The Rooster and the Syntactic Bracket

Hugo Herbelin, Arnaud Spiwack

The Rooster and the Syntactic Bracket*

Hugo Herbelin[1] and Arnaud Spiwack[2]

1 **Inria Paris-Rocquencourt**
Paris, France
hugo.herbelin@inria.fr
2 **Inria Paris-Rocquencourt**
Paris, France
arnaud@spiwack.net

Abstract

We propose an extension of pure type systems with an algebraic presentation of inductive and co-inductive type families with proper indices. This type theory supports coercions toward from smaller sorts to bigger sorts via explicit type construction, as well as impredicative sorts. Type families in impredicative sorts are constructed with a bracketing operation. The necessary restrictions of pattern-matching from impredicative sorts to types are confined to the bracketing construct. This type theory gives an alternative presentation to the calculus of inductive constructions on which the Coq proof assistant is an implementation.

1998 ACM Subject Classification F.3.3 Studies of Program Constructs

Keywords and phrases Coq, Calculus of inductive constructions, Impredicativity, Strictly positive type families, Inductive type families

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

In the Coq proof assistant [14] inductive types are treated as toplevel definitions. If it makes sense from a convenience or an efficiency point of view, the monolithic nature of the definitions make it hard to describe what they precisely mean. As a matter of fact, inductive definitions mean different things depending on the type they are defined in: specifically, some types are interpreted differently in impredicative sorts like **Prop** or the impredicative variant of **Set**.

In this article, we present a calculus of inductive and co-inductive constructions where inductive and co-inductive types are presented algebraically. The algebraic presentation is an extension of a PTS [3] with inductive and co-inductive type families. Thanks to its modularity, it is meant to serve as a description which is simpler to expose and more mathematically amenable than the monolithic scheme which is found in a practical system such as Coq. For the sake of clarity, the system is given with a single universe and explicit subtyping, although Coq has an unbounded cumulative hierarchy of universes and implicit subtyping. Apart from these technicalities, it is believed that our calculus of algebraic inductive and co-inductive constructions expresses all the features of the Set-impredicative Calculus of Inductive Constructions that Coq implements, *e.g.* in its version 8.4 when launched with option -impredicative-set.

* This research has received funding from the European Research Council under the FP7 grant agreement 278673, Project MemCAD

$$\frac{\Gamma \vdash A : s \quad x \text{ is fresh in } \Gamma}{\Gamma, x{:}A \vdash x : A} \qquad \frac{\Gamma \vdash u : A \quad \Gamma \vdash B : s \quad x \text{ is fresh in } \Gamma}{\Gamma, x{:}B \vdash u : A}$$

$$\frac{\Gamma \vdash \prod_{x:A} B : s \quad \Gamma, x{:}A \vdash u : B}{\Gamma \vdash \lambda x^A.\, u : \prod_{x:A} B} \qquad \frac{\Gamma \vdash u : \prod_{x:A} B \quad \Gamma \vdash v : A}{\Gamma \vdash u\, v : B[\, x \setminus v\,]}$$

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash B : s \quad A \equiv B}{\Gamma \vdash u : B} \qquad \left(\lambda x^A.\, u\right) v \rightsquigarrow u[\, x \setminus v\,]$$

Figure 1 Generic rules of PTS

This work draws most of its inspiration from Morris *&* al [9, 10] for the algebraic presentation of inductive type families in a predicative sort, and Awodey *&* Bauer [2] for the treatment of impredicative sorts.

We use examples from Coq to illustrate the algebraic presentation. To differentiate expressions in Coq from expressions in the algebraic presentation, the former are typeset in a sans-serif font and the latter in a roman font.

