Working with Mathematical Structures in Type Theory

Claudio Sacerdoti Coen and Enrico Tassi

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 – 40127 Bologna, ITALY
{sacerdot,tassi}@cs.unibo.it

Abstract. We address the problem of representing mathematical structures in a proof assistant which: 1) is based on a type theory with dependent types, telescopes and a computational version of Leibniz equality; 2) implements coercive subtyping, accepting multiple coherent paths between type families; 3) implements a restricted form of higher order unification and type reconstruction. We show how to exploit the previous quite common features to reduce the "syntactic" gap between pen&paper and formalised algebra. However, to reach our goal we need to propose unification and type reconstruction heuristics that are slightly different from the ones usually implemented. We have implemented them in Matita.

1 Introduction

It is well known that formalising mathematical concepts in type theory is not straightforward, and one of the most used metrics to describe this difficulty is the gap (in lines of text) between the pen&paper proof, and the formalised version. A motivation for that may be that many intuitive concepts widely used in mathematics, like graphs for example, have no simple and handy representation (see for example the complex hypermap construction used to describe planar maps in the four colour theorem [11]). On the contrary, some widely studied fields of mathematics do have a precise and formal description of the objects they study. The most well known one is algebra, where a rigorous hierarchy of structures is defined and investigated. One may expect that formalising algebra in an interactive theorem prover should be smooth, and that the so called De Bruijn factor should be not so high for that particular subject.

Many papers in the literature [9] give evidence that this is not the case. In this paper we analyse some of the problems that arise in formalising a hierarchy of algebraic structures and we propose a general mechanism that allows to tighten the distance between the algebraic hierarchy as is conceived by mathematicians and the one that can be effectively implemented in type theory.

In particular, we want to be able to formalise the following informal piece of mathematics[1] *without making more information explicit*, expecting the interactive theorem prover to understand it as a mathematician would do.

[1] PlanetMath, definition of Ordered Vector Space.

Example 1. Let k be an ordered field. An ordered vector space over k is a vector space V that is also a poset at the same time, such that the following conditions are satisfied

1. for any $u, v, w \in V$, if $u \leq v$ then $u + w \leq v + w$,
2. if $0 \leq u \in V$ and any $0 < \lambda \in k$, then $0 \leq \lambda u$.

Here is a property that can be immediately verified: $u \leq v$ iff $\lambda u \leq \lambda v$ for any $0 < \lambda$.

We choose this running example instead of the most common example about rings[9,16,3] because we believe the latter to be a little deceiving. Indeed, a ring is usually defined as a triple (C,+,∗) such that (C,+) is a group, (C,∗) is a semigroup, and some distributive properties hold. This definition is imprecise or at least not complete, since it does not list the neutral element and the inverse function of the group. Its real meaning is just that a ring is an *additive* group that is also a *multiplicative* semigroup (on the same carrier) with some distributive properties. Indeed, the latter way of defining structures is often adopted also by mathematicians when the structures become more complex and embed more operations (e.g. vector spaces, Riesz spaces, integration algebras).

Considering again our running example, we want to formalise it using the following syntax[2], and we expect the proof assistant to interpret it as expected:

```
record OrderedVectorSpace : Type := {
 V:> VectorSpace; (* we suppose that V.k is the ordered field *)
 p:> Poset with p.CA_po = V.CA_vs; (* the two carriers must be the same *)
 add_le_compat: ∀u,v,w:V. u ≤ v → u + w ≤ v + w;
 mul_le_compat: ∀u:V.∀α :k. 0 ≤ u → 0 < α → 0 ≤ α ∗ u
}.
lemma trivial: ∀R.∀u,v:R. (∀α . 0 < α → α ∗ u ≤ α ∗ v) → u ≤ v.
```

The first statement declares a *record type*. A record type is a sort of labelled telescope. A *telescope* is just a generalised Σ-type. Inhabitants of a telescope of length n are heavily typed n-tuples $\langle x_1, \ldots, x_n \rangle_{T_1,\ldots,T_n}$ where x_i must have type $T_i x_1 \ldots x_{i-1}$. The heavy types are necessary for type reconstruction. Instead, inhabitants of a record type with n fields are not heavily typed n-tuples, but lighter n-tuples $\langle x_1, \ldots, x_n \rangle_R$ where R is a reference to the record type declaration, which declares once and for all the types of fields. Thus terms containing inhabitants of records are smaller and require less type-checking time than their equivalents that use telescopes.

Beware of the differences between our records — which are implemented, at least as telescopes, in most systems like Coq — and *dependently typed records* "à la Betarte/Tasistro/Pollack" [5,4,8]:

1. there is no "dot" constructor to uniformly access by name fields of any record. Thus the names of these projections must be different, as $.CA_{po}$ and $.CA_{vs}$.

[2] The syntax is the one of the Matita proof assistant, which is quite close to the one of Coq. We reserve λ for lambda-abstraction.

We suppose that ad-hoc projections .k, .v, etc. are automatically declared by the system.

When we write x.v we mean the application of the .v function to x;

2. there is no structural subtyping relation "à la Betarte/Tasistro" between records; however, ad-hoc coercions "à la Pollack" can be declared by the user; in particular, we suppose that when a field is declared using ":>", the relative projection is automatically declared as a coercion by the system;
3. there are no manifest fields "à la Pollack"; the **with** notation is usually understood as syntactic sugar for declaring on-the-fly a new record with a manifest field; however, having no manifest fields in our logic, we will need a different explanation for the **with** type constructor, it will be given in Sec. 2.

When lambda-abstractions and dependent products do not type their variable, the type of the variable must be inferred by the system during type reconstruction. Similarly, all mathematical notation (e.g. "$*$") hides the application of one projection to a record (e.g. "?.$*$" where ? is a placeholder for a particular record). The notation "x:R" can also hide a projection R.CA from R to its carrier.

All projections are monomorphic, in the sense that different structures have different projections to their carrier. All placeholders in projections must be inferred during type reconstruction. This is not a trivial task: in the expression "$\alpha * \mathrm{u} \leq \alpha * \mathrm{w}$" both sides of the inequation are applications of the scalar product of some vector space R (since u and v have been previously assigned the type R.CA); since their result are compared, the system must deduce that the vector space R must also be a poset, hence an ordered vector space.

In the rest of the paper we address the problem of representing mathematical structures in a proof assistant which: 1) is based on a type theory with dependent types, telescopes and a computational version of Leibniz equality; 2) implements coercive subtyping, accepting multiple coherent paths between type families; 3) implements a restricted form of higher order unification and type reconstruction. Lego, Coq, Plastic and Matita are all examples of proof assistants based on such theories. In the next sections we highlight one by one the problems that all these systems face in understanding the syntax of the previous example, proposing solutions that require minimal modifications to the implementation.

