses algorithmes dans des espèces

```
species my_new_resultant(a_ring is integral_domain,
                         some_pols is univariate_polynomials(a_ring)
  inherits basic_objects =
  let resultant = ...;;
  theorem structural_result ...;;
```

Ajoute des fonctionalités

```
species modular_ring(a_ring is euclidean_domain,
                     some_val in ring)
  inherits commutative_ring =
  rep = a_ring ;;
  property clean_rep ...
```

Le développeur FoCaL

introduit des nouveaux types

```
type distr_rep a_ring a_mon_ordering =
  Zero in distr_rep(a_ring, a_mon_ordering) ;
  NonZero in a_ring -> a_mon_ordering ->
    distr_rep(a_ring, a_mon_ordering) -> distr_rep(a_ring, a_mon_ordering) ;(*
```

Construit de nouvelles espèces

```
species poly_test(a_ring is ring, a_deg is monomial_ordering)
   inherits ring =
  rep = distr_rep(a_ring, a_deg) ;
  let rec coefficient(p,d) = match p with
    | #Zero -> a_ring!zero
    | #NonZero(cc,dd,rp) ->
      if a_deg!lt(dd,d) then a_deg!zero
      else if a_deg!lt(d,dd) then self!coefficient(rp,d)
           else cc ;
```

Correction du Code ?

Approche mathématique:

```
let rec fib_bad n =
   if n <= 1 then n else
      fib_bad(n-1) + fib_bad(n-2);;
```

mauvaise complexité

Théorème:

Soit F la fonction Fibonacci, si $f_{n-1} = F(n-1)$ et $f_n = F(n)$ alors

$n \geq 1 \Longrightarrow F(n+1) = f_{n-1} + f_n$

Preuve: evident!

Corollaire: la fonction `fib_bet` est corrècte?

```
let fib_bet =
  let rec aux n =
    if (n=0) then (0,1) else
     let (f,f') = aux (n-1) in
     (f',f+f')
  in function n ->
     let (f,f') = aux n in f;;
```

La preuve n'est lisible que par des humains!

Embarquer les preuves avec le code?

```
species fibonacci inherits basic_object () =

  let rec fib_bad n = ...;
  local let rec fib_aux n = ...
  let fib_good n = #first(!fib_aux n);

  theorem fib_aux_is_correct:
    all n in integer, n >= 0 -> #first(!fib_aux(n)) = !fib_ba
    proof: assumed ;

  theorem fib_good_is_correct:
    all n in integer, n >= 0 -> #first(!fib_aux(n)) = !fib_ba
    proof:
      by !fib_aux_is_correct def !fib_good ;
end
```

Propriétés et Théorèmes

- Les propriétés spécifient des preuves:

```
species additive_monoid inherits set_with_zero =
  sig plus in self -> self -> self;
```

L'élément 0 est neutre pour + dans un monmoide additif:

```
  property zero_is_neutral :  all x in self,
    !equal(!plus(x,!zero),x)
      and
    !equal(!plus(!zero,x),x) ;
```

- Ce sont les axiomes d'une théorie abstraite:

L'élément neutre de + est unique:

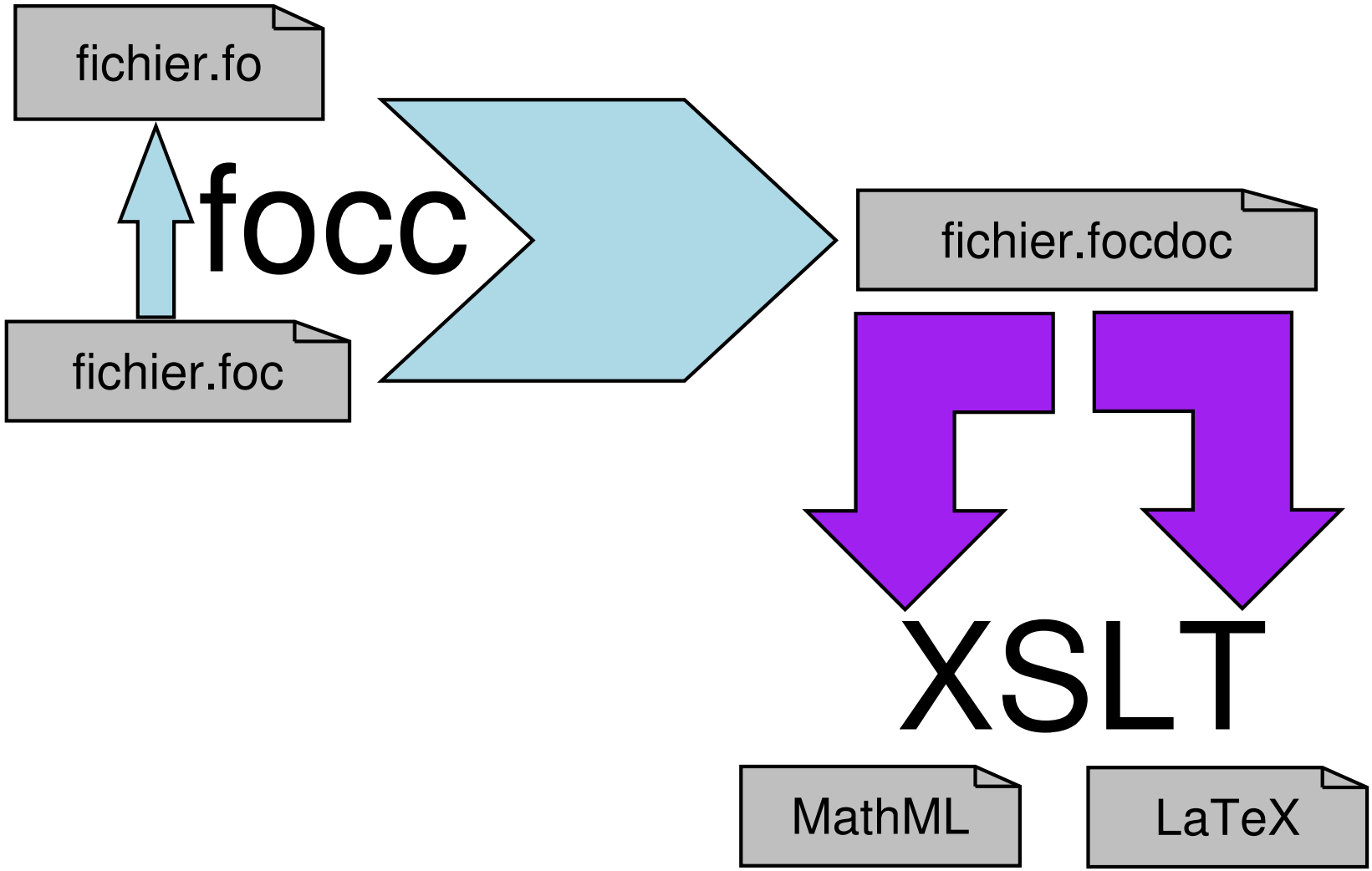
```
  theorem zero_is_unique :  all o in self,
    (all x in self,!equal(x,!plus(x,o))) ->
      !equal(o,!zero)
```

La librairie FoCaL

12000 lignes de sources FoCaL produisent 40000 lignes de Coq et 9500 lignes d'Ocaml.

- Domaines de base du Calcul Formel: Entiers, arithmetiques modulaires ...
- arithmetiques polynomiales:
  - Representations distribuées (creuses): des listes triées de couples degré, coefficient.
  - Representations récursives: une variable principale et les coefficients sont des polynômes en les autres variables.
- Algorithmes pour:
  - Calcul de résultant.
  - Factorisation des polynômes en une variable sur les corps finis.

Outil de Documentation: FoCDoC



/home/rr/Focal/focal/tools/FoCDoc/index.html

File Edit View Go Bookmarks Tabs Help

Home Back Forward Up Stop Reload History Print Find New Window 75%

file:///home/rr/Focal/focal/tools/FoCDoc/index.html

FoCDoC generated files

additive_law.xml	We specify some properties of additive laws which are commonly commutative and have neutral element named 0
additive_law_partial.xml	We specify some properties of additive laws which are commonly commutative and have neutral element named 0
additive_law_rr.xml	We specify some properties of additive laws which are commonly commutative and have neutral element named 0
big_integers.xml	Ocamls bigint in FoC
eat.xml	eat.foc describes some basics topological algebra in order to reflect part o EAT and Kenzo to FoC
finite_factorize.xml	This implements factorization using Quite-Naudin algorithm Note [rr]: this seems to have problems in characteristic 2
finite_fields.xml	finite fields

Multiplicative laws

File Edit View Go Bookmarks Tabs Help

Home Back Forward Up Stop Reload History Print Find New Window 75%

file:///home/rr/Focal/focal/tools/FoCDoc/multiplicative_law.xml

Multiplicative laws

The FoC Project

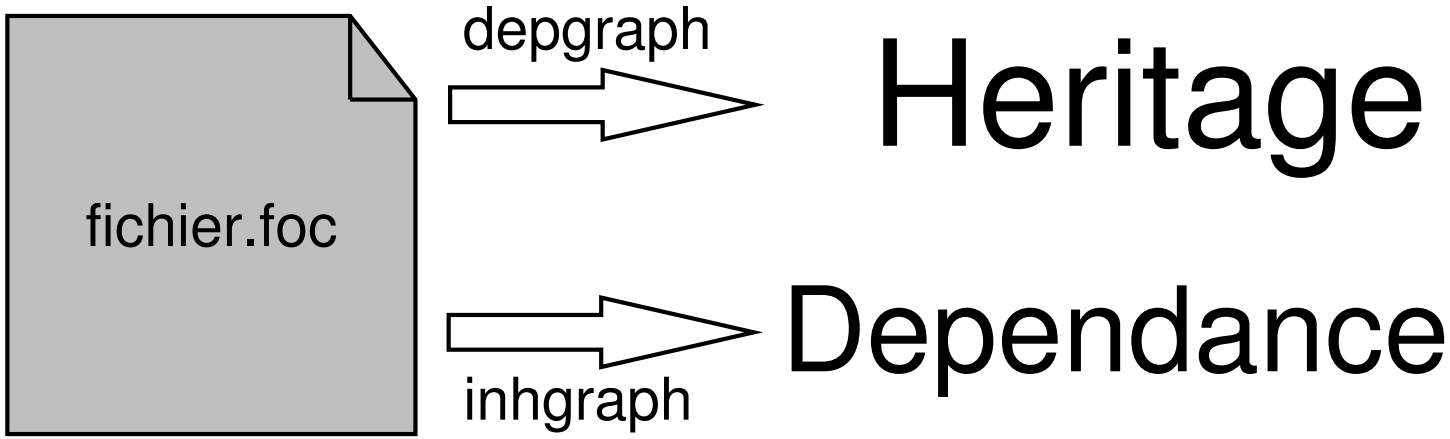
Common properties of multiplicative laws

