Specifying in Coq inheritance used in Computer Algebra Libraries

Sylvain Boulmé

Laboratoire d'Informatique de Paris 6 (LIP6),
Université Pierre et Marie Curie (Paris 6),
8, rue du Capitaine Scott, 75015 Paris, France.
Sylvain.Boulme@lip6.fr

Abstract. This paper is part of FOC[3] a project for developing Computer Algebra libraries, certified in Coq [2]. FOC has developed a methodology for programming Computer Algebra libraries, using modules and objects in Ocaml. In order to specify modularity features used by FOC in Ocaml, we are coding in Coq a theory for extensible records with dependent fields. This theory intends to express especially the kind of inheritance with method redefinition and late binding, that FOC uses in its Ocaml programs.
The unit of FOC are coded as records. As we want to encode semantic information on units, the fields of our records may be proofs. Thus, our fields may depend on each others. We called them *Drecords*. Then, we introduce a new datatype, called *mixDrec*, to represent FOC classes. Actually, mixDrecs are useful for describing a hierarchy of Drecords in a incremental way. In mixDrecs, fields can be only declared or they can be redefined. MixDrecs can be extended by inheritance.

1 Introduction

This work lies within the scope of FOC, a project for developing a library for Computer Algebra, written in Ocaml and certified in Coq. Units of such a library are aimed to offer different views of mathematical algebraic structures, from the purely abstract one, very close to mathematics to some fully implemented ones, where the carrier and the code for the operators are completely fixed. For example, there exists a unit U_1 describing the (abstract) notion of ring, a second one U_2 for the integral domains and a third one U_3 giving the implementation of $\mathbb{Z}/2\mathbb{Z}$ on the booleans. So a unit provides declarations of identifiers, possibly some code for certain identifiers, and, as this library is to be certified, some assertions on identifiers and code. A unit is rarely built independently of the others and there are usually several kinds of dependence between units. For example, the unit of integral domains U_2 is built upon U_1 and U_3 gives an instantiation of the abstract representation of the carrier of U_2. Therefore, our library is a hierarchy — or a graph — of units.

We face up the question: how to proceed to the certification of this library, in order to ensure that the assertions put in the units are correct? As a unit U is built upon some other units U_i, the assertions (so their proofs) of U may depend of some assertions of the U_i. These dependencies may be handled case by case, by a method just driven by the assertions in U. Evidently, this is not a good solution as some sharing of the treatment of dependencies can be done. The construction of the library must be able to be done in an incremental way. In particular, one must be able to build the units the ones from the others, by expressing only the differences between them. We propose in this paper a description in Coq of such a library construction system.

How to represent a unit? A unit introduces a certain number of functions, which could be called by other units of the library. Thus, one must provide a non-ambiguous name for the function of a given library. But at the same time, it is convenient that two functions *"having the same meaning"* in different units share the same name. We choose to represent the units of our library via records (with the "." notation) which offer at the same time a non-ambiguous way to indicate a function of a library (it is not for example by the case, in approaches with overloading), and at the same time the possibility of sharing field names between records: and thus to indicate by a same name of the different functions.

Thus, roughly speaking, a unit will be coded as a record. As our units will contains types, programs, proofs, in a spirit à la Curry-Howard, our records will have dependent fields. We called them *Drecord*.

In order to describe Drecords in a incremental way, we introduce a new datatype, called *mixDrec*. As classes in object-oriented language, mixDrecs are generators of Drecords. In mixDrecs, fields can be only declared or they can be redefined. MixDrecs can be extended by inheritance. Thus, mixDrecs describe a family of Drecords.

Our requirements on records are the following: a same name could be shared between any records types; fields may depends on other fields; we want a notion of subtyping between records type, with its associate coercion. We want defines operation on records (coercions, inheritance, ...).

2 Records with dependent fields in Coq

In this section, we present quickly the records of the current version of Coq, explaining why they do not meet these requirements. Then, we introduce our proper definition of records, called *Drecords*.

2.1 Records in Coq V6.3.1

In Coq V6.3.1, records are coded via inductive definitions. A type record S is simply an inductive type with one constructor, called *Build_S* by default. Field access is coded as projections associated to this inductive type. For instance the following definition of the type record *pair* introduces the name `pair` and the two labels `fst` and `snd`. The constructor `Build_pair` is associated to this type and the labels are used to create the access functions to the fields of the record:

```
Structure pair[A,B:Set]: Set := { fst: A; snd: B }.
```

is actually a macro for:

```
Inductive pair[A,B:Set]:Set :=
  Build_pair: A->B->(pair A B).

Definition fst := [A,B:Set; x:(pair A B)]
    Cases x of (Build_pair fst _) => fst end.

Definition snd := [A,B:Set; x:(pair A B)]
    Cases x of (Build_pair _ snd) => snd end.
```

There is an important restriction on labels, which must not be shared by different record types. With the mechanism of coercion, this restriction can be partially raised. Moreover, encoding records via a macro mechanism and coercions operate at the level of record types. So this implementation does not meet our requirements on records, and we have to introduce our own notion. We do that step by step, first introducing some notations and the coding of field access.

2.2 A first sight of Drecords

Roughly speaking, a *Drecord* can be seen as a function from a given finite set of labels to a set of fields. Applying this function to a given label performs *field access*. As the types of these fields may differ, a notion of *Drecord signature* associating its type to each field is therefore needed.

Notations A is a type parameter (of sort `Set`) of the theory, which denotes the type of the labels. The equality of A is assumed to be decidable.

Let L and L_1 be lists of labels and a a given label. We suppose given the following definitions:

- $\in$: $(a \in L)$ is true if a belongs to L.
- $\supset$: $(L \supset L_1)$ is true if L contains L_1.
- $\not\pitchfork$: $(L \not\pitchfork L_1)$ is true if L and L_1 have no common labels
- $\widehat{.}$: $\widehat{L}$ is true if L does not contain two occurrences of the same label.
- $\setminus$: $(L \setminus L_1)$ is a function which returns the list of labels in L which do not belong to L_1.

The code Coq given here is in "implicit arguments mode": some arguments of functions are left implicit but, if needed, they can be given with the notation $n!$ where n is the position of the argument.

Field access We suppose given a type called `sign` of sort `Type` which is the type of *Drecord signatures* and a function `Drecord` of type `sign`→`Type`. They will be defined further.

A *Drecord* is, *by definition*, a term of type `(Drecord s)`, where `s` is a term of type `sign`. The function `sign_l` returns the list of labels of a Drecord and will also be defined further. In the following, expressions `(sign_l s)` are denoted by |`s`|.