2 Dependently typed records in Type Theory

The first problem is understanding the **with** type constructor employed in the example. Pollack and alt. in [8] propose the model for a new type theory having in the syntax primitive dependently typed records, and show how to interpret records in the model. The theory lacks **with**, but it can be easily added to the syntax (adopting the rules proposed in [16]) and also interpreted in the model. However, no non-prototipical proof assistant currently implements primitive dependently typed records.

2.1 Ψ and Σ types

In [16], Randy Pollack shows that dependently typed records with uniform field projections and **with** can be implemented in a type theory extended with inductive types and the induction-recursion principle [10]. However, induction-recursion is also not implemented in most proof assistants, and we are looking for a solution in a simpler framework where we only have primitive records (or even simply primitive telescopes or primitive Σ-types), but no inductive types.

In the same paper, he also shows how to interpret dependently typed records with and without manifest fields in a simpler type theory having only primitive Σ-types and primitive Ψ-types. A Σ-type (Σx:T. P x) is inhabited by heavily typed couples $\langle$w,p$\rangle_{\mathrm{T,P}}$ where w is an inhabitant of the type T and p is an inhabitant of P w. The heavy type annotation is required for type inference. A Ψ-type (Ψx:T. p) is inhabited by heavily typed singletons $\langle$w$\rangle_{\mathrm{T,P,p}}$ where w is an inhabitant of the type T and p is a function mapping x of type T to a value of type P x. The intuitive idea is that $\langle$w, p[w]$\rangle_{\mathrm{T,P}}$ and $\langle$w$\rangle_{\mathrm{T,P},\lambda \mathrm{x:T.\ p[x]}}$ should represent the same couple, where in the first case the value of the second component is opaque, while in the second case it is made manifest (as a function of the first component). However, the two representations actually *are* different and morally equivalent inhabitants of the two types are not convertible, against intuition. We will see later how it is possible to represent couples typed with manifest fields as convertible couples with opaque fields. We will denote by .1 and .2 the first and second projection of a Σ/Ψ-type.

The syntax "Σx:T. P x **with** .2 = t[.1]" can now be understood as syntactic sugar for "Ψx:T. t[x]". The illusion is completed by declaring a coercion from Ψx:T. p to Σx:T. P x so that $\langle$w$\rangle_{\mathrm{T,P,p}}$ is automatically mapped to $\langle$w, p w$\rangle_{\mathrm{T,P}}$ when required.

Most common mathematical structure are records with more than two fields. Pollack explains that such a structure can be understood as a sequence of left-associating[3] nested heavily typed pairs/singletons. For instance, the record r $\equiv\langle$nat, list nat, @$\rangle_R$ of type R := {C : Type; T := list C; app: T → T → T} is represented as[4]

$$\begin{aligned}
&\mathrm{r}_0 \equiv \langle (),\ \mathrm{Type}\rangle_{\mathrm{Unit},\ \lambda \mathrm{C:Unit.\ Type}}\\
&\mathrm{r}_1 \equiv \langle \mathrm{r}_0\rangle_{\Sigma \mathrm{C:Unit.\ Type},\ \lambda \mathrm{x:}(\Sigma \mathrm{C:Unit.\ Type}).\mathrm{Type}_1,\ \lambda \mathrm{y:}(\Sigma \mathrm{C:Unit.\ Type}).\ \mathrm{list\ y.1}}\\
&\mathrm{r} \equiv \langle \mathrm{r}_1,\ @\rangle_{\Psi \mathrm{y:}(\Sigma \mathrm{C:Unit.\ Type}).\ \mathrm{list\ y.1},\ \lambda \mathrm{x:}(\Psi \mathrm{y:}(\Sigma \mathrm{C:Unit.\ Type}).\ \mathrm{list\ y.1}).\ \mathrm{x.2}\to\mathrm{x.2}\to\mathrm{x.2}}
\end{aligned}$$

of type Σx:(Ψ y:(ΣC: Unit. Type). list y.1).x.2 → x.2 → x.2.
However, the deep heavy type annotations are actually useless and make the term extremely large and its type checking inefficient. The interpretation of **with** also becomes more complex, since the nested Σ/Ψ types must be recursively traversed to compute the new type.

[3] In the same paper he also proposes to represent a record type with a right-associating sequence of Σ/Φ types, where a Φ type looks like a Ψ type, but makes it first fields manifest. However, in Sect. 5.2.2 he also argues for the left-associating solution.

[4] Type_1 in the definition of r_1 is the second universe in Luo's ECC [13]. Note that Type has type Type_1

2.2 Weakly manifest types

In this paper we drop Σ/Ψ types in favour of primitive records, whose inhabitants do not require heavy type annotations. However, we are back at the problem of manifest fields: every time the user declares a record type with n fields, to follow closely the approach of Pollack the system should declare 2^n record types having all possible combinations of manifest/opaque fields.

To obtain a new solution for manifest fields we exploit the fact that manifest fields can be declared using **with** and we also go back to the intuition that records with and without manifest fields should all have the same representation. That is, when x ≡3 (x is definitionally equal to 3) and p: P x, the term $\langle$x, p$\rangle_{\mathrm{R}}$ should be both an inhabitant of the record R := { n: nat; H: P n} and of the record R **with** n = 3. Intuitively, the **with** notation should only add in the context the new "hypothesis" x ≡3. However, we want to be able to obtain this effect without extending the type theory with **with** and without adding at run time new equations to the convertibility check. This is partially achieved by approximating x ≡3 with an hypothesis of type x = 3 where "=" is Leibniz polymorphic equality.

To summarise, the idea is to represent an inhabitant of R := {n: nat; H: P n} as a couple $\langle$x, p$\rangle_{\mathrm{R}}$ and an inhabitant of R **with** n=3 as a couple $\langle$c, q$\rangle_{\mathrm{R},\lambda \mathrm{c:R.\ c.n=3}}$ of type Σc:R. c.n=3. Declaring the first projection of the Σ-type as a coercion, the system is able to map every element of R **with** n=3 into an element of R.

However, the illusion is not complete yet: if c is an inhabitant of R **with** n=3, c.1.n (that can be written as c.n because .1 is a coercion) is Leibniz-equal to 3 (because of c.2), but is not convertible to 3. This is problematic since terms there were well-typed in the system presented by Pollack are here rejected. Several concrete example can already be found in our running example: to type u + w ≤ v + w (in the declaration of add_le_compat), the carriers $\mathrm{p.CA_{po}}$ and $\mathrm{V.CA_{vs}}$ must be convertible, whereas they are only Leibniz equal. In principle, it would be possible to avoid the problem by replacing u + w ≤ v + w with $[\mathrm{u+w}]_{\mathrm{p.2}} \leq [\mathrm{v+w}]_{\mathrm{p.2}}$ where $[\_]_{\_}$ is the constant corresponding to Leibniz elimination, i.e. $[\mathrm{x}]_w$ has type Q[M] whenever x has type Q[N] and w has type N=M. However, the insertion of these constants, even if done automatically with a couple of mutual coercions, makes the terms much larger and more difficult to reason about.