Load	sets_orders - additive_law
Open	sets_orders - additive_law
List of species	setoid_with_one - semi_group - monoid - commutative_monoid - regular_monoid - division_monoid - factorization_monoid

back to index of files

Species: `setoid_with_one`

Outils d'analyse



Le corps d'une méthode (`let`, `proof`, `rep`) m peut dépendre d'une autre méthode m': dans sa définition ou dans sa déclaration (type, ennoncé: `sig`, `property`)

- decl-dépendance: seule la *déclaration* de m' est importante
- def-dependance: il faut connaître la *définition* de m' pour typer m.

Les `let` n'introduisent pas de def-dépendances.

Dépendance dans les preuves

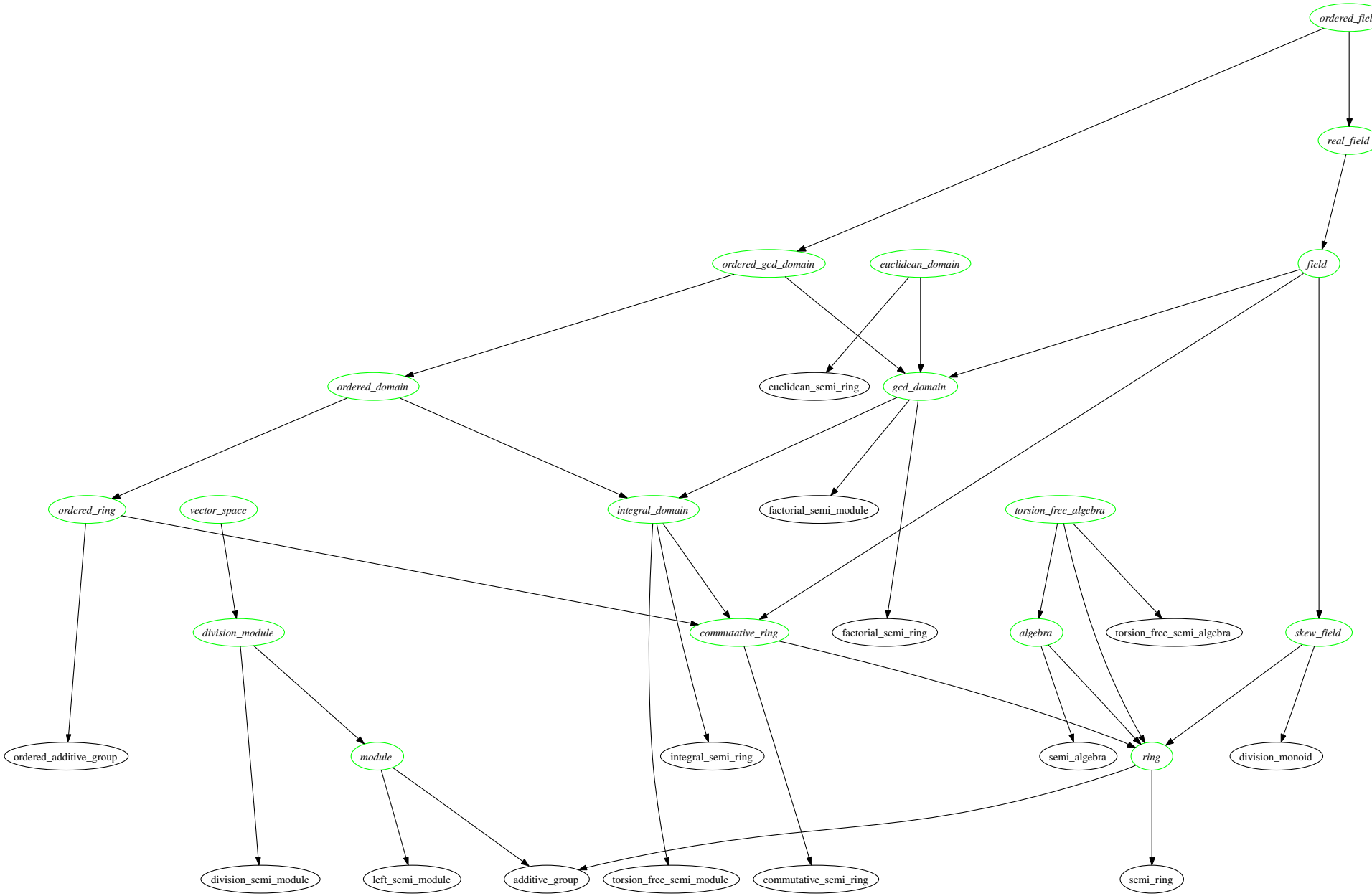
Les preuves peuvent introduire des def-dépendances:

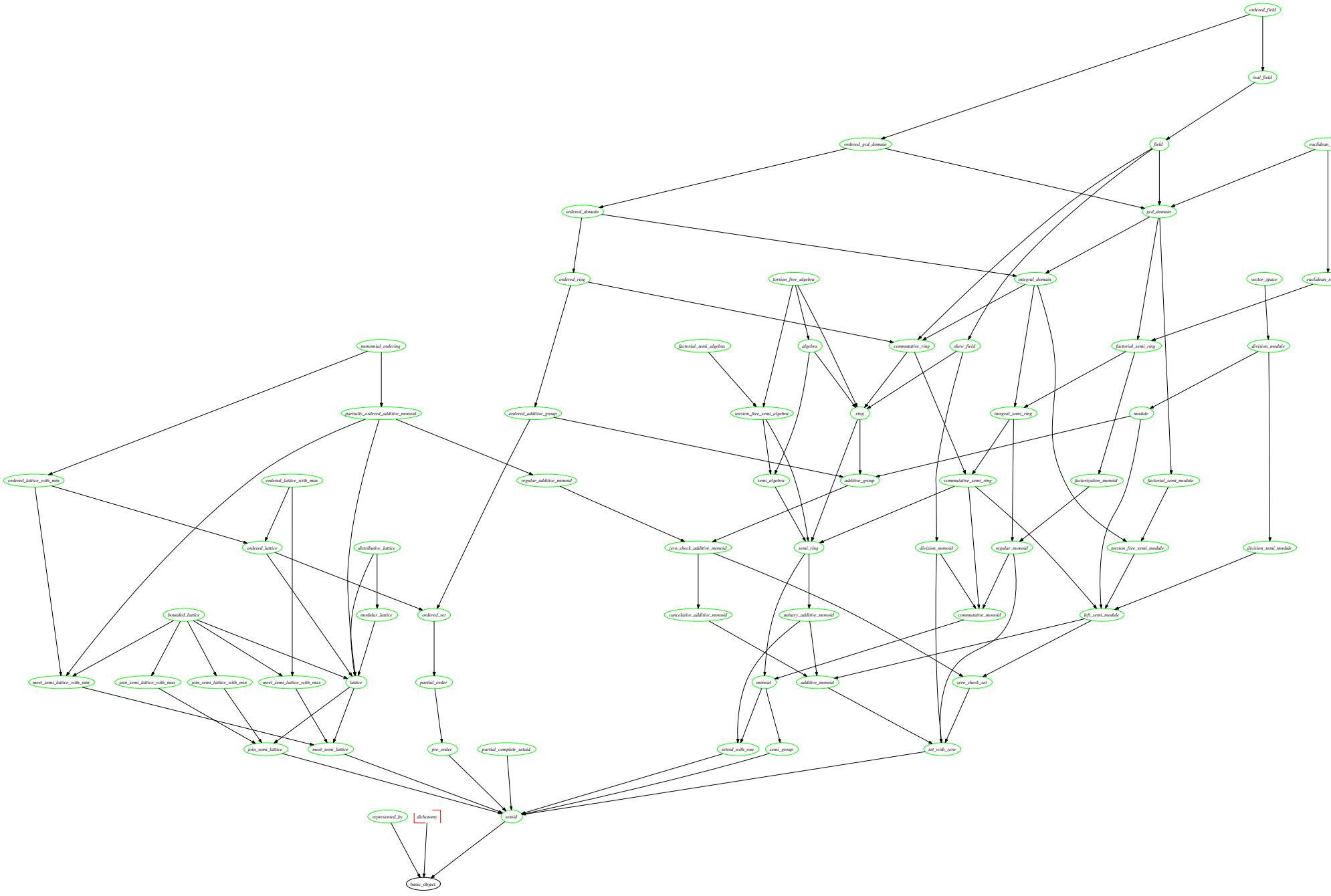
- on utilise la définition de m' (`unfold` m').
  Une redéfinition de m' entraine l'effacement de toutes les preuves qui dépendent de m'
- on énonce une propriété dont l'énoncé dépend de la valeur de m':