2 Pure type systems

To model the type system of Coq, we start with the classic presentation of pure type systems (PTS) of Barendregt [3], which we will then extend to model type families. A PTS is characterised by a *single* syntactic category of terms which are used both as λ-terms and as types. It has a single form of typing judgment $\Gamma \vdash u : A$, where u and A are terms, and Γ a context assigning terms to variables. A PTS has a set of *sorts*, which we shall denote schematically by the symbol s. Every sort is an atomic term. A PTS has a conversion relation $u \equiv v$. Here we diverge from the presentation of [3] which always uses β-conversion. Coq, on the other hand, uses $\beta\eta$-conversion on the fragment described in this section. In this article we will take the conversion rule as abstract, not even requiring it to be decidable. We will only require that it contains all the reduction rules which are given in the form $u \rightsquigarrow v$ (in this section, we only have β-reduction).

The typing rules of a PTS comprise of a set of generic rules given in Figure 1, together with a number of rules of the form

$$\frac{}{\vdash s_1 : s_2}$$

called axioms, and rules of the form

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \prod_{x:A} B : s_3}$$

of product formation rules. As usual we write $A \to B$ for $\prod_{x:A} B$ when x does not bind a variable in B.

As a starting point of the algebraic presentation, we shall use a PTS with two sorts, Type

and $\Box$, together with the following axiom:

$$\frac{}{\Gamma \vdash \mathrm{Type} : \Box}$$

and the following product formation rule:

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \prod_{x:A} B : \max s_1\, s_2}$$

where $\max s\, s = s$ and $\max \mathrm{Type}\, \Box = \max \Box\, \mathrm{Type} = \Box$

The sorts Type and $\Box$ are predicative. The sort $\Box$ plays a technical role in allowing type variable and the formation of type-level functions; it cannot, however, be referenced in terms. In the following sections, $\Box$ will also be used to be able to define types by pattern-matching (*strong elimination*).

To model the entire Coq system, Type and $\Box$ would be replaced with a hierarchy of predicative sorts Type_i, such that

$$\frac{}{\Gamma \vdash \mathrm{Type}_i : \mathrm{Type}_{i+1}}$$

are axioms. Adapting the presentation to a hierarchy of sorts is straightforward, but in the interest of keeping to the heart of the matter we give a presentation with two sorts.

3 Inductive type families

We shall now extend the algebraic presentation with a notion of *inductive type families* to model (dependent) datatypes. In this section we will stay in the predicative fragment of Coq.

Contrary to the inductive types of Coq, where inductive definitions must be *named* at *toplevel*, like in:

```
Inductive Even : Type :=
| eo : Even
| es : Odd → Even
with Odd : Type :=
| os : Even → Odd.
```

the presentation given here is essentially anonymous, and inductive families need not be defined at toplevel prior to use. Mutually inductive types such as Even and Odd are not modelled directly, rather they are encoded using an index:

```
Inductive EvenOdd : bool → Type :=
| eo : EvenOdd true
| es : EvenOdd false → EvenOdd true
| os : EvenOdd true → EvenOdd false.
Definition Even := EvenOdd true.
Definition Odd := EvenOdd false.
```

This encoding works as long as all the mutual definitions are all in the same sort. A variant for mutual definition involving **Type** and **Prop** is demonstrated in Section 4.3. When the types being defined are in different predicative sorts, however, we have to resort to another encoding which involves nested datatypes [4, Section 8.6].

We will not explore the latter kind of mutual definition. However, nested datatypes – where recursive calls occur as arguments of another type – such as:

Inductive List (A:**Type**) : **Type** :=
| nil : List A
| cons : A → List A → List A.
Inductive Tree : **Type** :=
| node : List Tree → Tree.

are indeed modelled in the algebraic presentation.