Field access is defined by two functions, `field` and `fieldT`, which takes a Drecord `D` and a label `a` as arguments. `(field D a)` is the field associated with `a` in `D` and `fieldT` returns the type of `(field D a)`.

```
fieldT: (s:sign) (Drecord s)->A->Type
field:(s:sign ; i:(Drecord S) ; a:A ; H:(a∈|s|))(fieldT i a)
```

Note that the expression `(fieldT D a)` makes sense only if the label `a` figures in `D`. So, `fieldT` is not supposed to be used directly.

We turn now to the coding of `sign` and `Drecord`. The difficulty is to express the dependencies between the fields. So, we make an intermediary step by introducing telescopes.

2.3 Telescopes

In [8], the concept of *telescope* is used to express dependencies between fields of record-like structures. This notion was first introduced by [6] to express dependencies between contexts.

We introduce here telescopes with labels which implementation departs from those of [8] but we still name this new version *telescopes*.

Pair with dependent fields A pair with dependent fields is a Cartesian-product-like type. As usual, it is coded by a dependent sum upon two canonical injections:

```
Structure dpair[T2:Type; T1:T2->Type] : Type:= {x2: T2; x1: (T1 x2)}.
```

Let us remark that the different occurrences of `Type` in this definition denote different *Type_n* where n is an implicit level of universe. The precedent definition is actually:

```
Structure dpair[T2:Type_i; T1:T2->Type_j]: Type_k:={x2: T2; x1: (T1 x1)}.
```

where $i < k$ and $j < k$. This mechanism (invisible for the user) prevents the construction of paradoxes (cf. [9] and [4]).

In this denotation Σ_2 represents a constructor of binary existential type, and $\emptyset_T$ is a type with one element: $\emptyset$, called here empty telescope.

Telescopes are defined by an iteration of this type `dpair` on itself (see below). Moreover, each field will be labelled. These labels may be used for example to define notions of field access, subsignature,... They are firmly attached to the fields and may be considered as a part of the definition of the telescope: for example, they are not submitted to α-conversion. Now, the content of a field may depend on the preceding ones in the structure. As considered in [7] and [5], in a high order context, dependencies cannot be expressed by labels, because of variable captures. Dependencies have to be expressed by bound variables.

Thus, informally, a *labelled telescope* can be denoted as a term of type

$$\Sigma_n[x_n : T_n^{a_n} \ ; \ \dots \ ; \ x_1 : (T_1^{a_1} x_n \dots x_2)]\emptyset_T$$

where Σ_n denotes a dependent sum type build upon n canonical injections; where T_i are functions with values in types, labelled by a_i, themselves independent of x_i variables; and, $\emptyset_T$ is a type with a single element: the empty telescope.

Formally, telescopes are defined through two types: `sigtel`, the type of "telescope signatures", and `impltel` the type of "telescope implementations". The sort of `sigtel` and `impltel` id `Type`, as fields may lie in `Set` or `Prop` or even in `Type`. But, because of universe constraints, fields of a telescope leave in a universe below the one of this telescope.

Telescope signatures The type of telescope signatures, `sigtel`, is defined by recurrence on lists. It uses two base types: `EsigT` the type having only one element which represents the type of the empty signature, and `(FunIn a T)` the type, labelled by `a`, of functions with values in `T`.

```
Inductive EsigT: Type := Esig_: EsigT.

Structure FunIn[a:A; T:Type]: Type:=
   {dom: Type; fun:> (dom->T)}.
```

```
Fixpoint sigtel[l:(list A)]:Type :=
  Cases l of
    nil => EsigT
  | (cons a m) => (FunIn a (sigtel m))
  end.
```

The ">" on `fun` in `FunIn` declares the `fun` projection as a Coq coercion. Without it, if `s` a term of type `(FunIn a T)` and `x` a term of type `(dom s)`, the term `(s x)` does not typecheck, because `(FunIn a T)` is not a function type. With this coercion, this term typechecks, and mean `((fun s) x)` (see below in `impltel`).

Telescope implementations Telescope implementations are defined in the same way, by an iteration on dependent pairs:

```
Inductive EimplT: Type := Eimpl_: EimplT.

Structure dpairT[T:Type,f:T->Type; a:A]: Type:=
  {dprojT1: T; dprojT2: (f dprojT1) }.
```

```
Fixpoint impltel[l:(list A)]:(sigtel l)->Type:=
 <[l:(list A)](sigtel l)->Type>Cases l of
      nil => [_]EimplT
    | (cons a m) =>
         [s](dpairT [x:(dom s)](impltel (s x)) a)
 end.
```

These two recursive types `sigtel` and `impltel` could alternatively be defined by using inductive types of Coq. For instance, `sigtel` could be write as:

```
Inductive sigtel: (list A)->Type :=
   Esig: (sigtel (nil A))
 | Csig: (a:A; l:(list A); dom:Type)(dom->(sigtel l))->(sigtel (cons a l)).
```

But, if recursive types (defined by fixpoint) are less general than inductive ones, they are more convenient to handle in some situation. Inversion lemmas on inductive types are not well automatically generated when using dependent types, whereas they simply correspond to reduction on recursive types.

Also, defining `impltel` by an inductive type requires to type it in a higher universe than `sigtel`, which is against intuition. With the recursive definition, `imptel` lies in a lower universe than `sigtel`. Thus, in all this implementation, we use only "not-recursive" inductive types (except for defining `cont0` and `subsig`, see below). For that, we pass the recursive calls as an argument of the inductive type (like in continuations): for example, `FunIn` corresponds to the *Csig* constructor, with `T` as parameter, to capture the recursive call. Then we use a fixpoint, to express the recursive calls. The counterpart of this method is that such recursive definitions of type are harder to establish and to understand.

Field access Field access is done by two functions, `fieldsig` which returns the type of the field, and `fiedimpl` which returns this field. These two functions are not totally defined. The type `Dummy` with a single constructor `foo` is used to express this partiality.

```
Inductive Dummy: Type := foo: Dummy.

Hypothesis eqA_dec:(x, y:A){x=y}+{~x=y}.

Fixpoint fieldsig[a:A; l:(list A)]:(s:(sigtel l))(impltel s)->Type :=
  <[l:(list A)](s:(sigtel l))(impltel s)->Type>Cases l of
      nil => [_;_]Dummy
    | (cons b m) => [s;i]
         if (eqA_dec a b) then
           [_](dom s)
         else
           [_](fieldsig a (dprojT2 i))
  end.

Fixpoint fieldimpl[a:A; l:(list A)]:(s:(sigtel l); i:(impltel s))(fieldsig a i):=
  <[l:(list A)](s:(sigtel l); i:(impltel s))(fieldsig a i)>Cases l of
      nil => [_;_]foo
```

```
    | (cons b m) =>
        [s;i]<[H:a=b+~a=b]if H then [_](dom s) else [_](fieldsig a (dprojT2 i))>
         if (eqA_dec a b) then
            [_](dprojT1 i)
         else
            [_](fieldimpl a (dprojT2 i))
  end.
```

From telescopes to Drecords A Drecord is a telescope which labels (a_i in the informal definition) are pairwised distinct. This is only a "semantic" property: the access to a field has to be non ambiguous. All the operations on Drecords may be built independently of this condition: they are firstly defined on telescopes. In a first attempt, we coded Drecords by considering at the same time operational and semantic aspects. It became quickly unmanageable because semantic considerations polutted the code.