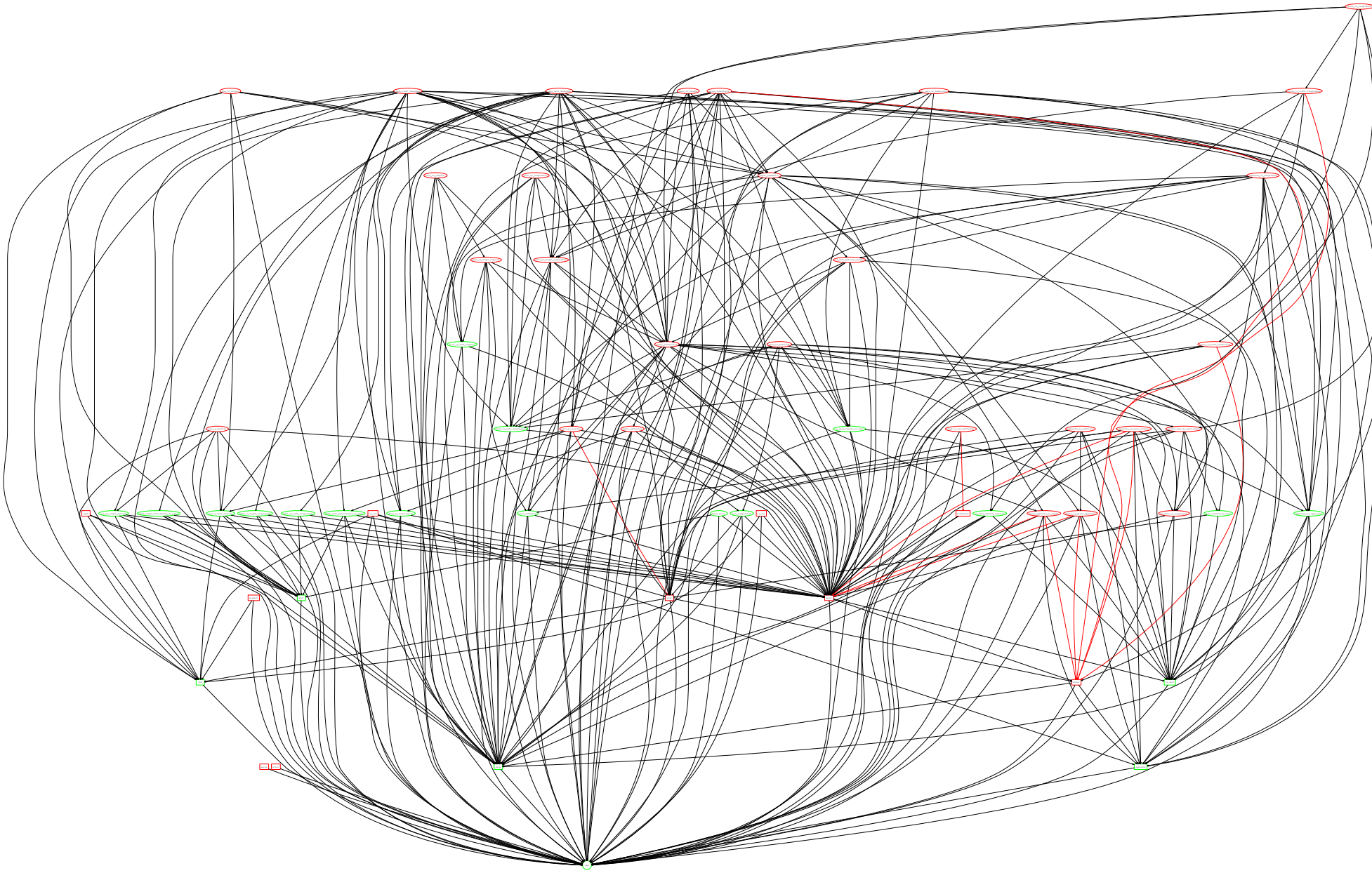
```
rep = nat

property foo:  all x in self, x > 0
```

  Les def-dépendances dans un type empêchent de créer l'interface correspondante. Interdit

L'Héritage multiple conduit à des effacement de preuves.


ordered_fiel
real_field
ordered_gcd_domain
euclidean_domain
field
ordered_domain
euclidean_semi_ring
gcd_domain
ordered_ring
vector_space
integral_domain
factorial_semi_module
torsion_free_algebra
division_module
commutative_ring
factorial_semi_ring
algebra
torsion_free_semi_algebra
skew_field
ordered_additive_group
module
integral_semi_ring
semi_algebra
ring
division_monoid
division_semi_module
left_semi_module
additive_group
torsion_free_semi_module
commutative_semi_ring
semi_ring


ordered_field
real_field
ordered_gcd_domain
field
euclidean_
ordered_domain
gcd_domain
ordered_ring
torsion_free_algebra
integral_domain
vector_space
euclidean_s
monomial_ordering
factorial_semi_algebra
algebra
commutative_ring
skew_field
factorial_semi_ring
division_module
partially_ordered_additive_monoid
ordered_additive_group
torsion_free_semi_algebra
ring
integral_semi_ring
module
ordered_lattice_with_min
ordered_lattice_with_max
regular_additive_monoid
semi_algebra
additive_group
commutative_semi_ring
factorization_monoid
factorial_semi_module
ordered_lattice
distributive_lattice
zero_check_additive_monoid
semi_ring
division_monoid
regular_monoid
torsion_free_semi_module
division_semi_module
bounded_lattice
modular_lattice
ordered_set
cancelative_additive_monoid
unitary_additive_monoid
commutative_monoid
left_semi_module
meet_semi_lattice_with_min
join_semi_lattice_with_max
join_semi_lattice_with_min
meet_semi_lattice_with_max
lattice
partial_order
monoid
additive_monoid
zero_check_set
join_semi_lattice
meet_semi_lattice
pre_order
partial_complete_setoid
setoid_with_one
semi_group
set_with_zero
represented_by
dichotomy
setoid
basic_object