2.3 Manifesting coercions

To overcome the problem, consider c of type R **with** n=3 and notice that the lack of conversion can be observed only in c.1.n (which is not definitionally equal to 3) and in all fields of c.1 that come after n (for instance, the second field has type P c.1.n in place of P 3). Moreover, the user never needs to write c.1 anywhere, since c.1 is declared as a coercion. Thus we can try to solve the problem by declaring a different coercion such that c.1.n is definitionally equal to 3. In our example, the coercion[5] is

[5] The name of the coercion is $\mathrm{k}^{\mathrm{n}}_{R}$ verbatim, R and n are not indexes.

---

definition $k^n_R : \forall M : \text{nat}.\ R$ **with** $n{=}M \rightarrow R :=$
$\lambda\, m{:}\text{nat}.\ \lambda\, x{:}(\Sigma c{:}R.\ c.n{=}M).\ \langle M, [x.1.H]_{x.2}\rangle_R$

Once k^n_R is declared as a coercion, $c.H$ is interpreted as $(k^n_R\ 3\ c).H$ which has type $P\ (k^n_R\ 3\ c).n$, which is now definitionally equal to $P\ 3$. Note also that $(k^n_R\ 3\ c).H$ is definitionally equal to $[c.1.H]_{c.2}$ that is definitionally equal to $c.1.H$ when $c.2$ is a closed term of type $c.1.n = 3$. When the latter holds, $c.1.n$ is also definitionally equal to 3, and the manifest type information is actually redundant, according to the initial intuition. The converse holds when the system is proof irrelevant, or, with minor modifications, when Leibniz equality is stated on a decidable type [12].

Coming back to our running example, $u + w \leq v + w$ can now be parsed as the well-typed term

$$u\ (V.+)\ w\ ((k^{CA_{po}}_{Poset}\ V.CA_{vs}\ p).\leq)\ v\ (V.+)\ w$$

Things get a little more complex when **with** is used to change the value of a field f1 that occurs in the type of a second field f2 that occurs in the type of a third field f3. Consider the record type declaration $R := \{ f_1{:}\ T;\ f_2{:}\ P\ f_1;\ f_3{:}\ Q\ f_1\ f_2\}$ and the expression R **with** $f_1{=}\ M$, interpreted as $\Sigma c{:}R.\ c.f_1{=}\ M$. We must find a coercion from R **with** $f_1{=}\ M$ to R declared as follows

---

definition $k^{f_1}_R : \forall M{:}T.\ R$ **with** $f_1 = M \rightarrow R :=$
$\lambda\, M{:}T.\ \lambda\, x{:}(\Sigma c{:}R.\ c.f1{=}M).\ \langle M, [c.1.f_2]_{c.2}, w\rangle$

---

for some w that inhabit $Q\ M\ [c.1.f_2]_{c.2}$ and that must behave as $c.1.f_3$ when $c.1.f_1 \equiv M$. Observe that $c.1.f_3$ has type $Q\ c.1.f_1\ c.1.f_2$, which is definitionally equivalent to $Q\ c.1.f_1\ [c.1.f_2]_{\text{refl}_T\ c.1.f_1}$, where $\text{refl}_T\ c.1.f_1$ is the canonical proof of $c.1.f_1{=}\ c.1.f_1$. Thus, the term w we are looking for is simply $[[c.1.f_3]]_{c.2}$ which has type $Q\ M\ [c.1.f_2]_{c.2}$ where $[[\_]]_\_$ is the constant corresponding to *computational dependent elimination* for Leibniz equality:

---

lemma $[[\_]]_p : Q\ x\ (\text{refl}_A\ \ x) \rightarrow Q\ y\ p.$
where $x : A, y : A, p : x = y, Q : (\forall z.\ x = z \rightarrow \text{Type})$ and $[[M]]_{\text{refl}_A\ x} \equiv M.$

To avoid handling the first field differently from the following, we can always use $[[\_]]_\_$ in place of $[\_]_\_$.

The following derived typing and reduction rules show that our encoding of **with** behaves as expected.

Phi-Start

$$\frac{}{\vdash \emptyset\ \text{valid}}$$

Phi-Cons

$$\frac{\vdash \Phi\ \text{valid} \quad R, l_1, \ldots, l_n\ \text{free in}\ \Phi \qquad T_i : \Pi l_1 : T_1. \ldots . \Pi l_{i-1} : T_{i-1}. Type \quad i \in \{1, \ldots, n\}}{\vdash \Phi, R = \langle l_1 : T_1, \ldots, l_n : T_n\rangle : Type\ \text{valid}}$$

Form

$$\frac{(R = \langle l_1 : T_1, \ldots, l_n : T_n \rangle : Type) \in \Phi \quad \Gamma, l_1 : T_1, \ldots, l_{i-1} : T_{i-1} \vdash a : T_i}{\Gamma \vdash R \text{ with } l_i = a : Type}$$

Intro

$$\frac{(R = \langle l_1 : T_1, \ldots, l_n : T_n \rangle : Type) \in \Phi \quad \Gamma \vdash R \text{ with } l_i = a : Type \\ \Gamma \vdash M_k : T_k\ M_1\ \ldots\ M_{i-1}\ a\ M_{i+1}\ \ldots\ M_{k-1} \quad k \in \{1, \ldots, i-1, i+1, \ldots, n\}}{\Gamma \vdash \langle \langle M_1, \ldots, M_{i-1}, a, M_{i+1}, \ldots, M_n \rangle_R, refl_A\ a \rangle_{R, \lambda r:R.\, a=a} : R \text{ with } l_i = a}$$

Coerc

$$\frac{(R = \langle l_1 : T_1, \ldots, l_n : T_n \rangle : Type) \in \Phi \\ \Gamma \vdash R \text{ with } l_i = a : Type \quad \Gamma \vdash c : R \text{ with } l_i = a}{\Gamma \vdash k_R^{l_i}\ a\ c : R}$$

Coerc-Red

$$\frac{(R = \langle l_1 : T_1, \ldots, l_n : T_n \rangle : Type) \in \Phi}{\Gamma \vdash k_R^{l_i}\ a\ \langle \langle M_1, \ldots, M_n \rangle_R, w \rangle_{R,s} \rhd \langle M_1, \ldots, M_{i-1}, a, [[M_{i+1}]]_w, \ldots, [[M_n]]_w \rangle_R}$$