This semantic property is encoded by putting guards in order to restrict the use of the telescopes. These guards are the predicate denoted by $\in$, $\supset$, $\not\cap$ and $\frown$. They are decidable (under the assumption of the decidability of labels equality), and they can be discharged by the Coq typechecker.

Coding the guards on the lists To express the fact that guards express only that the "Drecord semantic" is fulfilled, they are put into `Prop`. As they are decidable, their values may be: the type `True` or the type `False`. They are implemented as instantiations of the two predicates `AllD` and `ExD` below.

```
Variable P:A->(list A)->Prop.
Hypothesis P_dec: (x:A; l:(list A)){(P x l)}+{~(P x l)}.

Fixpoint AllD[l:(list A)]: Prop :=
   Cases l of
      nil => True
    | (cons a m) => if (P_dec a m) then [_](AllD m) else [_]False end.

Fixpoint ExD[l:(list A)]: Prop :=
   Cases l of
      nil => False
    | (cons a m) => if (P_dec a m) then [_]True else [_](ExD m) end.
```

Definition 1. *Signatures of Drecord are a triple: a list* `sign_l`*, a proof* `sign_l_df` *that this list is double-free (guard instantiating* `AllD` *above), and a sigtel* `sign_p` *built on* `sign_l`*.*

```
Structure sign: Type := {
    sign_l: (list A);
    sign_l_df: ⌢sign_l;
    sign_p:> (sigtel sign_l) }.
```

Let us remark here, that `sign_p` is a Coq coercion, from `sign` to `sigtel`.

Definition 2. *Drecords are simply telescopes, built upon a Drecord signature:*

```
Definition Drecord[s:sign]:=(impltel s).
```

The transformation of telescope into a Drecord is thus only a type coercion.

Definition 3. *Functions for accessing fields are:*

```
Definition fieldT:(s:sign)(Drecord s)->A->Type
  :=[s;i;a](fieldsig eqA_dec a i).

Definition field:(s:sign; i:(Drecord s); a:A; H:(a∈|s|))(fieldT i a)
  :=[s;i;a;H](fieldimpl eqA_dec a i).
```

3 Operations and relations between Drecord signatures

This section presents operators and properties of Drecords. They are first informally introduced, then the operators and properties on the underlying telescopes are described.

3.1 Subsignature relation and coercion between Drecords

Properties Between Drecord signatures, there is a natural relation, `subsign`, of subsignature: s_1 is a subsignature of s_2 (we will informally write $s_1 :> s_2$), if s_1 can be transformed into s_2 by forgetting or permuting some fields. Internally, this concept of subsignature is implemented as an inductive type `subsig` on telescope signatures, which allow to reason by induction on "proofs" of subsignatures (cf. below).

Theorem 1. *The relation* `subsign` *of type* `sign->sign->Type` *is a preorder, whose associated relation of equivalence can be easily defined: it corresponds to* `subsign` *on signatures of equal length.*

Proof This has been proved in Coq:

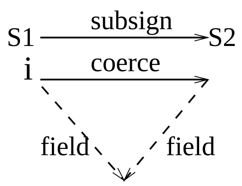
```
(* Preorder properties*)
Lemma subsign_refl:(s:sign)(subsign s s).

Lemma subsign_trans:(s1,s2,s3:sign)(subsign s1 s2)->(subsign s2 s3)->(subsign s1 s3).

(* Properties of the associated equivalence  *)
Lemma subsign_antisym:(s1,s2:sign)(subsign s1 s2)->(subsign s2 s1)
      ->(length |s1|)=(length |s2|).

Lemma subsign_sym:(s1,s2:sign)(subsign s1 s2)
      ->(length |s1|)=(length |s2|)->(subsign s2 s1).
```

□

Associated with this concept of subsignature, there is a function `coerce` for converting Drecords, which preserves the extensional behavior of Drecords (equality of the fields accessed via the same labels). These coercions will correspond to coercions on objects in the FOC project.

Theorem 2. *Let* $s1$ *and* $s2$ *two signatures such that* $s1 :> s2$*. Then the function* `coerce` *of type* `(Drecord s1)->(Drecord s2)` *has the following property (called* `subsign_ext`*):*

Let i *be a* `(Drecord s1)`*, then for any label* a *in* `|s2|`*, the access to* a *via* `field` *on* i *is equal to the one on* `(coerce i)`.



Proof Let us assume the definition of `eqT_dep`[1], the dependent equality over types. The theorem `subsign_ext` has been proved in Coq:

```
Lemma subsign_cont:(s1,s2:sign)(subsign s1 s2)->|s1|⊃|s2|.

Lemma subsign_ext_T: (s1,s2:sign; H:(subsign s1 s2); i:(Drecord s1); a:A)(a∈|s|)
       ->(fieldT i a)==(fieldT (coerce H i) a).

Theorem subsign_ext: (s1,s2:sign; H1:(subsign s1 s2); i:(Drecord s1); a:A; H2:(a∈|s2|))
      (eqT_dep 2![T:Type]T (field i (cont_incl (subsign_cont H1) H2)[2])
                           (field (coerce H1 i) H2)).
```

□

This last property (shown in Coq) guarantees the correction of the implementation of subsignature with respect to its semantics. In particular, for every "proof" of subsignature between two signatures, the associated coercions at Drecords level are equivalent (with respect to field access).