Production de preuves, Organisation

- Le compilateur `focc` produit:
  - Des sources Ocaml sources contenant les signatures et les definitions,
  - Des sources Coq qui contiennent en plus les propriétés et les preuves.
- Les sources Coq sont un cadre pour faire les preuves:
  - un chaptitre Coq pour chaque espèce
  - une espèce est encodée comme un enregistrement avec types dépendants.
- Les preuves Coq doivent être produites
  - `assumed`
  - par l'utilisateur (obsolete) où
  - en utilisant Zenon

Anciennes preuves

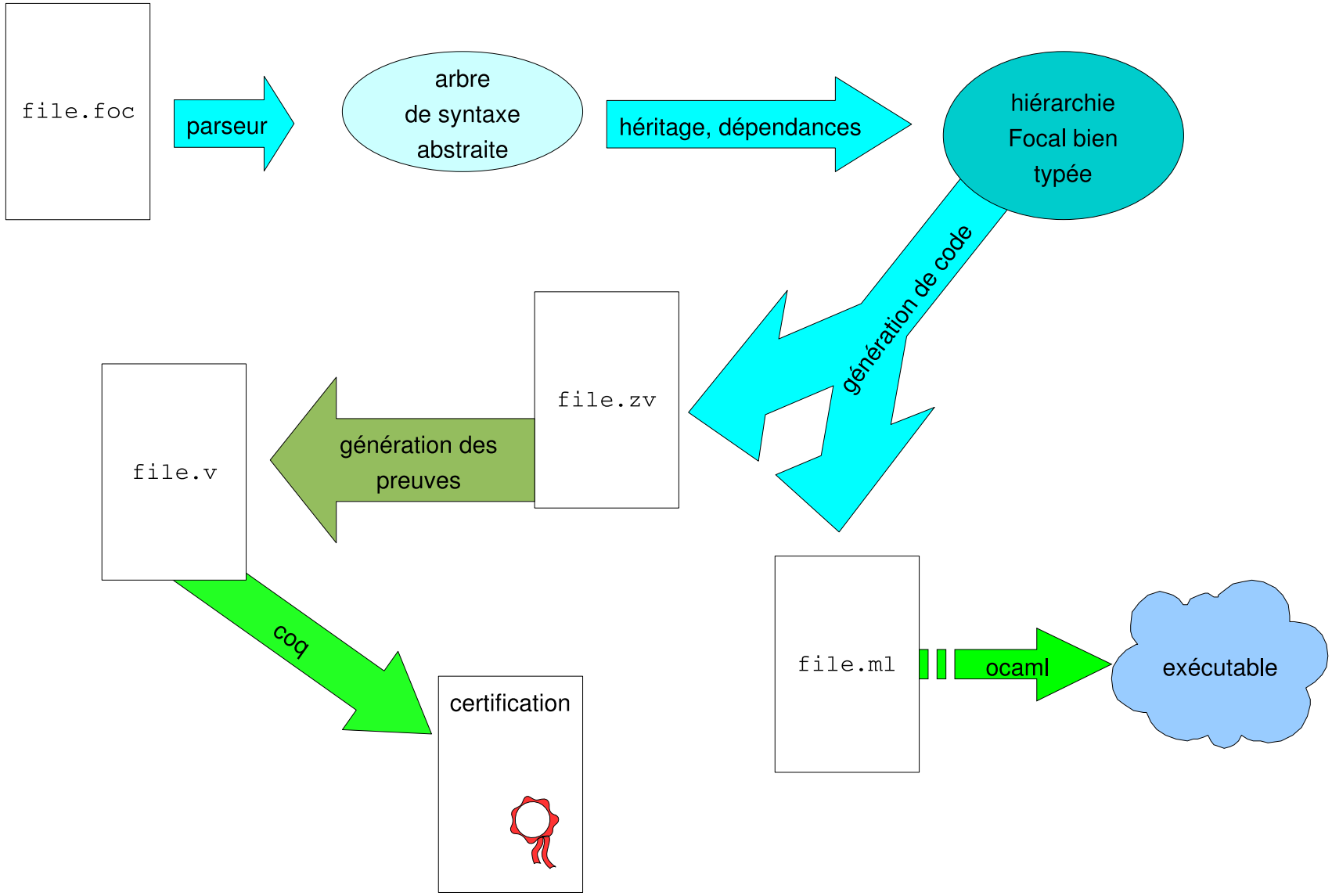
On montre `zero_is_unique`. Introduction des dépendances:

```
decl :  zero;
decl :  equal_symmetric; ...
```

Les preuves sont transmises à Coq telles quelles:

```
{*
intros.
...
apply (abst_equal_symmetric (abst_plus abst_zero o) o).
assumption.
generalize (H abst_zero).
*};
```