Proj

$$\frac{(R = \langle l_1 : T_1, \ldots, l_n : T_n \rangle : Type) \in \Phi \\ \Gamma \vdash R \text{ with } l_i = a : Type \quad \Gamma \vdash c : R \text{ with } l_i = a}{\Gamma \vdash (k_R^{l_i}\ a\ c).l_j : T_j\ (k_R\ a\ c).l_1\ \ldots\ (k_R\ a\ c).l_{j-1}}$$

Proj-Red1

$$\frac{(R = \langle l_1 : T_1, \ldots, l_n : T_n \rangle : Type) \in \Phi}{\Gamma \vdash (k_R^{l_i}\ a\ \langle \langle M_1, \ldots, M_n \rangle_R, w \rangle_{R,s}).l_j \rhd M_j}\, j < i$$

Proj-Red2

$$\frac{(R = \langle l_1 : T_1, \ldots, l_n : T_n \rangle : Type) \in \Phi}{\Gamma \vdash (k_R^{l_i}\ a\ c).l_i \rhd a}$$

Proj-Red3

$$\frac{(R = \langle l_1 : T_1, \ldots, l_n : T_n \rangle : Type) \in \Phi}{\Gamma \vdash (k_R^{l_i}\ a\ \langle \langle M_1, \ldots, M_n \rangle_R, w \rangle_{R,s}).l_j \rhd [[M_j]]_w}\, i < j$$

2.4 Deep with construction

In order to interpret the **with** type constructor on "deep" fields, it is sufficient to follow the same schema, changing the coercion to make their sub-records manifest. Formally, when Q := {f: T; l: U} and R := {q: Q; s: S}, we interpret R **with** q.f = M with Σc:R. c.q.f = M and we declare the coercion:

---

definition $k_R^{q.f}$: ∀M:T. R **with** q.f = M → R :=
  λM:T. λx:(Σc:R. c.q.f=M).
    k_R^q (k_Q^f M ⟨x.1.1, x.2⟩$_{Q,\ \lambda q:Q.\ q.f=M}$)
      (**match** x **with** ⟨⟨a, l⟩$_Q$, s⟩$_R$, w⟩$_{R,\ \lambda r:R.\ r.q.f=M}$ ⇒
        ⟨⟨⟨a, l⟩$_Q$, s⟩$_R$, [[refl$_Q$ ⟨a, l⟩$_Q$]]$_w$⟩$_{R,\ \lambda r:R.\ r.q=k_Q^f\ M\ \langle\langle a,l\rangle_Q,\ w\rangle_{Q,\ \lambda q:Q.\ q.f=M}}$)

Note that the computational rule associated to the computational dependent elimination of Leibniz equality is necessary to type the previous coercion:

$$\langle\langle\langle \mathrm{a},\ \mathrm{l}\rangle_{\mathrm{Q}},\ \mathrm{s}\rangle_{\mathrm{R}},[[\mathrm{refl}_{\mathrm{Q}}\langle \mathrm{a},\ \mathrm{l}\rangle_{\mathrm{Q}}]]_{\mathrm{w}}\rangle_{\mathrm{R},\ \lambda \mathrm{r:R.\ r.q=k^{f}_{Q}M}\langle\langle \mathrm{a,l}\rangle_{\mathrm{Q}},\mathrm{w}\rangle_{\mathrm{Q}},\lambda \mathrm{q:Q.\ q.f=M}}$$

is well typed since $\mathrm{refl}_{\mathrm{Q}}\ \langle \mathrm{a},\ \mathrm{l}\rangle_{\mathrm{Q}}$ has type $\langle \mathrm{a},\ \mathrm{l}\rangle_{\mathrm{Q}}= \langle \mathrm{a},\ \mathrm{l}\rangle_{\mathrm{Q}}$ that is equivalent to $\langle \mathrm{a},\ \mathrm{l}\rangle_{\mathrm{Q}}= \mathrm{k}^{f}_{Q}\ \mathrm{a}\ \langle\langle \mathrm{a},\ \mathrm{l}\rangle_{\mathrm{Q}},\ \mathrm{refl}_{\mathrm{T}}\ \mathrm{a}\rangle_{\mathrm{Q},\lambda \mathrm{q:Q.\ q.f=q.f}}$; thus $[[\mathrm{refl}_{\mathrm{Q}}\ \langle \mathrm{a},\ \mathrm{l}\rangle_{\mathrm{Q}}]]_{w}$ has type $\langle \mathrm{a},\ \mathrm{l}\rangle_{\mathrm{Q}}= \mathrm{k}^{f}_{Q}\ \mathrm{M}\ \langle\langle \mathrm{a},\ \mathrm{l}\rangle_{\mathrm{Q}},\ \mathrm{w}\rangle_{\mathrm{Q},\ \lambda \mathrm{q:Q.\ q.f=M}}$.

As expected, $(\mathrm{k}^{\mathrm{q.f}}_{\mathrm{R}}\mathrm{M\ c}).\mathrm{q.f} \triangleright \mathrm{M}$ for all c of type R **with** q.f = M. Due to lack of space we omit all other derived typing and reduction rules associated to the deep **with** construct.

2.5 Nested with contructions

Finally, from the derived typing and reduction rules it is not evident that a type R **with** l_a=M **with** l_b=N can be formed. Surprisingly, this type poses no additional problem. The system simply de-sugars it as

$$\Sigma \mathrm{d}{:}\ (\Sigma \mathrm{c{:}R.\ c.}\, \mathrm{l}_a{=}\mathrm{M}).\ (\mathrm{k}^{l_a}_{R}\ \mathrm{M\ d}).\mathrm{l}_b{=}\mathrm{N}$$

and, as explained in the next section, automatically declares the composite coercion $\mathrm{k}^{\mathrm{l_a,l_b}}_{\mathrm{R}}{:=}\ \lambda \mathrm{M,N,c.\ k}^{l_b}_{R}\ \mathrm{N}\ (\mathrm{k}^{l_a}_{R}\ \mathrm{M\ c})$ as a coercion from R **with** l_a=M **with** l_b=N to R such that: $(\mathrm{k}^{\mathrm{l_a,l_b}}_{\mathrm{R}}\ \mathrm{M\ N\ c}).\mathrm{l}_a \triangleright \mathrm{M}$ and $(\mathrm{k}^{\mathrm{l_a,l_b}}_{\mathrm{R}}\ \mathrm{M\ N\ c}).\mathrm{l}_b \triangleright \mathrm{N}$ and

$$(\mathrm{k}^{\mathrm{l_a,l_b}}_{\mathrm{R}}\ \mathrm{M\ N}\ \langle\langle\langle \mathrm{M}_1,\ldots,\ \mathrm{M}_n\rangle_{R},\ \mathrm{w}_a\rangle,\ \mathrm{w}_b\rangle).\mathrm{l}_i \triangleright \{\{\mathrm{M}_i\}\}_{w_a,w_b}$$

where $\{\{\mathrm{M}_i\}\}_{w_a,w_b}$ is M_i (if i<a and i<b), $[[\mathrm{M}_i]]_{w_a}$ (if a < i < b), $[[\mathrm{M}_i]]_{w_b}$ (if b < i < a), $[[[[\mathrm{M}_i]]_{w_a}]]_{w_b}$ (if a < b < i or b < a < i).