3.1 Regular types

To be able to traverse terms of inductive type, the core PTS constructions is extended with a *recursive fixed point* on functions:

$$\frac{\Gamma \vdash \prod_{x_1:A_1,\ldots,x_{n-1}:A_{n-1},x_n:A_n} B : s \qquad \text{guarded}\, f\, x_1 \ldots x_{n-1}\, x_n\, u \qquad \Gamma, f{:}\prod_{x_1:A_1,\ldots,x_{n-1}:A_{n-1},x_n:A_n} B, x_1{:}A_1, \ldots, x_{n-1}{:}A_{n-1}, x_n{:}A_n \vdash u : B}{\Gamma \vdash \text{fix}\ f\ x_1{:}A_1\ \ldots\ x_{n-1}{:}A_{n-1}\ x_n{:}A_n \Rightarrow u : \prod_{x_1:A_1,\ldots,x_{n-1}:A_{n-1},x_n:A_n} B}$$

Recursive fixed points are unfolded when fully applied

$$(\text{fix}\ f\ x_1{:}A_1\ \ldots\ x_n{:}A_n \Rightarrow u)\, v_1 \ldots v_n \rightsquigarrow u[\, x_i \setminus v_i \,]$$

To ensure strong normalisation, this reduction rule is limited, and a guard condition is imposed on the recursive calls to f. It is not, however, the object of this article to discuss these restriction or the guard condition. Briefly, Coq currently relies on a single *structural* argument in the block $x_1, \ldots, x_n$: fixed points are not unfolded until their structural argument starts with a constructor, and the guard condition ensures that each recursive call is performed on a subterm of said structural argument, for a relaxed notion of subterm. Other possibilities exist: Agda2 [11] uses any number of arguments as structural, and tries to find a lexicographic ordering. Yet another possibilities is to use sized types [1]. We shall simply assume that an adequate guard condition is given.

We now extend the grammar of type constructors. The presentation of this article is largely inspired by the synthetic definition of *strictly positive families* by Morris *&* al [9, 10], but is adapted to *intensional type theory*. The presentation of [9, 10] is designed for generic programming inside a type theory, they give codes for strictly positive families which are then decoded into an actual type of the ambient theory. No elimination principle needs to be given for the strictly positive families, as they are implicit in their decoding. Here, we are defining the syntax of inductive type families, including their elimination rules.

The regular type constructors, whose typing rules are given in Figure 2, are the empty type 0, the unit type 1, and the sum of two types. The elimination rules are given in the form of *dependent pattern-matching* with a syntax made to remind of that of Coq. We shall

Sum type

$$\frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma \vdash B : \mathrm{Type}}{\Gamma \vdash A + B : \mathrm{Type}}$$

$$\frac{\Gamma \vdash A + B : s \quad \Gamma \vdash t : A}{\Gamma \vdash \mathrm{inl}\, t : A + B} \qquad \frac{\Gamma \vdash A + B : s \quad \Gamma \vdash u : B}{\Gamma \vdash \mathrm{inr}\, u : A + B}$$

$$\frac{\Gamma \vdash u : A + B \quad \Gamma, x{:}A + B \vdash P : s \quad \Gamma, y{:}A \vdash v : P[\,x \setminus \mathrm{inl}\, y\,] \quad \Gamma, z{:}A \vdash w : P[\,x \setminus \mathrm{inr}\, z\,]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{inl}\, y \Rightarrow v \\ \mid\ \mathrm{inr}\, z \Rightarrow w\end{array} : P[\,x \setminus u\,]}$$

$$\begin{array}{l}\mathrm{match}\ \mathrm{inl}\, u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{inl}\, y \Rightarrow v \\ \mid\ \mathrm{inr}\, z \Rightarrow w\end{array} \rightsquigarrow v[\,y \setminus u\,]$$

$$\begin{array}{l}\mathrm{match}\ \mathrm{inr}\, u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{inl}\, y \Rightarrow v \\ \mid\ \mathrm{inr}\, z \Rightarrow w\end{array} \rightsquigarrow w[\,z \setminus u\,]$$

Unit type

$$\frac{}{\Gamma \vdash 1 : \mathrm{Type}} \qquad \frac{}{\Gamma \vdash () : 1}$$

$$\frac{\Gamma \vdash u : 1 \quad \Gamma, x{:}1 \vdash P : s \quad \Gamma \vdash v : P[\,x \setminus ()\,]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ () \Rightarrow v\end{array} : P[\,x \setminus u\,]}$$

$$\begin{array}{l}\mathrm{match}\ ()\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ () \Rightarrow v\end{array} \rightsquigarrow v$$