[1] With the following definition with implicit arguments on:

```
Inductive eqT_dep[U:Type; P:U->Type; p:U; x:(P p)]: (q:U)(P q)->Prop:=
          eqT_dep_intro: (eqT_dep x x).
```

[2] proof of `a∈|s1|`

Implementation The subsign relation from `s1` to `s2` is defined as a succession of elementary transformations on signatures (forgetting fields, permuting fields, ...). More formally, it is divided in two parts. First, `subsign_cont0` gives the skeleton these elementary transformations to pass from |`s1`| to |`s2`|. And then these succession of transformations is expressed at the level of sigtels by `subsign_p`:

```
Structure subsign[s1,s2:sign]: Type :=
  { subsign_cont0: (cont0 |s1| |s2|);
    subsign_p:> (subsig subsign_cont0 s1 s2)}.
```

`cont0` is defined by:

```
Inductive cont0:(list A)->(list A)->Set :=
   cont0_nil: (l:(list A))(cont0 l (nil A))
 | cont0_cons: (a1,a2:A)(l1,l2:(list A))a1=a2->(cont0 l1 l2)->
                     (cont0 (cons a1 l1) (cons a2 l2))
 | cont0_lift:(a:A; l,m:(list A))(cont0 l m)->(cont0 (cons a l) m)
 | cont0_swap:(a1,a2:A; l1,l2:(list A))
     (cont0 l1 l2)->(cont0 (cons a1 (cons a2 l1)) (cons a2 (cons a1 l2)))
 | cont0_trans: (l1,l2,l3:(list A))(cont0 l1 l2)->(cont0 l2 l3)->(cont0 l1 l3).
```

Then `subsig` is defined as follows:

```
Inductive subsig: (l1, l2:(list A))(cont0 l1 l2)->(sigtel l1)->(sigtel l2)->Type :=
   subsig_E: (l:(list A); s:(sigtel l))(subsig (cont0_nil l) s Esig)
 | subsig_C: (a:A; l1, l2:(list A); T:Type; f:T ->(sigtel l1); g:T ->(sigtel l2);
     H:(cont0 l1 l2))
      ((x:T)(subsig H (f x) (g x)))->(subsig (cont0_cons (refl_equal A a) H)
                                                (Csig a f) (Csig a g))
 | subsig_lift: (a:A; l1, l2:(list A); T:Type; f:T ->(sigtel l1); s:(sigtel l2);
     H:(cont0 l1 l2))
       ((x:T)(subsig H (f x) s))->(subsig (cont0_lift a H) (Csig a f) s)
 | subsig_swap:
     (a1, a2:A; l1, l2:(list A); T1, T2:Type;
       f:T1 -> T2 ->(sigtel l1); g:T2 -> T1 ->(sigtel l2))
       (H:(cont0 l1 l2))
       ((x:T1; y:T2)(subsig H (f x y) (g y x))) ->
       (subsig (cont0_swap a1 a2 H)
           (Csig a1 [x:T1](Csig a2 [y:T2](f x y)))
           (Csig a2 [y:T2](Csig a1 [x:T1](g y x))))
 | subsig_trans:
     (l1, l2, l3:(list A); s1:(sigtel l1); s2:(sigtel l2); s3:(sigtel l3);
          H1:(cont0 l1 l2); H2:(cont0 l2 l3))
          (subsig H1 s1 s2) -> (subsig H2 s2 s3) -> (subsig (cont0_trans H1 H2) s1 s3).
```

where

```
Definition Esig: (sigtel (nil A)) := Esig_.
Definition Csig: (a:A; l:(list A); T:Type)(T->(sigtel l))->(sigtel (cons a l))
  :=[a,l,T,f](Build_FunIn a f).
```

If we would have not defined `cont0`, but directly `subsig`, it would not be possible to have a right notion of "succession of transformations". This notion is however necessary to define the merge of signature (cf. 3.3), because the use of permutation rule will have an influence on the order of the label list in the result signature.

The function of coercions between impltels, `coerce_impltel`, is then a trivial induction of a proof of subsig. Here is its type :

```
Definition coerce_impltel: (l1, l2:(list A); s1:(sigtel l1); s2:(sigtel l2);
 H:(cont0 (eq A) l1 l2))(subsig H s1 s2) -> (impltel s1) -> (impltel s2).
```

Thus,

```
Definition coerce: (s1,s2:(sign))(subsign s1 s2)->(Drecord s1)->(Drecord s2)
  :=[s1,s2;H;i](coerce_impltel H i).
```

3.2 Extending a Drecord signature some fields

Description A simple extension of signature, `flatsign`, consists to add some **new** fields at the end of a given signature `s`. But these new fields may depend on the fields of `s`. So, this new part of signature must be expressed in a context containing the fields of `s` i.e. a Drecord of signature `s`.[3] The part of signature may thus be expressed by a function of the type `(Drecord s)->(sigtel l)`, where `l` is a double-free list of the new labels.

Theorem 3. *The simple extension of a signature* `s`*, is a subsignature of* `s`.

Proof This has been proved in Coq:

```
Variable s:sign.
Variable l:(list A)).
Hypothesis H1:l̂.
Hypothesis H2:(l∩̸|s|).
```

```
Definition flatsign: ((Drecord s)->(sigtel l))->sign.

Theorem flatsign_subsign:
  (f:(Drecord s)->(sigtel l))(subsign (flatsign f) s).
```

□

By construction |`flatsign`| is the concatenation of |`s`| and from `l`. As a particular case of extension of signatures, one derive the concatenation of signatures:

```
Variables s1,s2:sign.
Hypothesis H:(|s2|∩̸|s1|).
```

```
Definition concatsign:=
 (flatsign (sign_l_df s2)⁴ H [_:(Drecord s1)](sign_p s2)).

Lemma subsign_concatsign_1: (subsign concatsign s1).
Lemma subsign_concatsign_2: (subsign concatsign s2).
```

Implementation Actually, `flatsign` corresponds to the `flatsig` operation on telescopes. For readability reasons, we will now denote `(impltel s)` by $\overline{s}$ and `(coerce_impltel H i)` by $i_{|_H}$. `flatsig` is a trivial induction on `l1`, which type is:

```
flatsig: (l1,l2:(list A))(s1:(sigtel l1))s̅1̅->(sigtel l2))->(sigtel (l1^l2)).
```

Here are some important properties of `flatsig` used to build merging of signatures. The terms "`cont0`" are here hidden with "`...`"

- The extension of a signature `s1` by `s2` that does not depend on `s1` is a subsignature of `s2`.
```
Lemma flatsig_constant: (l1,l2:(list A); s1:(sigtel l1); s2:(sigtel l2))
  (subsig ... (flatsig [_:s̅1̅]s2) s2).
```
- Subsignature is compatible with extension
```
Lemma subsig_flatsig_coerce: (l1,l2:(list A); s1:(sigtel l1); s2:(sigtel l2); ...;
 H:(subsig ... s2 s1); l3:(list A); f:s̅1̅->(sigtel l3))
   (subsig ... (flatsig [i](f i|H)) (flatsig [i](f i))).

Lemma subsig_flatsig: (l1,l2,l3:(list A); s:(sigtel l1); f:s̅->(sigtel l2);
 g:s̅->(sigtel l3); ...)
  ((i:s̅)(subsig ... (f i) (g i)))->
    (subsig ... (flatsig f) (flatsig g)).
```
- Subsignature is compatible with the permutation of a field with a packet of fields:
```
Lemma subtel_Csig_flatsig :
 (l1:(list A); s:(sigtel l1); a:A; l2,l3:(list A); T:Type;
   f:T->s̅->(sigtel l2); g:T->(sigtel l3); ...)
   ((x:T)(subsig ... (flatsig [i](f x i)) (g x)))
     ->(subsig ... (flatsig [i](Csig a [x](f x i))) (Csig a g)).
```

3.3 Merging signatures

A more delicate point is to define an operation for merging signatures. The concatenation of signatures previously defined can only be applied to signatures having no common label. On the contrary, the merge operation allows signatures to share some labels. But it imposes that these shared labels correspond to compatible fields.