2.6 Signature strengthening and with commutation

To conclude our investigation of record types with manifest fields in type theory, we consider a few additional properties, which are *signature strengthening* and **with** commutation.

An important typing rule for dependently typed records with manifest fields is signature strengthening: a record c of type R must also have type R **with** f = R.f and the other way around. In our setting R **with** f = R.f is interpreted as $\Sigma \mathrm{c{:}R.\ c.f} = \mathrm{c.f}$ and we can couple the coercion $\mathrm{k}^{\mathrm{f}}_{\mathrm{R}}$ from R **with** f = R.f to R with a dual coercion ι_{R} from R to R **with** f = R.f such that: $\forall \mathrm{w.\ k}^{\mathrm{f}}_{\mathrm{R}}(\iota_{\mathrm{R}}(\mathrm{w})) \equiv \mathrm{w}$, $\forall \mathrm{w}.\iota_{\mathrm{R}}(\mathrm{k}^{\mathrm{f}}_{\mathrm{R}}(\mathrm{w})) = \mathrm{w}$ and the latter Leibniz equality is strengthened to definitional equality when w.2 is a closed term or the system is proof irrelevant. The same can be achieved with minor modificaitons when the equality on the type of the f field is decidable.

with commutation is the rule that states the definitional equality of R **with** f=M **with** g=N and R **with** g=N **with** f=M when both expressions are well-typed. In our interpretation, the two types are not convertible since they are represented by different nestings of Σ-types. Moreover, for any label l that

follows f and g in R, the l projection of two canonical inhabitants of the two types built from the same terms are provable equal, but not definitionally equal: in the first case we obtain a term $[[[[\mathrm{M}]]_{w_f}]]_{w_g}$ for some $\mathrm{w_f}$ and $\mathrm{w_g}$, and in the second case we obtain a term $[[[[\mathrm{M}]]_{w_g}]]_{w_f}$. A proof of their equality is simply $[[[[\mathrm{refl_T\ M}]]_{w_f}]]_{w_g}$. Definitional equality holds when when w_f or w_g are canonical terms — in particular when they are closed terms — or if at least one of the two types has a decidable equality. In practice, **with** commutation can often be avoided declaring a pair of mutual coercions between the two commutated types.

2.7 Remarks on code extraction

Algebra has been a remarkable testing area for code extraction, see the constructive proof developed in [9] for example. The encoding of manifest fields presented in the previous sections behave nicely with respect to code extraction. The manifest part of a term is encoded in the equality proof, that is erased during extraction, projections like $\mathrm{k}^{\mathrm{f}}_{\mathrm{R}}$, are extracted to functions that simply replace one field of the record in input. All occurrences of $[[\,\_\,]]\,\_$ are also erased.

3 Structures, inheritance, shared carrier and unification.

Following the ideas from the previous section, we can implement **with** as syntactic sugar: R **with** f = M is parsed as Σc:R. c.f = M and our special coercion $\mathrm{k}^{\mathrm{f}}_{\mathrm{R}}$ respecting definitional equality is defined from Σc:R. c.f=M to R. The scope of the special coercion should be local to the scope of the declaration of a variable of type R **with** f=M. When **with** is used in the declaration of one record field, as in our running example, the scope of the coercion extends to the following record fields, and also to the rest of the script.

As our running example shows, one important use of **with** is in multiple inheritance, in order to share multiply imported sub-structures. For instance, an ordered vector space inherits from a partially ordered set and from a vector space, under the hypothesis that the two carriers (singleton structures) are shared. Since sub-typing is implemented by means of coercions, multiple inheritance with sharing induces multiple coercion paths between nodes in the coercion graph (see Fig. 1; dotted lines hide intermediate structures like groups or Riesz spaces). When the system needs to insert a coercion to map an inhabitant of one type to an inhabitant of a super-type, it must choose one path in the graph. In order to avoid random choices that lead to unwanted interpretations and to type errors (in systems having dependent types), *coherence* of the coercion graph is usually required [14,3]. The graph is coherent when the diagram commutes according to $\beta\eta$-conversion. However in the following we drop η-conversion which is not supported in Coq and Matita.

One interesting case of multiple coherent paths in a graph is constituted by coherent paths between two nodes and the arcs between them obtained by composition of the functions forming one path. Indeed, it is not unusual in large formalisation as CoRN [9] to have very deep inheritance graphs and to need

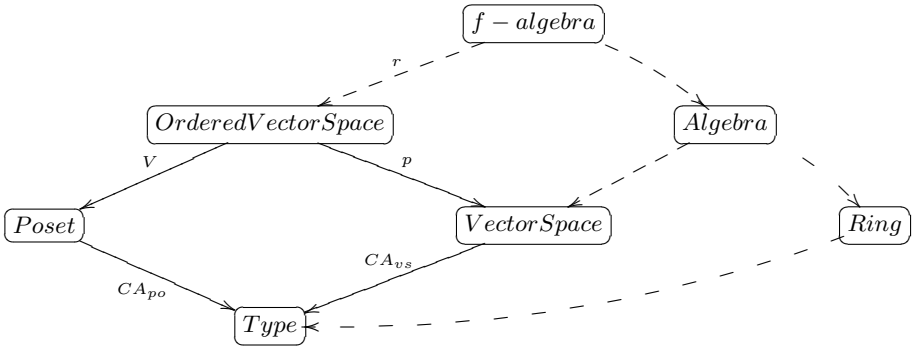

Fig. 1. Inheritance graph from the library of Matita

to cast inhabitants of very deep super-types to the root type. For instance, the expression $\forall$x: R should be understood as $\forall$x: k R where k is a coercion from the ordered, archimedean, complete field of real numbers to its carrier. Without composite coercions, the system needs to introduce a coercion to ordered, archimedean fields, then another one to ordered fields, another one to fields, and then to rings, and so on, generating a very large term and slowing down the type-checker.