Empty type

$$\frac{}{\Gamma \vdash 0 : \mathrm{Type}} \qquad \frac{\Gamma \vdash u : 0 \quad \Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash \mathrm{match}\ u\ \mathrm{return}\ A\ \mathrm{with}\ \cdot : A}$$

Figure 2 Regular type constructors

often omit the typing predicate when it is clear from the context, especially when it does not depend on the branch. With this material we can define a first example type, namely the booleans:

$$\begin{cases} \mathbb{B} = 1 + 1 \\ \text{true} = \text{inl}\,() \\ \text{false} = \text{inr}\,() \end{cases}$$

3.2 Inductive type families

Inductive families differ from regular inductive types in that they are parametrised by *indices*, that is they are functions $F : A \to \text{Type}$ for some A. An inductive family of the form $\lambda x^A.\, R$, is said to be *uniformly parametrised* by A. In general, inductive families are not uniformly parametrised: the value of the index is allowed to vary in recursive calls, and constructors may build values of $F\,x$ for certain x only. Remember, for instance, the EvenOdd family:

```
Inductive EvenOdd : bool → Type :=
| eo : EvenOdd true
| es : EvenOdd false → EvenOdd true
| os : EvenOdd true → EvenOdd false.
```

The inductive family constructors, presented in Figure 3, warrant individual discussion. First, notice that as a simplifying hypothesis, inductive families have exactly one index. This is, of course, not a limitation in expressive power as multiple indices can be encoded as a dependent sum, and the unit type allows us to write families without an index.

The construction $\mu X^{A \to \text{Type}}.\, F$ constructs the *inductive fixed point* of F. It acts on type families, because indices vary through recursive calls to X. To be able to form an inductive fixed point, occurrences of X must be strictly positive in F, rules for strict positivity are given in Figure 4. The rules of Figure 4 are a simple set which suits the needs of this article, however in practice, we may want to consider strict positivity rules involving elimination rules and a finer treatment of application. Strict positivity ensures that no non-terminating term can be written without recursive fixed points, so that the guard condition suffices to enforce termination. Paradoxes which can be derived from non-positive or non-strictly positive inductive fixed points can be found in [13, Chapter 4, Section 4.2][8, Chapter 3][4, Chapter 8]. To avoid clutter, we give a presentation where inductive fixed points can be freely rolled and unrolled thanks to the conversion. An alternative can be to give an explicit term constructor for fixed points, see Section 3.4.

We will also use an inductive fixed point on nullary families, defined as:

$$\mu X^{\text{Type}}.\, F = \left(\mu Y^{1 \to \text{Type}}.\, F[\, X \setminus Y\,()\,]\right)()$$

from which we have that $\mu X^{\text{Type}}.\, F$ can be freely rolled from or unrolled to $F\left[\, X \setminus \mu X^{\text{Type}}.\, F\,\right]$.

With inductive fixed points, we can, for instance, define the accessibility predicate. In Coq:

```
Inductive Acc (A:Type) (R:A→A→Type) (x:A) : Type :=
| acc_intro : (forall y:A, R y x → Acc A R y) → Acc A R x.
```