[3] Do not forget that telescopes were introduced to formalize the concept of context

[4] proof of $\widehat{|\texttt{s2}|}$

Description This constraint on sharing requires the existence of a subsignature (possibly empty) common to these two signatures and containing all labels present in both signatures. This constraint on signatures being not decidable, the user is asked to discharge it completely. Later on, it is planed to write a tactic (heuristic) which succeeds in discharging sometimes this constraint.

A way to implement the merging operation, is first to write the inverse of `flatsign`, i.e., a function `apsign` such that given two signatures s_1 and s_2 with $s_1 :> s_2$, `apsign` factorizes s_2 in s_1, by expressing the fields of s_1 which do not belong to s_2 in function of those of s_2.

```
apsign: (s1,s2:sign)(subsign s1 s2)->(Drecord s2)->sign.

apsign_subsign_l:(s1,s2:sign; H:(subsign s1 s2))
    (subsign (flatsign ... [i:(Drecord s2)](apsign H i)) s1).

apsign_subsign_r:(s1,s2:sign; H:(subsign s1 s2))
    (subsign s1 (flatsign ... [i:(Drecord s2)](apsign H i))).
```

With `apsign` defined, `mergesign` is quite simple:

```
Variables s,s1,s2:sign.
Hypothesis H1: (subsign s1 s).
Hypothesis H2: (subsign s2 s).
Hypothesis H3: (|s2|\|s|)⋂̸(|s1|\|s|).

Definition mergesign :=
    (flatsign ... [i:(Drecord s)](concatsign 1!(apsign H1 i) 2!(apsign H2 i) H3)).
```

Theorem 4. *The merge of two signatures is both a subsignature of these two signatures.*

Proof In Coq, we proved the following lemmas:

```
Lemma subsign_mergesign_1: (subsign mergesign s1).
Lemma subsign_mergesign_2: (subsign mergesign s2).
```

□

By construction |`mergesign`| is the concatenation of three lists: |`s`|, (|`s1`|\|`s`|) and (|`s2`|\|`s`|).

Implementation This function `apsign` is defined at the telescope level, by structural induction on the proof that $s_1 :> s_2$. The main difficulty of this definition (and even of all this development) comes at `subsig_trans` case.

Informally, assuming that `apsig` with the type

$$(s_1 :> s_2) \to (\overline{s_2} \to (\texttt{sigtel}\ |s_1| \setminus |s_2|))$$

for s_1 and s_2 sigtels, we are defining it by induction on the proof that $s_1 :> s_2$. The `subsig_trans` case corresponds to :

$$\begin{array}{l} H_1 : s_1 :> s_2 \\ f : \ \ \overline{s_2} \to (\texttt{sigtel}\ |s_1| \setminus |s_2|) \\ H_2 : s_2 :> s_3 \\ g : \ \ \overline{s_3} \to (\texttt{sigtel}\ |s_2| \setminus |s_3|) \\ \hline \qquad \overline{s_3} \to (\texttt{sigtel}\ |s_1| \setminus |s_3|) \end{array}$$

The idea is to give the following term for this goal :

$$[i : \overline{s_3}](\texttt{apsig}\ (\texttt{flatsig}\ [j : \overline{(\texttt{flatsig}\ g)}](f\ j_{|_{s_2}}))\ i)$$

In order to write "$[j : \overline{(\texttt{flatsig}\ g)}](f\ j_{|_{s_2}})$" (expression called h below), `apsign_subsign_l` has to be proved in the same time than `apsig` is build.

Also, unfortunately, `apsig` is not applicable to `flatsig` h because `apsig` is being defined by structural induction. Fortunately, one can that prove `flatsig` h is a signature signature whose s_3 is a prefix (in the same sense than prefix on words). Thus, we introduce this prefix relation between telescopes signatures (showing for instance that if s_2 is a prefix of s_1 then $s_1 :> s_2$). Then, we define the function `apsig_pre` corresponding to `apsig` for the prefix relation.

Finally, `apsig` will be defined as the first projection of `apsig_dpair`, which type is :

```
Definition apsig_dpair:(l1,l2:(list A); s1:(sigtel l1); s2:(sigtel l2);
 H:(cont0 l1 l2); H1:(subsig H s1 s2))
   (dpair [f:(impltel s2)->(sigtel (extract0 H))]
          (subsig (cont0_app_extract0 H) (flatsig [i](f i)) s1)).
```

where `extract0` is here a function which takes a proof `H` of `(cont0 l1 l2)` and which returns a list "`l1` without the element of `l2`". The order of the elements in this list depends on `H` (or more especially, of the permutations in `H`). Actually what we abusively denote in all this paper by $l1 \setminus l2$ is `(extract0 H)`.

`apsig_dpair` is defined by the induction described above. And the term corresponding to the `subsig_trans` case is finally :

$$[i : \overline{s_3}](\texttt{apsig\_pre}\ (flatsig\ h)\ i)$$

3.4 Renaming

Renaming has not yet been implemented. We just present which renaming operations we intend to have. A partial function f from labels to labels is a *renaming on labels*, a partial function f if it is injective on its definition domain, D_f. Such functions form a group for the composition law ($D_{f \circ g} = g^{-1}(I_g \cap D_f)$ and $D_{f^{-1}} = I_f$).

Given a renaming on labels f, the associated *renaming on lists of labels* is the map extension of f from list of labels to list of labels (thus, its domain is the set of lists whose elements are in D_f).

By extension, a *renaming on Drecord signatures* (resp. *Drecords*) is the corresponding mapping on Drecord signatures (resp. Drecords).

We can now define a notion of *subsignature modulo renaming. A signature S_1 is a subsignature of another signature S_2 modulo renaming*, if there exists two renamings on signatures f and g such that $(f\ S_1)$ is a subsignature of $(g\ S_2)$.

If such f and g exist, then $f(|S_1|) \supset g(|S_2|)$. Then $(g|S_2|) \subset D_{f^{-1}}$, and $|S_2| \subset D_{f^{-1} \circ g}$. Thus, this definition is equivalent to the following one: *A signature S_1 is a subsignature of another signature S_2 modulo renaming, if there exist a renaming on signatures f such that S_1 is a subsignature of (fS_2).*

Let S_1 a subsignature of $(f\ S_2)$, we call c the associated coercion from Drecords of signature S_1 to Drecords of signature $(f\ S_2)$. We can define a coercion from Drecords of signature S_1 to Drecords of signature S_2 as the function $f^{-1} \circ c$.