If coherent DAGs of coercions pose no problem to conversion, they do for unification, although this aspect has been neglected in the literature. In particular, consider again our running example, whose coercion graph is shown in Fig. 1. Suppose that the user writes the following (false) statement: $\forall$x. $-$x $\leq$ x where $-$x is just syntactic sugar for $-1 *$ x. The statement will be parsed as $\forall$x:?$_1$. -1 ?$_4$.$*$ x ?$_5$.$\leq$ x and the type reconstruction engine will produce the following two unification constraints: ?$_1 \approx$?$_4$.CA$_{vs}$ (since x is passed to ?$_4$.$*$) and ?$_4$.CA$_{vs} \approx$?$_5$.CA$_{po}$ (since $-1 *$ x is passed to ?$_5$.$\leq$). The first constraint is easily solved, "discovering" that x should be an element of a vector space, or a element of one super-structure of a vector space (since ?$_4$ can still be instantiated with a coercion applied to an element of a super-structure). However, the second constraint is problematic since it asks to unify two applications (?$_4$.CA$_{vs}$ and ?$_5$.CA$_{po}$) having different heads. When full higher-order unification is employed, the two heads (two projections) are unfolded and unification will eventually find the right unifier. However, unfolding of constants during unification is too expensive in real world implementations, and higher order unification is never implemented in full generality, preferring an incomplete, but deterministic unification strategy.

Since expansion of constants is not performed during unification, the constraint to be solved is actually a rigid-rigid pattern with two different heads. To avoid failure, we must exploit the coherence of the coercion graph. Indeed, since the arguments of the coercions are metavariables, they can still be instantiated with any possible path in the graph (applied to a final metavariable representing

the structure the path is applied to). For instance, $?_4.\mathrm{CA}_{vs}$ can be instantiated to $?_6.\mathrm{p.CA}_{vs}$ where $?_6$ is an ordered vector space and the vector space $?_4$ is obtained from $?_6$ forgetting the poset structure.

Thus the unification problem is reduced to finding two coherent paths in the graph ending with CA_{vs} and CA_{po}. A solutions is given by paths $?_6.\mathrm{V.CA}_{po}$ and $?_6.\mathrm{p.CA}_{vs}$. Another one by $?_7.\mathrm{r.V.CA}_{po}$ and $?_7.\mathrm{r.p.CA}_{vs}$ where $?_7$ is an f-algebra.

Among all solutions the most general one corresponds to the *pullback* (in categorical terms) of the two coercions, when it exists. In the example, the pullback is given by V and p. All other solutions (e.g. $\mathrm{r}\,.\,\mathrm{V}$ and $\mathrm{r}\,.\,\mathrm{p}$) factor trough it.

If the pullback does not exist (i.e. there are different solutions admitting anti-unifiers), the system can just pick one solution randomly, warning the user about the arbitrary choice. Coercion graphs for algebraic structures usually enjoy the property that there exists a pullback for every pairs of coercions with the same target.

Finally, note that the Coq system does not handle composite coercions, since these would lead to multiple paths between the same types. However, long chains of coercions are somehow problematic for proof extraction. According to private communication, an early experiment in auto-packing chains of coercions was attempted, but dropped because of the kind of unification problems just explained. After implementing the described procedure for unification up to coherence in Matita, we were able to implement coercion packing.

4 Type reconstruction with unification and coercions.

Syntactic de-sugaring of **with** expression for a large hierarchy of mathematical structures has been made by hand in Matita, proving the feasibility of the approach. In particular, combining de-sugaring with the unification up to coherence procedure described in the previous paragraph, we are able to write the first part of our running example in Matita.

The statement of the trivial lemma, however, is not accepted yet. To fully understand which problem still arises, we need to introduce type reconstruction and coercion synthesis algorithms more formally.

4.1 Type reconstruction algorithm

Coq, Lego and Matita use similar algorithms based on [1,2,17] to insert coercions in user provided ill-typed terms to make them well-typed. Coercions can be inserted in three different positions: around arguments expected to be sorts (e.g. when typing bound variables), around application heads (to fix the arity of the head), and around application arguments (to fix their types).

In the following presentation the coercion synthesis judgement

$$\Gamma \vdash t \quad \overset{\mathcal{R}}{\rightsquigarrow} \quad \Gamma \vdash t' : T$$

means that a term t in a well-typed context Γ can be internalised as a well-typed term t' of type T; t' is obtained by inserting coercions in t. s ranges over sorts (Prop or Type in CIC). c ranges over declared coercions.

The rules given in [1,17] are the following:

lam

$$\frac{\Gamma \vdash T \overset{\mathcal{R}}{\rightsquigarrow} \Gamma \vdash T' : T'' \quad \Gamma \vdash T'' \not\equiv s \quad \Gamma \vdash c : T'' \to s' \quad \Gamma, x : c\ T' \vdash b \overset{\mathcal{R}}{\rightsquigarrow} \Gamma, x : c\ T' \vdash b' : U}{\Gamma \vdash \lambda x : T.b \overset{\mathcal{R}}{\rightsquigarrow} \Gamma \vdash \lambda x : c\ T'.b' : \Pi x : c\ T'.U}$$

prod

$$\frac{\Gamma \vdash T \overset{\mathcal{R}}{\rightsquigarrow} \Gamma \vdash T' : T'' \quad \Gamma \vdash T'' \not\equiv s \quad \Gamma \vdash c : T'' \to s' \quad \Gamma, x : c\ T' \vdash U \overset{\mathcal{R}}{\rightsquigarrow} \Gamma, x : c\ T' \vdash U' : s}{\Gamma \vdash \Pi x : T.U \overset{\mathcal{R}}{\rightsquigarrow} \Gamma \vdash \Pi x : c\ T'.U' : s}$$

app-head

$$\frac{\Gamma \vdash f \overset{\mathcal{R}}{\rightsquigarrow} \Gamma \vdash f' : F \quad \Gamma \vdash F \not\equiv \Pi x : B.C \quad \Gamma \vdash c : F \to \Pi x : A.U \quad \Gamma \vdash (c\ f')\ a \overset{\mathcal{R}}{\rightsquigarrow} \Gamma \vdash u : U'}{\Gamma \vdash f\ a \overset{\mathcal{R}}{\rightsquigarrow} \Gamma \vdash u : U'}$$

app-arg

$$\frac{\Gamma \vdash f \overset{\mathcal{R}}{\rightsquigarrow} \Gamma \vdash f' : \Pi x : B.U \quad \Gamma \vdash a \overset{\mathcal{R}}{\rightsquigarrow} \Gamma \vdash a' : A \quad \Gamma \vdash A \not\equiv B \quad \Gamma \vdash c : A \to B}{\Gamma \vdash f\ a \overset{\mathcal{R}}{\rightsquigarrow} \Gamma \vdash f'\ (c\ a') : U[c\ a'/x]}$$

All these rule have a negative precondition. If the precondition is positive, then the coercion is not needed and thus not inserted.

