Making proofs in a hierarchy of mathematical structures

A case study within the Computer Algebra System FoC

Virgile Prevosto[1,2]
Mathieu Jaume[1]

[1] SPI – LIP6 – University Paris 6

[2] Projet Moscova – INRIA

Proving properties

Proving properties ...

- easy for small programs
- more complicated when programming "in the large": must ensure the coherence of the whole software architecture.
- coherence of a hierarchical architecture:
  - inheritance lookup
  - dependencies analysis
  - type checking ...
- few logical frameworks with inheritance mechanisms.

Proving properties in the FoC's hierarchy of mathematical structures

- The computer algebra system FoC:
  - Inheritance and refinements mechanisms.
  - Specification of algorithms
  - Proofs of correctness.
- Proofs:
  - verified by Coq.
  - Coq environment (dependencies, inheritance)...
  - ...obtained by a static analysis of the FoC code.

Problem of "proofware engineering"

- The FoC system allows to do the proofs just when you decide ... of course before using the unit being built.
- January 2003: training a group of twenty undergraduate students to do small proofs in FoC.
- Issue: Place of a proof in the hierarchy
  - make it as soon as possible?
    - May be working at a too abstract level.
    - Will the proof be used in a concrete implementation ?
  - make it as late as possible?
    - Duplication of proofs.
    - Separation between code and proofs.

Overview of FoC

The FoC Language

Specification and Implementation of Algebraic Structures

`species`

Method	Declared	Defined
Programming	`sig`	`let [rec]`
Certification	`property`	`theorem`
Carrier	`rep`	`rep = int`

Methods

```
species setoid =
 rep;
 sig eq in self -> self -> bool;
 let neq (x,y) = #notb(self!eq(x,y));
 property eq_refl: all x in self, self!eq(x,x);
 theorem neq_nrefl: all x in self, not(self!neq(x,x))
    proof: def neq;
           decl eq_refl;
           ... ;
end
```

Inheritance

```
species partial_order inherits setoid =
sig leq in self->self->bool;
property leq_reflexive: all x in self, self!leq(x,x);
let lt(x,y) = #and_b(self!leq(x,y),#not_b(self!eq(x,y)));
theorem lt_is_not_leq : all x y in self,
  ((self!lt(x,y) -> (self!leq(x,y) and self!neq(x,y))) and
  (self!leq(x,y) -> (self!lt(x,y) or self!eq(x,y))))
  proof:
    def: lt neq;
    decl: leq;
    ... ;
end
```

Parameters

```
species cartesian (a is setoid,  b is setoid) inherits setoid =
  rep  = (a!rep * b!rep);
```

`a` and `b` represent *complete* species, implementing *any* structure deriving from `setoid`.

interface: abstract view	**collection**: concrete implementation
`rep;`	`rep = int`
sig `eq in [...]`	**let** `eq = [ ... ]`
property `eq_refl:  [...]`	**theorem** `eq_refl:  [...]`
	`proof:  [...]`

Dependencies analysis

Dependencies

Find calls to self!y...

- in the body of a method x.
- in its type.
- avoid cycles out of a `let rec`.
- decl-dependency: we must have the type of y
- def-dependency: We must have its type and its definition.

The different kinds of dependencies

	decl	def
body	`let x = self!y`	`Unfold self!y`
type	`property y_spec:` `P(self!y)`	`rep = nat;` **Forbidden** `property foo:` `∀ x ∈ self,` `nat_plus(x,1)>0`

Def-dependency in a type we cannot build the interface

Dependencies upon the Carrier

- decl-dependency: a sub-expression has type `self`:

```
sig plus in self->self->self;
let f = fun (x in self) -> self!plus(x,x);
```

- def-dependency: unification between `self` and its concrete representation.

```
rep = nat;
let f = fun (x in self) -> #nat_plus(x,x)
```

Inheritance and Dependencies

Dependencies analysis occurs after inheritance resolution ➜ normal form of a species.

```
species a = rep;  sig x in self;  let y = self!x;  end

species b = rep;  let x = self!y;  sig y in self;  end

species c inherits a,b =  rep = int;  end
```

inherits

```
let y = self!x
```

inherits

```
let x = self!y
```

Def-dependencies and Inheritance

- x redefined ➡ erase *all* proofs that def-depends upon x...
- ... ➡ user must reprove them using the new definition of x.