4 Example of use of this theory

In our example, we start by defining the type of the labels, as a (very simple) finite enumerated type, called `Label`, which contains all the labels needed.

```
Inductive Label: Set := T: Label | EQ : Label | EQ_refl: Label | EQ_sym: Label
  | EQ_trans: Label | OP: Label | OP_EQ_l: Label.

Lemma eqLabel_dec : (x,y:Label){x=y}+{~x=y}.
  Intros x y; Case x; Case y; Auto; Right; Discriminate.
Defined.
```

We build `setoid_sig`, the setoid signature, and then the Drecord type `setoid`. We use extensible grammars of Coq to mask the mechanisms of guard on the lists: the system checks that the list of the labels of `setoid_sig` is double-free (the user has no proof to do). failure occurs:

```
Definition setoid_sig:=(<eqLabel_dec>Sign
 T:Type;
 EQ: T->T->Prop;
 EQ_refl: (x:T)(EQ x x);
 EQ_sym: (x,y:T)(EQ x y)->(EQ y x);
 EQ_trans: (x,y,z:T)(EQ x y)->(EQ y z)->(EQ x z)).

Definition setoid:=(Drecord setoid_sig).
```

Thanks to a function Build of our Drecords theory, we make a function for building a setoid by giving all its fields in the order where they appear in the signature. For example, types themselves form a setoid defined by:

```
Definition Build_setoid:=(Build setoid_sig).

Definition Type_setoid: setoid :=
  (Build_setoid (refl_eqT Type) (sym_eqT Type) (trans_eqT Type)).
```

The fields of Drecords are accessed via the binary operator @. It masks the check that the required field belongs actually to the Drecord:

```
Definition carrier: setoid->Type:=[s](s @ T).
```

The type semigroup is build by using the extension of signature on setoids, adding a binary law, associative, and compatible with the equivalence relation:

```
Definition semigroup_on:=[s:setoid](Sigtel
     OP:((s @ T)->(s @ T)->(s @ T)) ;
     OP_EQ_l: (x,y,z:(s @ T))((s @ EQ) x y)->((s @ EQ) (OP x z) (OP y z)) ;
     OP_EQ_r: (x,y,z:(s @ T)) ((s @ EQ) x y)->((s @ EQ) (OP z x) (OP z y)) ;
     OP_ass : (x,y,z:(s @ T))((s @ EQ) (OP (OP x y) z) (OP x (OP y z)))).

Definition semigroup_sig:=(Flat [s](semigroup_on s)).
Definition semigroup:=(Drecord semigroup_sig).
```

We benefit then from conversions of the theory, to transform them into Coq coercions.

```
Definition sgp_setoid: semigroup -> setoid := (coerce (FlatSub [s](semigroup_on s))).
Coercion sgp_setoid: semigroup >-> setoid.
```

Then, the type order of ordered setoids is built on the same model than semigroup. Merging semigroup_sig and order_sig (with setoid_sig as shared subsignature) enables us to create a Drecord type ordsg which can be extended for describing the ordered semigroups (it is necessary to add assumptions of compatibility between the order relation and the operation of the semigroup). Unfortunately, these computations on types are too large for the current implementation of Coq. That is due at least in a partial way to the fact that the Drecord are not primitive. For instance, the complexity for accessing fields of a Drecord seems to be exponential in function of its length.

5 MixDrecs

For managing mecanisms for declaring deferred fields or redefining fields, we introduce a new data type, called mixDrec, corresponding to classes in object oriented languages. The signature of Drecord gives an interface for specifying how it can be used. MixDrecs are a way to specify how Drecords can be build.

5.1 Motivations

We introduce the notion of mixDrec an example in a virtual syntax, inspired from Coq. The proofs are λ-terms. We use the fact that $\neg P$, where P is a proposition, is defined as $P \to \texttt{False}$.

The mixDrec of setoids may be defined as follows (for concision of the example, eq is only supposed reflexive), where the fields prefixed by decl are only declared, and the fields prefixed by def are defined.

$$
setoid := \left\{\begin{array}{l}
\texttt{decl}\ T:\ \texttt{Type}\\
\texttt{decl}\ eq:\ T \to T \to \texttt{Prop}\\
\texttt{decl}\ eq\_refl:\ \forall(x:T)(eq\ x\ x)\\
\texttt{def}\ \ neq:\ T \to T \to \texttt{Prop}\\
\qquad := \ \lambda[x,y:T]\neg(eq\ x\ y)\\
\texttt{def}\ \ neq\_spec:\ \forall(x,y:T)(neq\ x\ y) \to \neg(eq\ x\ y)\\
\qquad := \ \lambda[x,y:T;H:\neg(eq\ x\ y)]H\\
\texttt{def}\ \ neq\_nrefl:\ \forall(x:T)\neg(neq\ x\ x)\\
\qquad := \ \lambda[x:T;H:(neq\ x\ x)](neq\_spec\ x\ x\ H\ (eq\_refl\ x))
\end{array}\right\}
$$

Dependences between fields appear in fields signatures or definitions, and there are of two kinds like in Coq. The dependence of a field m_2 on a field m_1 is said *opaque* if expressing the signature or the definition of m_2 requires to know the signature of m_1, but not its definition. The dependence is said *transparent* if expressing the definition of m_2 requires to know the definition of m_1.

For instance, there are opaque dependencies of neq_nrefl with every other fields in *setoid* (by transitive closure). But the dependence of *neq_spec* on *neq* is transparent, and dependencies of *neq_spec* on *eq* and T are opaque.

These notions of dependencies give a sufficient condition to ensure the coherence of the redefinition of fields. Indeed, the only requirements for the (re)definition of a field m to be valid, is to respect the type of m, if this one has been precedently be declared (or defined). When building a mixtel B by redefining some field m of a mixtel A, the fields of A that depends opaquely on m have not to be redefined, because their definition will still be coherent with their type. But the definition of the fields of A depending transparently on m are lost, because they may become incoherent.

In *setoid*, we may redefine *neq* without having to redefine `neq_nrefl` (so, we freely inherit it). But, if we redefine *neq*, the definition of `neq_spec` is lost. It is a kind of proof obligation, the user has to redefine them.

The user will have to carefully manage these kinds of dependencies: opaque dependencies will express generic properties, but transparent ones will express fundamental specifications.