These rules have been employed in the type reconstruction algorithm of Coq and Matita. The type reconstruction algorithm is obtained from the syntax directed type inference algorithm by adding metavariables [15] in the calculus (standing for missing sub-terms) and by replacing conversion ($\equiv$) with unification ($\approx$). We thus extend our judgement with an environment Θ that is a list of metavariable declarations ($\Gamma \vdash ?_i : T$) or metavariable instantiations. With $?_i^s$ we state a metavariable that can only be instantiated with a sort (Prop or Type in CIC); with $?_i^c$ a metavariable that can only be instantiated with a coercion. $\overset{\mathcal{R}}{\rightsquigarrow}$ can now instantiate metavariables performing unification, thus the whole judgement is extended to

$$\Theta : \Gamma \vdash t \overset{\mathcal{R}}{\rightsquigarrow} \Theta' : \Gamma \vdash t' : T$$

Insertion of coercions interacts badly with open terms. Consider, for instance, the following example and assume a coercion from natural numbers to integers.

$\forall$ P : int $\to$ Prop. $\forall$ Q : nat $\to$ Prop. $\forall$ b. P b $\wedge$ Q b.

Here P b is processed before Q b. The rule **app-arg** is not applied, since the type of b is a metavariable $?_i$ and $?_i \approx \mathsf{int}$. Then Q b is processed, but now b has type int, $\mathsf{int} \not\approx \mathsf{nat}$ and there is no coercion from int to nat. The problem here is that a coercion was needed around the first occurrence of b but since its type was flexible **app-arg** was not triggered.

To solve the problem, one important step is the realization that rules that insert coercions and rules that do not are occurrences of the same rule when identities are considered as coercions. In [6,7], Chen proposes an unified set of rules that also employes least upper bounds (*lub*) in the coercion graph to compute less coerced solutions. Chen's rule for application adapted with metavariables in place of coercions is the following:

I-app

$$\frac{\Theta : \Gamma \vdash f \overset{\mathcal{R}}{\rightsquigarrow} \Theta' : \Gamma \vdash f' : C \quad \Theta'' = \Theta', \Gamma \vdash ?_i^c : C \to_{lub} \Pi x : A.B \qquad \Theta'' : \Gamma \vdash a \overset{\mathcal{R}}{\rightsquigarrow} \Theta''' : \Gamma \vdash a' : A' \quad \Theta'''' = \Theta''', \Gamma \vdash ?_j^c : A' \to A}{\Theta : \Gamma \vdash f\ a \overset{\mathcal{R}}{\rightsquigarrow} \Theta'''' : \Gamma \vdash (?_i^c\ f')\ (?_j^c\ a') : B[(?_j^c\ a')/x]}$$

Adopting this rule, the problematic example above is accepted:
$\forall\, \mathrm{P} : \mathsf{int} \to \mathrm{Prop}.\ \forall\, \mathrm{Q} : \mathsf{nat} \to \mathrm{Prop}.\ \forall\, \mathrm{b}.\ \mathrm{P\ b} \wedge \mathrm{Q\ b}$ is understood as $\forall\, \mathrm{P} : \mathsf{int} \to \mathrm{Prop}.\ \forall\, \mathrm{Q} : \mathsf{nat} \to \mathrm{Prop}.\ \forall\, \mathrm{b}\colon ?_1.\ (?_2^c\ \mathrm{P})\ (?_3^c\ \mathrm{b}) \wedge (?_4^c\ \mathrm{Q})\ (?_5^c\ \mathrm{b})$ where $?_1$ can be instantiated with nat, $?_3^c$ with the coercion from nat to int and all other coercions with the identity.

From this example it is clear that Chen's rules modified with metavariables are able to type every term generating a large number of constraints that must inefficiently be solved at the very end looking at the coercion graph.

Note, however, that rule **I-appl** in its full generality is not required to accept our running example. We believe this not to be a coincidence. Indeed, most formulae in the algebraic domain are of a particular shape: 1) universal quantifications are either on structure types (e.g. $\forall G$: Group) or elements of some structure (e.g. $\forall g : G$ to be understood as $\forall g : G.CA$); 2) functions either take structures in input (e.g. $G \times G$); or they manipulate structure elements whose domain is left implicit (e.g. $\_^- : M.CA \to nat \to M.CA$ for some monoid M). In particular, all operations in a structure are functions of the second kind.

Under this assumption, rule **I-appl** can be relaxed to rule app-head-arg, which is given below together with the rules for explicitly and implicitly typed universal quantification.

lam-explicit

$$\frac{\Theta : \Gamma \vdash T \overset{\mathcal{R}}{\rightsquigarrow} \Theta' : \Gamma \vdash T' : T'' \qquad \Theta'' = \Theta', \Gamma \vdash ?_j^s, \Gamma \vdash ?_i^c : T'' \to ?_j^s \quad \Theta'' : \Gamma, x :?_i^c\ T' \vdash b \overset{\mathcal{R}}{\rightsquigarrow} \Theta''' : \Gamma \vdash b' : U}{\Theta : \Gamma \vdash \lambda x : T.b \overset{\mathcal{R}}{\rightsquigarrow} \Theta''' : \Gamma \vdash \lambda x :?_i^c\ T'.b' : \Pi x :?_i^c\ T.U}$$

lam-implicit

$$\frac{\Theta' = \Theta, \Gamma \vdash ?_i :?^s_j \quad \Theta' : \Gamma, x :?_i \vdash b \quad \overset{\mathcal{R}}{\rightsquigarrow} \quad \Theta'' : \Gamma, x :?_i \vdash b' : T'}{\Theta : \Gamma \vdash \lambda x :?.b \quad \overset{\mathcal{R}}{\rightsquigarrow} \quad \Theta'' : \Gamma \vdash \lambda x :?_i.b' : \Pi x :?_i.T'}$$

prod-explicit

$$\frac{\begin{array}{c}\Theta : \Gamma \vdash T \quad \overset{\mathcal{R}}{\rightsquigarrow} \quad \Theta' : \Gamma \vdash T' : T'' \quad \Theta'' = \Theta', \Gamma \vdash ?^s_j, \Gamma \vdash ?^c_i : T'' \to ?^s_j \\ \Theta'' : \Gamma, x :?^c_i\ T' \vdash U \quad \overset{\mathcal{R}}{\rightsquigarrow} \quad \Theta''' : \Gamma, x :?^c_i\ T' \vdash U' : s\end{array}}{\Theta : \Gamma \vdash \Pi x : T.U \quad \overset{\mathcal{R}}{\rightsquigarrow} \quad \Theta : \Gamma \vdash \Pi x :?^c_i\ T'.U' : s}$$