On the Coq side

Method Generator

```
species s =
  let x = def_x_s;
  let y =
    def_y_s[self!x];
end
species t
 inherits s =
  let x = def_x_t;
end
```

```
Chapter s.
  Definition gen_x_s:= def_x_s.
  Local x_s:= gen_x_s.
  Definition gen_y_s:=
λx.def_y_s[selxf!x].
  Local y_s:= (gen_y_s x_s).
Chapter t.
  Definition gen_x_t:= def_x_t.
  Local x_t:=gen_x_t.
  Local y_t:=(gen_y_s x_t)
```

- A generator for each method defined in **the body** of s
- *late binding* ➔ choosing the good generator
- A local definition of each defined method (inherited or not)

- Abstraction with respect to the methods y depends upon

Generators and Def-dependencies

```
Section setoid__neq_nrefl.
  Variable abst_T: Set.
  Variable abst_eq: (abst_T)-> (abst_T)-> bool__t.

  Local abst_neq: (abst_T)-> (abst_T)-> bool__t:=
     (setoid__neq abst_T abst_eq   ).
  Variable abst_eq_refl: (x: abst_T) (Is_true (abst_eq  x x)).

  Definition setoid__neq_nrefl:  (x: abst_T)
  ~( (Is_true ((abst_neq  x x)) )).
  ...
End setoid__neq_nrefl.
```

Redefinitions and Proofs erasure

Erasure of proofs : an example (1)

```
species partial_order inherits setoid =
sig leq in self-> self->bool;
let lt(x,y) = #and_b(self!leq(x,y),#not_b(self!eq(x,y)));
theorem lt_is_not_leq : all x y in self,
  (( self!lt(x,y) -> ( self!leq(x,y) and self!neq(x,y))) and
  ( self!leq(x,y) -> ( self!lt(x,y) or self!eq(x,y))))
   proof: def: lt neq; decl: leq; ... ;
...
```

- `lt` is coded but not specified
- the only possibility is that `lt_is_not_leq` def-depends upon `lt` and `neq`.

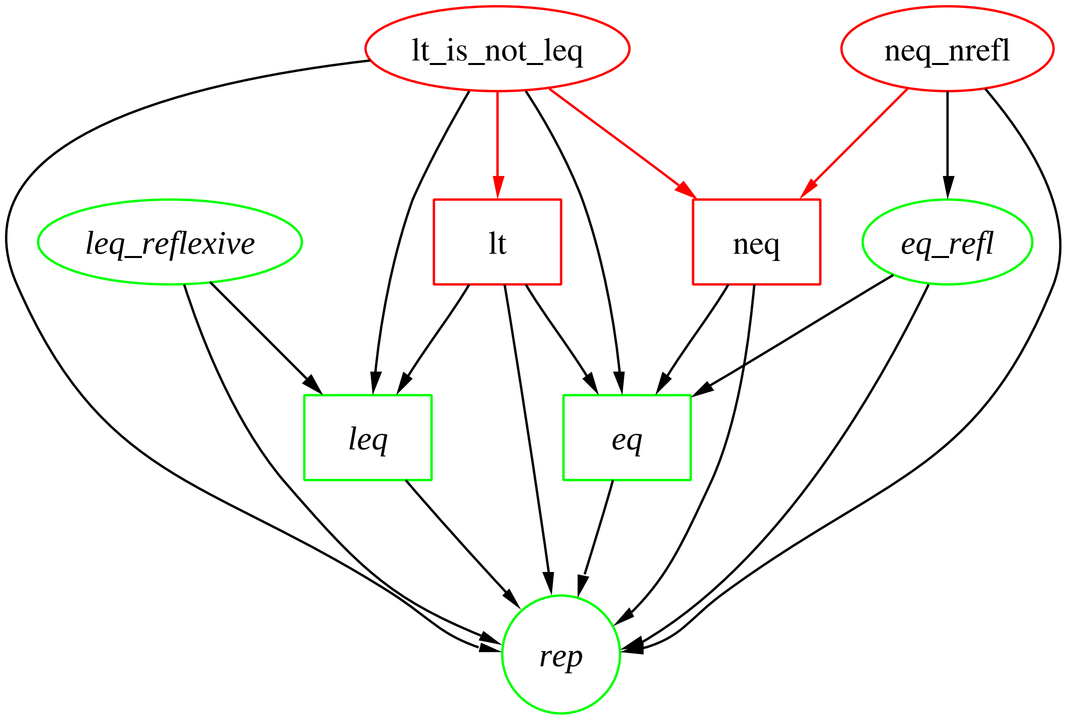
Erasure of proofs : an example (2)

```
species ordered_set inherits partial_order =
 property total_order : all x y in self,
  #or_b( self!leq(x,y), self!leq(y,x));
 let eq(x,y) = #and_b( self!leq(x,y), self!leq(y,x));
 proof of equal_reflexive = decl: leq_reflexive;def: equal;...;
 let lt(x,y) = #not_b( self!leq(y,x));
 proof of lt_is_not_leq = ... ;
...
```