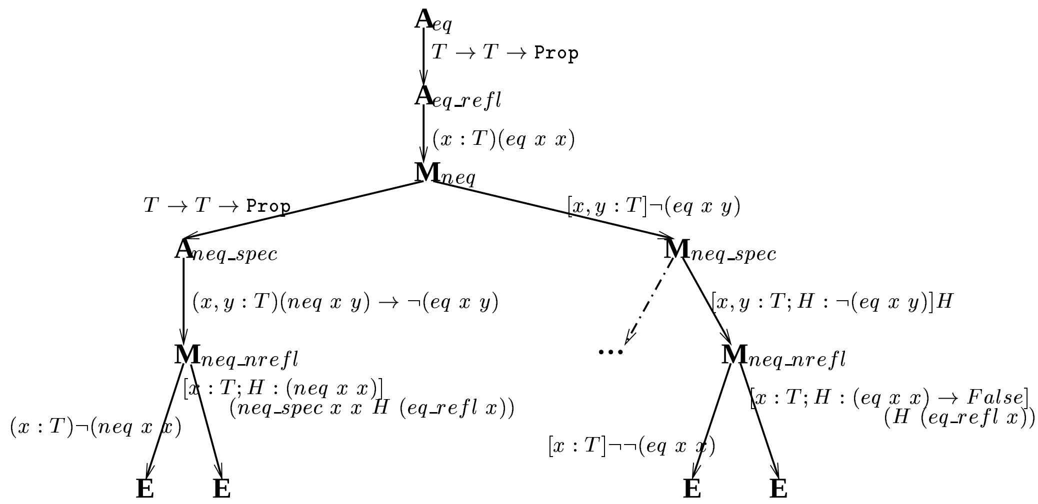
5.2 Informal description

Roughly speaking, mixDrecs contain both defined fields, and declared (but not yet defined) fields. There is thus a correspondence between Drecord implementations and mixDrecs whose all fields are defined. And dually, there is a correspondence between Drecord signatures and mixDrecs whose all fields are declared.

As for Drecords, we define a type, called *mixtel*, on sigtel. Then a mixDrec is a well-formed mixtel, whose list of labels is double-free.

Unlike telescopes signatures which have a linear structure, mixtels have a tree structure with two kinds of internal nodes: nodes known as "abstract" which represent roughly speaking declared fields, and those known as "manifest" which represent defined fields. "Abstract" nodes correspond to the "nodes" of telescope signatures: they contain the same information, in particular, they have a unique son. "Manifest" nodes contain several informations: the name of the field, its type, its implementation, and two sons: a mixtel which depends in an opaque way on this field (as in telescope signatures) and a mixtel which depends in a transparent way on this field (as in telescope implementations).

Mixtel corresponding to the *setoid* mixDrec above



A :abstract nodes ; **M** : manifest nodes ; **E** : leafs (empty mixtel).

In a manifest node, the two sons represent the same mixtel: the left son (opaque) represents this mixtel if one forgets (or changes) the implementation associated with the node, the right son (transparent) represents this mixtel when this implementation is preserved. A *well-formed* mixtel is such that the right son of every manifest nodes is a more-defined view of the left son mixtel.

At last, let us precise a little the concepts of declared and defined field in a well-formed mixtel. A field will be defined in a well-formed mixtel when in the rightest branch, the node carrying its label is manifest. On the contrary, a field will be declared, when in this branch, this node is abstract. In this case, all the mixtel nodes which carry this label will be abstract (under the assumption of mixtel well-formness). When all the fields of a mixtel are defined, a telescope implementation may be built directly: it is the rightest branch.

5.3 Implementation

The type `mixtel` is defined by two parts: first, the structure of trees is expressed only on lists of labels (via a predicate on lists `pre` which gives which labels are "Abstract" or "Manifest"), and then the dependencies between fields are expressed in `mixtel`. As for telescopes, `pre` and `mixtel` are recursive types instead of inductive types. Thus, we introduce respectively `flag` and `mix` which contains the "inductive part" of these definitions.

```
Inductive UnitS:Set:= unitS:UnitS.

Inductive flag[U:Set]:Set :=
   Abs: U->(flag U)
 | Man: U->U->(flag U).

Fixpoint pre[l:(list A)]: Set :=
  Cases l of
       nil => UnitS
  | (cons a m) => (flag (pre m))
  end.

Definition abs_of:=[U:Set;x:(flag U)]
  Cases x of (Abs p) => p
           | (Man p _) => p
  end.
```

```
Variable T:Type.
Variable U:Set.
Variable F:T->U->Type.

Structure dpair[f:T->Type]:Type :=
  {dproj1:T;dproj2:(f dproj1)}.

Definition option:(flag U)->Type:=
 [p]Cases p  of
   (Abs _)=>UnitT
 | (Man _ p2)=>(dpair [x:T](F x p2))
 end.

Structure mix[p:(flag U);a:A]:Type:=
 {mix_spec:(x:T)(F x (abs_of p));
  mix_val:(option p) }.
```

```
Fixpoint mixtel[l:(list A)]:(sigtel l)->(pre l)->Type :=
  <[l:(list A)](sigtel l)->(pre l)->Type>Cases l of
         nil => [_;_]EsigT
    | (cons a m) => [s;p](mix [x:(dom s)](mixtel (s x)) p a)
  end.
```

5.4 Properties and operations

The predicate `is_all_def` tests if all labels in a `pre` are defined. The function `new` which generates an impltel from a mixtel, whose all labels are defined.

```
Definition new:(l:(list A); s:(sigtel l); p:(pre l))(is_all_def p)
  ->(mixtel s p)->(impltel s).
```

Definition 4. *Then, given a signature* `s`*, and two mixtels* `m1` *and* `m2`*,* `(match_mix m1 m2)` *is a proof that the fields of m_1 are a more-defined view (with the same definition) of their equivalent in m_2 (*`match_pre` *is the correspondent notion on* `pre`*).*

```
l:(list A)
s:(sigtel l)
p1,p2:(pre l)
```

```
Definition match_mix:
  (match_pre p1 p2)
    ->(mixtel s p1)->(mixtel s p2)->Type.
```

`match_mix` is a transitive relation, but not reflexive (nor anti-reflexive).