prod-implicit

$$\frac{\Theta' = \Theta, \Gamma \vdash ?^s_j, \Gamma \vdash ?_i :?^s_j \quad \Theta' : \Gamma, x :?_i \vdash U \quad \overset{\mathcal{R}}{\rightsquigarrow} \quad \Theta'' : \Gamma, x :?_i \vdash U' : s}{\Theta : \Gamma \vdash \Pi x :?.U \quad \overset{\mathcal{R}}{\rightsquigarrow} \quad \Theta'' : \Gamma \vdash \Pi x :?_i.U' : s}$$

app-head-arg

$$\frac{\begin{array}{c}\Theta : \Gamma \vdash f \quad \overset{\mathcal{R}}{\rightsquigarrow} \quad \Theta' : \Gamma \vdash f' : F \quad \Theta' : \Gamma \vdash a \quad \overset{\mathcal{R}}{\rightsquigarrow} \quad \Theta'' : \Gamma \vdash a' : A \\ \langle \Theta''', c_f : F \to \Pi x : T \to U \rangle = lub_\Pi(\Theta'', \Gamma, F) \\ \langle \Theta'''', c_a : A \to T \rangle = lub(\Theta''', \Gamma, A, T)\end{array}}{\Theta : \Gamma \vdash f\ a \quad \overset{\mathcal{R}}{\rightsquigarrow} \quad \Theta'''' : \Gamma \vdash (c_f\ f')\ (c_a\ a') : U'[c_a\ a'/x]}$$

The auxiliary function $lub_\Pi(\Theta, \Gamma, F)$ returns a couple $\langle \Theta', c : T \to \Pi x : U.V \rangle$ such that in Θ' and Γ we have $F \approx T$ and $\Pi x : U.V$ is the least upper bound of all solutions in the coercion graph. Note that, according to the restrictions we made, F cannot be a flexible term. Thus the computation of the least upper bound is as in Chen.

The auxiliary function $lub(\Theta, \Gamma, T, U)$ returns a couple $\langle \Theta', c : T \to U \rangle$ such that the type of the coercion c can be unified to $T \to U$ in Γ and Θ' and the coercion is the least upper bound of the solutions in the coercion graph. The *lub* function is defined according to the restriction on functions in the algebraic language. Indeed, by hypothesis we must only consider the following two cases corresponding to the two kind of functions in our language:

1. f has type $S \to T$ for some structure type S and some type T and a has type $?_1$ or it has type R for some structure type R. In the first case the *lub* is the identity coercion and $?_1$ is unified with S. In the second case the *lub* is the coercion from R to S in the coercion graph.
2. f has type $?_1.CA_R \to ?$ and a has type $?_2$ or it has type $?_2.CA_S$. In both cases the *lub* is the identity coercion and the type of a is unified with $?_1.CA_R$ exploiting the coercion graph as explained in Sect. 3.

Finally, as expected, our rules are not complete outside the fragment we choose. For instance, assume a coercion from natural numbers to integers and consider the following statement:

lemma broken : $\forall$ f : ($\forall$ A : Type. A $\rightarrow$A $\rightarrow$ Prop). f $?_i$ 3 -2 $\wedge$f $?_i$ -2 3.

Here the type of f is completely specified, and the rule **prod-explicit** is applied. The term f $?_i$, which is outside our fragment, has type $?_i \rightarrow$ ($?_i \rightarrow$ Prop) and it is passed an argument of type nat the first time and an argument of type int the second time. No backtracking free algorithm would be able to type this term.

5 Conclusions.

In this paper we addressed the problem of representing mathematical structures in a proof assistant based on a type theory with dependent types, telescopes and a computational version of Leibniz equality. We show how to represent dependently typed records with manifest fields in type theory exploiting coercive subtyping and unification up to coherence in coercion graphs.

We made a significant advancement with respect to [16] since we do not require induction-recursion to have the **with** construct. Unification up to coherence seems also a novel approach.

We have also identified a significant fragment of algebra for which a backtracking-free coercion-aware type reconstruction algorithm can be efficiently implemented. This latter result requires further investigation (to enlarge the fragment) and a formal proof of completeness.

References

1. Anthony Bailey. Coercion synthesis in computer implementations of type-theoretic frameworks. In *TYPES '96: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 9–27, London, UK, 1998. Springer-Verlag.
2. Anthony Bailey. *The Machine-Checked Literate Formalisation Of Algebra In Type Theory*. PhD thesis, University of Manchester, 1998.
3. Gilles Barthe. Implicit coercions in type systems. In *Types for Proofs and Programs: International Workshop, TYPES 1995*, pages 1–15, 1995.
4. Gustavo Betarte and Alvaro Tasistro. Formalization of systems of algebras using dependent record types and subtyping: An example. In *Proceedings of the 7th. Nordic workshop on Programming Theory, Gothenburg*, 1995.
5. Gustavo Betarte and Alvaro Tasistro. Extension of Martin-Löf's type theory with record types and subtyping. In *Twenty-five Years of Constructive Type Theory*. Oxford Science Publications, 1998.
6. Gang Chen. *Subtyping, Type Conversion and Transitivity Elimination*. PhD thesis, University Paris 7, 1998.
7. Gang Chen. Coercive subtyping for the calculus of constructions. In *The 30th Annual ACM SIGPLAN - SIGACT Symposium on Principle of Programming Language (POPL)*, 2003.
8. Thierry Coquand, Randy Pollack, and Makoto Takeyama. A logical framework with dependently typed records. *Fundamenta Informaticae*, 65(1-2):113–134, 2005.
9. Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-corn, the constructive coq repository at nijmegen. In *MKM*, pages 88–103, 2004.

10. Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), 2000.
11. Georges Gonthier. A computer-checked proof of the four-colour theorem. Available at `http://research.microsoft.com/~gonthier/4colproof.pdf`.
12. Michael Hedberg. Unpublished proof formalized in lego by T. Kleymann and in coq by B. Barras. `http://coq.inria.fr/library/Coq.Logic.Eqdep_dec.html`.
13. Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.
14. Zhaohui Luo. Coercive subtyping. *J. Logic and Computation*, 9(1):105–130, 1999.
15. César Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.
16. Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.
17. Amokrane Saibi. Typing algorithm in type theory with inheritance. In *The 24th Annual ACM SIGPLAN - SIGACT Symposium on Principle of Programming Language (POPL)*, 1997.