- `lt` redefined
- `lt_is_not_leq` must be reproved

Compiling `ordered_set`

Dependencies graphs



Dependence graph for partial_order

Is the proof in partial_order really useful ?

... It depends on the hierarchy

If many species or collections inherit from `partial_order`, without redefining `lt`	➔	Yes !
If no species (except `ordered set`) or collection inherits from `partial_order`	➔	Not really

But further developments of the library may use partial orderings

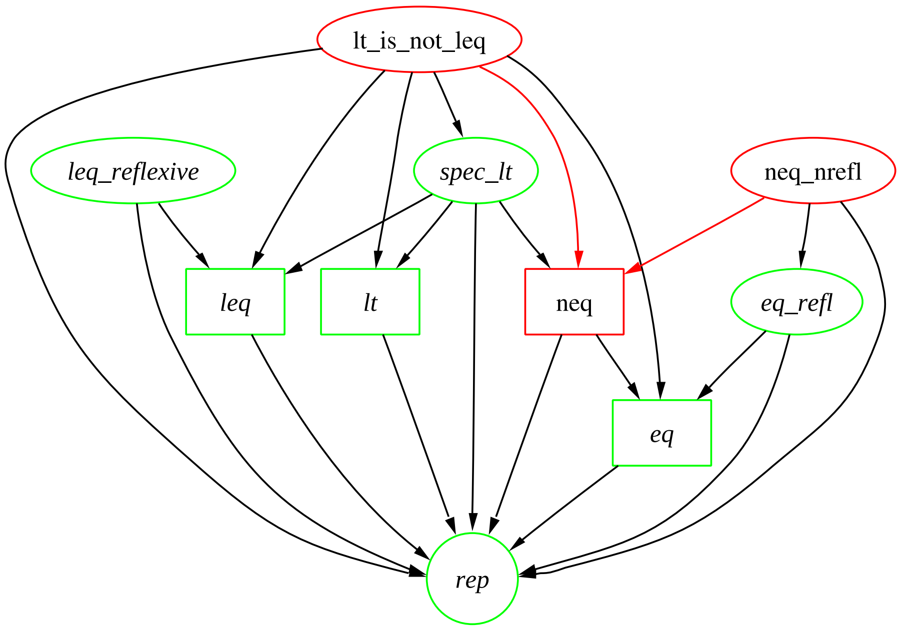
...

First issue: specification by properties (1)

Giving a true specification of `lt`: a declaration of `lt` and a property characterizing it. (the computational part is not considered).

```
species partial_order  inherits setoid =
  sig leq in  self-> self->bool;
  sig lt  in self-> self->bool;
  property spec_lt :  all x y  in self,
   !lt(x,y) <-> (!leq(x,y) and !neq(x,y));
  theorem lt_is_not_leq :  all x y  in self,
    ((!lt(x,y) -> (!leq(x,y)  and !neq(x,y)))  and
    (!leq(x,y) -> (!lt(x,y)  or !eq(x,y))))
     proof: def:neq; decl: spec_lt leq;...;
...
```

First issue: specification by properties (2)



Dependence graph for partial_order

Later, at each (re)definition of `lt`, as in the species `ordered_set`, we will have to prove the property `spec_lt`, but not `lt_is_not_leq`

Second issue (1)

- closer to an implementation.
- may be useful when the first definition of `lt` was already given and used, properties and proofs being added later