Theorem 5. *We have then the following property: if all the labels of* `m2` *are defined, then all the labels of* `m1` *are also (*`is_all_def_match` *property), and the telescope implementation generated, via* `new`*, by* `m2` *is equal to the one generated by* `m1`*.*

```
m1:(mixtel s p1)
m2:(mixtel s p2)
```

```
Theorem match_mix_new:(H1:(is_all_def p2);
  H2:(match_pre p1 p2))(match_mix H2 m1 m2)->
    (new H1 m2)===(new (is_all_def_match H1 H2) m1).
```

Definition 5. *A well-formed mixtel is a mixtel that checks the reflexivity of* `match_mix`.

```
Definition wellmixed:=
  [p:(pre l);H:(wellpre p);m:(mixtel s p)](match_mix H m m).
```

We have then two operations on mixtels: `merge_mix` for merging two mixtels of same signatures, and `lift_mix` for lifting a mixtel in a subsignature.

```
Definition merge_mix:(l:(list A); s:(sigtel l); p1,p2:(pre l))
    (mixtel s p1)->(mixtel s p2)->(mixtel s (merge_pre p1 p2)).

Definition lift_mix:(l1,l2:(list A); H:(l1 ⊃ l2);
 s1:(sigtel l1); s2:(sigtel l2))(subsig H s1 s2)->(p:(pre l2))
      (mixtel s2 p)->(mixtel s1 (lift_pre H p)).
```

The following properties are currently in way to be proved (but there are some difficulties to handle big terms). `wellmix_merge` says that `merge_mix` transforms wellformed mixtels in a wellformed mixtel. And `match_mix_merge_1` says that `merge_mix m1 m2` matches `m1`. As merging on mixtels is not commutative (because, when fields of same name have different definitions, you have to choose one), it gives a control of what definition will be choosen by the system in case of conflict: the left will always win.

```
Lemma wellmix_merge: (wellmixed H1 m1) -> (wellmixed H2 m2)->
    (wellmixed (wellpre_merge H1 H2) (merge_mix m1 m2)).

Lemma match_mix_merge_1: (wellmixed H1 m1) -> (wellmixed H2 m2)->
    (match_mix (match_pre_merge_1 H1 H2) (merge_mix m1 m2)  m1).
```

That's the point where the current implementation stops. There are some lemmas left to write in order to check the correctness of `lift_mix`. But we will need to extend the notion of `match_mix` for mixtels of different signatures.

5.5 Using mixtels

The correctness of the operations `merge_mix` and `lift_mix` are not totally proved. But semantic aspects are sufficiently disjoint of operational ones to allow to use them already. We have tried a little example that we describe now, without giving the sources because the syntax is too much ugly. We keep here the informal syntax given for *setoid*.

First, we have coded the *setoid* mixtel mentioned before. Then, we have define *Bool_base* as follow, where *bool* is an inductive type with two constructors, *negb* the negation on booleans, and *Is_true*, the conversion from booleans to `Prop`:

$$
Bool\_base := \left\{ \begin{array}{l}
\texttt{def}\ T:\ \texttt{Type} \\
\quad := \ bool \\
\texttt{def}\ eq:\ T \to T \to \texttt{Prop} \\
\quad := \ \lambda[b_1, b_2 : bool](Is\_true\ (if\ b_1\ then\ b_2\ else\ (negb\ b_2))) \\
\texttt{def}\ eq\_refl:\ \forall(x:T)(eq\ x\ x) \\
\quad := \ \lambda[b : bool] if\ b\ then\ I\ else\ I. \\
\texttt{def}\ neq:\ T \to T \to \texttt{Prop} \\
\quad := \ \lambda[b_1, b_2 : bool](Is\_true\ (if\ (negb\ b_1)\ then\ (negb\ b_2)\ else\ b_2)) \\
\texttt{def}\ neq\_spec:\ \forall(x,y:T)(neq\ x\ y) \to \neg(eq\ x\ y) \\
\quad := \ \lambda[b1, b2 : bool] if\ b1\ \ then\ if\ b2\ then\ \lambda[H, \_]H\ else\ \lambda[\_, H]H \\
\qquad\qquad\qquad\qquad\qquad else\ if\ b2\ then\ \lambda[\_, H]H\ else\ \lambda[H, \_]H
\end{array} \right\}
$$

The aim of the operation is to merge *setoid* and *Bool_base*. As *neq_nrefl* is a field of *setoid* but not in *Bool_base*, we must first *lift Bool_base* by the signature of *setoid*. It makes from *Bool_base* a mixtel with a declared field *neq_nrefl*. Then, we can make the merge. But, the merge is not commutative, because *neq* differs in the two mixtels. We tried the two possibilities. Results are computed with a reasonable efficiency.

5.6 From mixtels to mixDrecs

MixDrecs are wellformed mixtels, on a signature of Drecords (double-free signature of telescope). Wellformness can be discharged via a tactic (which fails when the underlying mixtel is not wellformed). Thus, one could imagine, that the user enters its mixtels, and a macro (which may fail) transform them into mixDrecs. The problem is now to give a syntax to the user.

Currently, the syntax is very bad, because, the user is obliged to describe the tree-like structure of the above figure. With a semantical analysis, it may be possible to parse "mixtels" in the informal syntax presented here, to put them into the tree-like structure. But it is generally not a good idea to make semantic analysis while parsing: here, the user will not see easily in the code the kind of dependences between fields. But these dependencies are fundamental for its understanding of the code.

A good solution, is maybe to mark syntacticaly field identifiers, whose definition is needed, by putting them between delimitors. From this syntax, it should be possible to generate the tree-like structure without typing, but only syntactic manipulation...

6 Conclusion

This paper presents an encoding in Coq of a framework aimed to be used for specifying and implementing a computer algebra library. This framework rests upon a notion of structures, which share some features with object-oriented language: especially, structures can be described the one from the others, by differences.

This encoding is based on a record description in Coq (Drecords), enriched with a class-like notion, also coded in Coq: mixDrecs. They allow to describe a hierarchy of Drecords in a incremental way. In mixDrecs, fields may only be declared, or may be redefined. MixDrecs may be extended by inheritance.

There are some features left to be implemented. Mainly, renaming, mixtel "semantical aspects" and a "user interface" for mixtels.

However, in the current implementation of Coq, this encoding cannot really be used on interesting examples, because of efficiency limitations.

In futures works, we will try to give a more abstracted presentation of this framework, free from the Coq description.

References

1. B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de Doctorat, Université Paris 7, 1999.
2. B. Barras, S. Boutin, C. Cornes, J. Courant, , Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual.* Projet Coq, INRIA-Rocquencourt – CNRS-ENS Lyon, january 1999.
3. S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain, and R. Rioboo. On the way to certify computer algebra systems. In *Calculemus 99 Systems for Integrated Computation and Deduction*, volume 23 of *ENTCS*. Elsevier, 1999.
4. Th. Coquand. An analysis of girard's paradox. In *Symposium on Logic in Computer Science*. Cambridge, MA. IEEE PRESS, 1986.
5. J. Courant. *Un calcul de modules pour les systèmes de types purs.* Thèse de Doctorat, Ecole Normale Supérieure de Lyon, 1998.
6. N. G. de Bruijn. Telescoping mapping in typed lambda calculus. *Information and Computation*, pages 189–204, April 1991.
7. R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st Symposium on Principles of Programming Languages*, pages 123–137. ACM PRESS, 1994.
8. L. Pottier. *Télescopes : des records généralisés définis dans Coq lui-même.* Communication personnelle, sources disponibles sur http://www-sop.inria.fr/croap/CFC/Tel/index.html, 1999.
9. B. Werner. *Une Théorie des Constructions Inductives.* Thèse de doctorat, Université de Paris 7, 1994.