Inductive fixed point

$$\frac{\Gamma \vdash A : s \quad \Gamma, X{:}A \to \mathrm{Type} \vdash F : A \to \mathrm{Type} \quad \mathsf{sp}_X\, F}{\Gamma \vdash \mu X^{A\to \mathrm{Type}}.\, F : A \to \mathrm{Type}}$$

$$\mu X^{A\to s}.\, F \equiv F\big[\, X \setminus \mu X^{A\to s}.\, F \,\big]$$

Proper indices

$$\frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma \vdash B : \mathrm{Type} \quad \Gamma, x{:}A \vdash T : \mathrm{Type} \quad \Gamma, x{:}A \vdash f : B}{\Gamma \vdash \textstyle\sum^f_{x:A} T : B \to \mathrm{Type}}$$

$$\frac{\Gamma \vdash \textstyle\sum^f_{x:A} T : B \to s \quad \Gamma \vdash u : A \quad \Gamma \vdash v : T[\, x \setminus u \,]}{\Gamma \vdash (u, v)^f_{x:A.T} : \left(\textstyle\sum^f_{x:A} T\right)(f[\, x \setminus u \,])}$$

$$\frac{\begin{array}{c}\Gamma \vdash u : \left(\textstyle\sum^f_{x:A} T\right) b \qquad \Gamma, y{:}B, z{:}\left(\textstyle\sum^f_{x:A} T\right) y \vdash P : s \\ \Gamma, i{:}A, j{:}T\, i \vdash v : P\Big[\, y \setminus f\, i\,,\ z \setminus (i,j)^f_{x:A.T} \,\Big]\end{array}}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ z\ \mathrm{in}\ y\ \mathrm{return}\ P\ \mathrm{with} \\ \big|\ (i,j)^f_{x:A.T} \Rightarrow v\end{array} : P[\, y \setminus b\,,\ z \setminus u \,]}$$

$$\begin{array}{l}\mathrm{match}\ (u,v)^f_{x:A.T}\ \mathrm{as}\ z\ \mathrm{in}\ y\ \mathrm{return}\ P\ \mathrm{with} \\ \big|\ (i,j)^f_{x:A.T} \Rightarrow w\end{array} \rightsquigarrow w[\, i \setminus u\,,\ j \setminus v \,]$$

Figure 3 Inductive type families

$$\frac{}{\mathsf{sp}_X\, y} \qquad \frac{}{\mathsf{sp}_X\, 0} \qquad \frac{}{\mathsf{sp}_X\, 1}$$

$$\frac{X \text{ is fresh in } A \quad \mathsf{sp}_X\, B}{\mathsf{sp}_X\, (\textstyle\prod_{x:A} B)} \qquad \frac{\mathsf{sp}_X\, A \quad \mathsf{sp}_X\, B}{\mathsf{sp}_X\, (A + B)} \qquad \frac{X \text{ is fresh in } A \quad \mathsf{sp}_X\, F}{\mathsf{sp}_X\, (\mu Y^A.\, F)}$$

$$\frac{X \text{ is fresh in } A \quad \mathsf{sp}_X\, T}{\mathsf{sp}_X\, (\lambda x^A.\, T)} \qquad \frac{\mathsf{sp}_X\, U \quad X \text{ is fresh in } t}{\mathsf{sp}_X\, (U\, t)} \qquad \frac{\mathsf{sp}_X\, B \quad B \equiv A}{\mathsf{sp}_X\, A}$$

$$\frac{X \text{ is fresh in } f \quad \mathsf{sp}_X\, A \quad \mathsf{sp}_X\, T}{\mathsf{sp}_X\, \left(\textstyle\sum^f_{x:A} T\right)}$$

Figure 4 Strict positivity condition

This type represents the generic form of termination proofs: any terminating recursive fixed point can be made structural over a proof of accessibility. In the algebraic presentation, it is defined as:

$$\begin{cases} \mathrm{Acc} = \lambda A^{\mathrm{Type}}\, R^{A\to A\to \mathrm{Type}}.\, \mu\mathrm{Acc}^{A\to\mathrm{Type}}.\, \lambda x^{A}.\, \prod_{y:A} R\, y\, x \to \mathrm{Acc}\, y \\ \mathrm{acc}_{\mathrm{intro}} = \lambda A^{\mathrm{Type}}\, R^{A\to A\to\mathrm{Type}}\, x^{A}\, f^{\prod_{y:A} R\, y\, x\to \mathrm{Acc}\, A\, R\, y}.\, f \end{cases}$$

Because inductive fixed points are treated transparently, the constructor is rather trivial. However, notice how, in the definition of Acc, the parameter x is treated differently from A and R. The reason is that A and R are *uniform parameters*, in that they do not vary through recursive calls, whereas x does: it is a *non-uniform parameter*. The parameter x is, hence, represented as an index. However, such an index is not sufficient to encode types like EvenOdd.