Connaissance de la traduction, style impératif, l'énoncé n'apparait pas.


file.foc
parseur
arbre
de syntaxe
abstraite
héritage, dépendances
hiérarchie
Focal bien
typée
génération de code
file.zv
génération des
preuves
file.v
coq
certification
file.ml
ocaml
exécutable

Le démonstrateur Zenon

- logique classique (un axiom de plus à Coq, `Classical`)
- Techniques de tableaux pour les preuves
- traduction en Coq, chaque preuve introduit une section

```
Section additive_monoid__zero_is_unique.
Variable abst_T: Set.
Variable abst_equal:  (abst_T)-> (abst_T)-> bool__t.
...
```

En cours

- automatiser plus de preuves
- ajouter des principes d'induction

Comment écrire une preuve?

Leslie Lamport 1991

```
proof:
```

- Une preuve est un arbre
  - Introduction des noms:

    ```
    <1>1 assume o in self
    ```

  - On introduit et on nomme les hypothèses:

    ```
    H1:  all x in self, !equal(x,!plus(x,o))
    ```

  - Énoncé de la conclusion:

    ```
    prove !equal(o,!zero)
    ```

  - On insère une arbre de niveau 2 qui démontre le résultat!
  - La conclusion en découle:

    ```
    <1>2 qed;
    ```

Preuves hierarchiques

- Une preuve s'énonce en (**by**) en utilisant d'autres énoncé:

  ```
  prove !equal(o,!zero)
  ```

  – o est neutre à droite:

    ```
    <2>1 prove !equal(!zero,!plus(!zero,o))
         by <1>:H1
    ```

    on peut faire référence aux hypothèses nomées plus haut dans l'arbre

  – 0 est neutre à gauche, on combinne les égalités:

    ```
    <2>2 prove !equal(o,!zero)
         by <2>1, !zero_is_neutral,
            !equal_transitive, !equal_symmetric
    ```

  – Implicitement on conclu d'eaprès tous les points précédents:

    ```
    <2>3 qed by <2>2
    ```

  Les preuves sont plus déclaratives, les énoncés sont dans la preuve.

Des preuves plus concrètes

Dans un ensemble on a deux opérations naturelles ”=” et “≠” liées:

```
sig equal in self -> self -> bool;
property equal_reflexive :  ...
let different (x,y) = basics#not_b(!equal(x,y));
theorem same_is_not_different :  all x y in self,
  !equal(x,y) -> not(!different(x,y))
  proof:
    def !different;
```

Zenon peu utiliser la **def**inition de “≠”.

Les `by` introduisent des def-dépendances, les `def` des def-dépendances.

Avantages

- Plus proche du raisonnement mathématique
- Sources plus lisibles
- Ne nécessite pas de maipulation explicite du terme de preuve
- les propriétés sont plus souvent réutilisables.
- L'arbre de syntaxe abstraite est connu de `focc` (dépendances)
- Chaque preuve est indépendante
- Zenon peut être remplacé par d'autres outils (déduction modulo, CIME, . . .)
- Le support `tptp` autorise l'utilisation en réseau (MathWeb)

Les treillis

650 lignes de FoCaL, 525 lignes d'Ocaml, 9500 lignes de Coq

Un inf demis treillis est un ensemble avec une opération ($\cap$, `inf`)

- associative (`inf_is_associative`)
- idempotente (`inf_idempotent`)
- commutative (`inf_commutes`)

binaire (`inf_is_congruent`) qui induit un ordre:

- `let order_inf(x,y) = !equal(x, !inf(x, y));`
-
```
theorem order_inf_reflexive:  all x y in self,
  !equal(x,y) -> !order_inf(x,y)
  proof:
    by !inf_idempotent def !order_inf;
```

```
theorem order_inf_is_transitive :  all x y z in self,
  !order_inf(x,y) -> !order_inf(y,z) -> !order_inf(x,z)
  proof:
  <1>1 assume x y z in self
          H1:  !order_inf (x, y) H2:  !order_inf (y, z)
       prove !order_inf (x, z)
       <2>0 prove !equal (x, !inf (x, y))
            by <1>:H1 def !order_inf
       <2>1 prove !equal (x, !inf (x, !inf (y, z)))
            by <2>0, <1>:H2, !inf_is_congruent, !equal_transitive
            def !order_inf
       <2>2 prove !equal (x, !inf (!inf (x, y), z))
            by <2>1, !inf_is_associative, !equal_transitive, !equal_symmetric
       <2>3 prove !equal (x, !inf (x, z))
            by <2>2, <2>0, !equal_symmetric, !equal_transitive, !inf_is_congru
       <2>4 qed by <2>3 by def !order_inf
  <1>2 qed;
```

- `inf` defini bien un infimum:

```
theorem order_inf_is_infimum:  all x y i in self,
  !order_inf(i,x) -> !order_inf(i,y) -> !order_inf(i,!inf(x,y))
  proof:
    by !inf_is_associative,
      !equal_symmetric, !equal_transitive,
      !inf_is_left_congruence
    def !order_inf;
```

- Dans un sup demis treillis on a une opération `sup`, un ordre `order_sup` et des propriétés analogues.
- Les lois d'absorption montrent que l'ordre est unique:

```
theorem order_sup_refines_order_inf:  all x y in self,
  !order_inf(y,x) -> !order_sup(x,y)
  proof:
    by !inf_commutes, !inf_absorbes_sup, !sup_is_congruent,
       !equal_symmetric, !equal_transitive
    def !order_inf, !order_sup;
```

Des programmes concrets?