Starting from the definition of `lt`:

- introduce `spec_lt` as a new theorem
- prove `lt_is_not_leq` using only a decl-dependency upon `spec_lt`

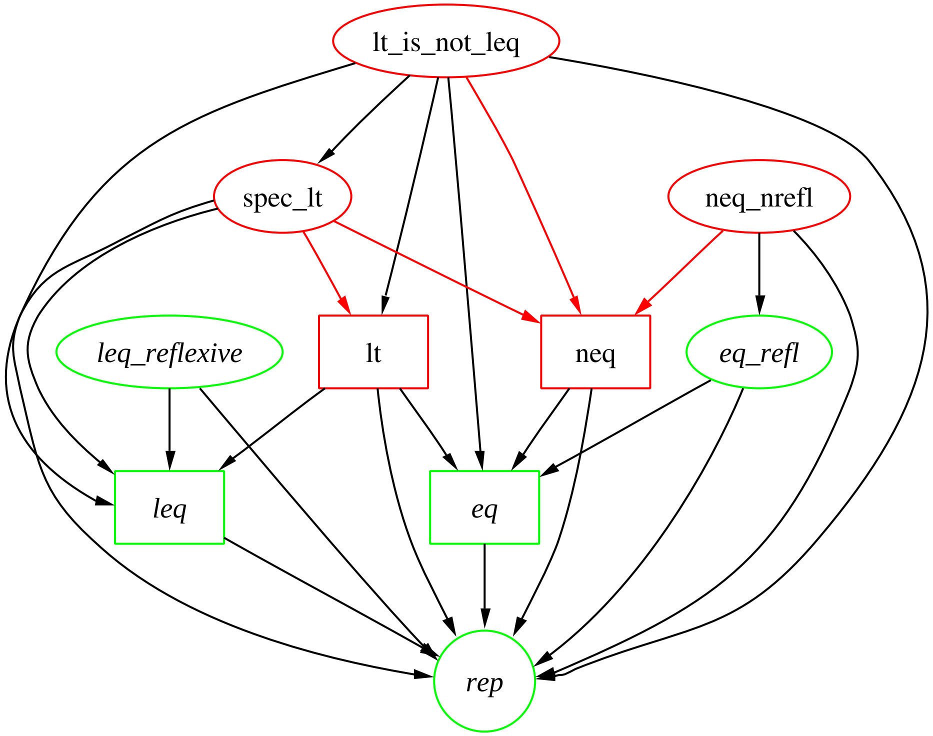
`spec_lt` can be seen as the "minimal hypothesis" on `lt` which is required to prove `lt_is_not_leq`

During inheritance, instead of (re)proving `lt_is_not_leq`, it suffices to (re)prove `spec_lt` (which should be shorter than the proof of `lt_is_not_leq`).

Second issue (2)

```
species partial_order  inherits setoid =
 sig leq  in self-> self->bool;
 let lt(x,y) = basics#and_b(!leq(x,y),basics#not_b(!eq(x,y)));
 theorem spec_lt :  all x y  in self,
  !lt(x,y) <-> (!leq(x,y)  and !neq(x,y))
 proof :def : lt neq; decl: leq;...;
 theorem lt_is_not_leq : all x y  in self,
  ((!lt(x,y) -> (!leq(x,y)  and !neq(x,y)))  and
  (!leq(x,y) -> (!lt(x,y)  or !eq(x,y))))
   proof: def:neq; decl:spec_lt leq;...;
...
```

Second issue



Dependence graph for partial_order

Some coding rules

* Shape of the hierarchy
* Minimize number of theorems with def-dependencies
* Minimize complexity of theorems with def-dependencies
* Do not have more than one def-dependency per theorem

Conclusion

- inheritance steps and redefinition
- invalidation and erasure of proofs
- need to decide when to do a proof
- informations given by the system
- towards a coding style ?

Perspectives

- Adding the specifications of each of the functions used in the library,
- proving them ... following the methodology presented here
- need for a specific proof tool (currently under design)
  - as automated as possible
  - based on Lamport's hierarchical style:
  - a proof consists of intermediate lemmas, each of them being proved by other lemmas, until reaching some trivial properties ...

  might allow to refine the def-dependency analysis to the intermediate lemmas level