Representing proper indices requires a new type construction, which we write $\sum^{f}_{x:A} T$. This construction comes from [9, 10], where it is inspired by a categorical point of view: in a sufficiently extensional setting, $\sum^{f}_{x:A} T$ is the right adjoint to a pullback functor. The similarity with the usual notation of dependent sum is not fortuitous, indeed we can define dependent sum as a special case of proper indexing:

$$\begin{cases} \sum_{x:A} B = \left(\sum^{()}_{x:A} B\right)() \\ (u,v)_{x:A.B} = (u,v)^{()}_{x:A.B} \end{cases}$$

We also write $A \times B$ and (u, v) for $\sum_{x:A} B$ and $(u, v)_{x:A.B}$, respectively, when x is not free in B.

In the case of dependent sums, the index is trivial. When it is not, however, the pattern matching return clause P is allowed to depend on the value of the index. This is the purpose of Coq's in-pattern. With the algebraic presentation, the in-pattern has the pleasant property of being confined to the proper indexing construction, hopefully making its meaning more explicit. The syntax differs a little from that of Coq, however: Coq renders the in clause as a pattern with the type name at the head:

```
match n as n' in EvenOdd b return P n' b with
...
end.
```

In the algebraic presentation, types not having a name, the in clause simply consists of a name for the index.

The prototype of proper indexing is the identity type, which we name Eq. In Coq:

```
Inductive Eq (A:Type) (x:A) : A → Type :=
| eq_refl : Eq A x x
```

in the algebraic presentation:

$$\begin{cases} \mathrm{Eq} = \lambda A^{\mathrm{Type}}\, x^{A}.\, \sum^{x}_{\_:1} 1 \\ \mathrm{eq}_{\mathrm{refl}} = \lambda A^{\mathrm{Type}}\, x^{A}.\, ((),())^{x}_{\_:1.1} \end{cases}$$

In fact, dependent sums and identity types are sufficient to define proper indexing. Indeed $\sum^f_{x:A} T$ can be redefined as:

$$\begin{cases} \sum^f_{x:A} T = \lambda y^B.\, \sum_{x:A} (\mathrm{Eq}\, B\, y\, f) \times T \\ (u,v)^f_{x:A.T} = (u, (\mathrm{eq}_{\mathrm{refl}}\, B\, f[\,x \setminus u\,], v))_{x:A.(\mathrm{Eq}\, B\, (f[\,x \setminus u\,])\, f) \times T} \end{cases}$$

It is closer to the spirit of Coq, but in no way essential, to take a proper indexing construction rather than equality as primitive. In Morris & al [9, 10], the dependent sum and equality of the ambient type theory is used to define $\sum^f_{x:A} T$ which is then taken as primitive.

An other choice lies in the use of $A + B$ as primitive. It is the only type construction which allows to define a type with distinct elements. However, a common alternative is to take $\mathbb{B}$ as primitive, in which case we can define $A + B$ as:

$$\begin{cases} A + B = \sum_{b:\mathbb{B}} \left| \begin{array}{l} \text{match } b \text{ with} \\ \text{true} \Rightarrow A \\ \text{false} \Rightarrow B \end{array} \right. \\ \mathrm{inl} = \lambda x^A.\, (\mathrm{true}, x)_{b:\mathbb{B}.\left| \begin{array}{l} \text{match } b \text{ with} \\ \text{true} \Rightarrow A \\ \text{false} \Rightarrow B \end{array} \right.} \\ \mathrm{inr} = \lambda y^B.\, (\mathrm{false}, y)_{b:\mathbb{B}.\left| \begin{array}{l} \text{match } b \text{ with} \\ \text{true} \Rightarrow A \\ \text{false} \Rightarrow B \end{array} \right.} \end{cases}$$