Les entités (`self`) sont codées par des valeurs de (`rep`).

Établir des invariants sur les valeurs de `self`?

```
species represented_by(s is basic_object)
  inherits basic_object =
    rep = s;
```

fonctions de coercion:

```
    let from_rep(e in s) in self = #foc_error("from_rep:  not implem
    let to_rep (x in self) in s = x;
```

L'utilisateur établit ses invariants:

```
    sig represents_some_self in s -> Prop;
    letprop correct_representation(x) = !represents_some_self(!to_re
end
```

Ensembles quotients

Les mathématiques ont des quotients en plus des sommes et des produits.

E un ensemble (`setoid`), "$\equiv$" une relation d'équivalence, implémenter $E/\equiv$?

En Calcul formel on utilise un représentant distingué de la classe d'équivalence. Une notion de "réduction":

Un projecteur est une opération idempotente p de E dans E:

```
species projections(s is setoid) inherits basic_object =
  sig project in s -> s;
  property project_is_congruent:  all x y in s,
    s!equal(x,y) -> s!equal(!project(x),!project(y));
  property project_is_projection :  all x in s,
    s!equal(!project(x),!project(!project(x)));
end
```

Implémentation

Les ensembles quotients utilisent des projecteurs:

```
species quotient_set(s is setoid, proj is projections(s))
  inherits represented_by(s), setoid =
  letprop represents_some_self(r) = s!equal(!from_rep(r),r);
  let from_rep(x) = proj!project(x);
  let equal(x,y) = s!equal(x,y);
  proof of equal_reflexive =
    by s!equal_reflexive def !equal;
  proof of equal_symmetric =
    by s!equal_symmetric def !equal;
  proof of equal_transitive =
    by s!equal_transitive def !equal;
```

Correction

On produit toujours des éléments réduits

```
let element = proj!project(s!element);
theorem sample_is_reduced :  !correct_representation(!element)
  proof:
  <1>1 prove !represents_some_self(!element)
        <2>1 prove s!equal(!element,!from_rep(!element))
             by s!equal_reflexive, proj!project_is_projection
             def !element, !from_rep
        <2>f qed by <2>1,
                    s!equal_symmetric, s!equal_reflexive, s!equal_t
                    def !from_rep, !represents_some_self
  <1>2 qed by <1>1 def !to_rep, !correct_representation
  ;
```

Invariants?

On ne peut pas écrire

```
property my_invariant:  all x in self,
  s!equal(x,proj!project(x)
```

Car l'énoncé introduit une def-dépendance entre `self` et `rep`.

Vers un mécanisme d'invariants? Internaliser,

```
proof:
  <1>1 prove all x in self, correct_representation(x) ->
                s!equal(x,proj!project(x)
  <1>2 prove all x in self, s!equal(x,proj!project(x)
                correct_representation(x) ->
  <1>3  une vraie proriété utilise <1>:1 et <1>:2
       qed
```

Perspectives

- mathématiques effectives:
  - Topologie algébrique,
  - Théories des anneaux de Prufer.
- utilisation non mathématique:
  - sécurité d'un aeroport
  - politiques de sécurité
- compilation vers d'autres cibles:
  - C++, C
  - F#
- génération automatique de tests:
  - les propriétés servent de spécifiations,
  - production de valeurs pour tester.

http://focal.inria.fr/

References

[BHH+99] S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain, and R. Rioboo. On the way to certify computer algebra systems. In *Proceedings of the Calculemus workshop of FLOC'99 (Federated Logic Conference, Trento, Italie)*, volume 23 of *ENTCS*. Elsevier, 1999.

[BHR01] Sylvain Boulmé, Thérèse Hardin, and Renaud Rioboo. Some hints for polynomials in the FoC project. In Steve Linton and Roberto Sebastiani, editors, *Calculemus 2001 Proceedings*, June 2001.

[MP03] Manuel Maarek and Virgile Prévosto. Focdoc: The documentation of foc. In *Calculemus 2003 Proceedings*, September 2003.

[PD02] Virgile Prevosto and Damien Doligez. Algorithms and proof inheritance in the foc language. *Journal of Automated Reasoning*, 29(3-4):337–363, December 2002.

[PJ03] Virgile Prévosto and Mathieu Jaume. Making proofs in a hierarchy of mathematical structures. In *Calculemus 2003 Proceedings*, September 2003.

[Pro03] FoC Project. *The FoC System Reference Manual Version 0*. SPI LIP6, 2003.

[RH04] Renaud Rioboo and Thérèse Hardin. Les objets des mathématiques. *RSTI - L'objet*, Octobre 2004.