3.3 Examples

The previous section concludes the description of the predicative fragment of the algebraic presentation. We can now give definitions of the remaining inductive families we have encountered in terms of the algebraic presentation. Starting with EvenOdd:

```
Inductive EvenOdd : bool → Type :=
| eo : EvenOdd true
| es : EvenOdd false → EvenOdd true
| os : EvenOdd true → EvenOdd false.
```

translates, in the algebraic presentation, to:

$$\begin{cases} \mathrm{EvenOdd} = \mu\mathrm{EvenOdd}^{\mathbb{B}\to\mathrm{Type}}.\, \left(\sum^{\mathrm{true}}_{\_:1} 1 + \mathrm{EvenOdd}\,\mathrm{false}\right) + \left(\sum^{\mathrm{false}}_{\_:1} \mathrm{EvenOdd}\,\mathrm{true}\right) \\ \quad \mathrm{eo} = \mathrm{inl}\,((), \mathrm{inl}\,())^{\mathrm{true}}_{\_:1.1} \\ \quad \mathrm{es} = \lambda x^{\mathrm{EvenOdd}\,\mathrm{false}}.\, \mathrm{inl}\,((), \mathrm{inr}\,x)^{\mathrm{true}}_{\_:1.1} \\ \quad \mathrm{os} = \lambda x^{\mathrm{EvenOdd}\,\mathrm{true}}.\, \mathrm{inr}\,((), x)^{\mathrm{false}}_{\_:1.1} \end{cases}$$

Here is the definition of List in Coq:

```
Inductive List (A:Type) : Type :=
| nil : List A
| cons : A → List A → List A.
```

and in the algebraic presentation:

$$\begin{cases} \text{List} = \lambda A^{\text{Type}}.\, \mu\text{List}^{\text{Type}}.\, 1 + A \times \text{List} \\ \text{nil} = \lambda A^{\text{Type}}.\, \text{inl}\,() \\ \text{cons} = \lambda A^{\text{Type}}\, x^{A}\, l^{\text{List}\,A}.\, \text{inr}\,(x, l) \end{cases}$$

Finally the definition of Tree:

```
Inductive Tree : Type :=
| node : List Tree → Tree.
```

translates to:

$$\begin{cases} \text{Tree} = \mu\text{Tree}^{\text{Type}}.\, \text{List}\,\text{Tree} \\ \text{node} = \lambda f^{\text{List}\,\text{Tree}}.\, f \end{cases}$$

Note that in the definition of Tree, we must β-reduce List Tree for the recursive definition to be strictly positive.

A more complex example is given by the type of lists indexed by their length, often called *vectors*:

```
Inductive Nat : Type :=
| o : Nat
| s : Nat → Nat.
Inductive Vector (A:Type) : Nat → Type :=
| vnil : Vector A o
| vcons : forall n, A → Vector A n → Vector A (s n).
```

It is encoded in the algebraic presentation as:

$$\begin{cases} \text{Nat} = \mu\text{Nat}^{\text{Type}}.\, 1 + \text{Nat} \\ \quad o = \text{inl}\,() \\ \quad s = \lambda n^{\text{Nat}}.\, \text{inr}\, n \\ \text{Vector} = \lambda A^{\text{Type}}.\, \mu V^{\text{Nat}\to\text{Type}}.\, \lambda n^{\text{Nat}}.\, \left(\sum\nolimits^{o}_{\_:1} 1\right) n + \left(\sum\nolimits^{s\,n'}_{n':\text{Nat}} A \times V\, n'\right) n \\ \quad \text{vnil} = \lambda A^{\text{Type}}.\, \text{inl}\,((),())^{o}_{\_:1.1} \\ \text{vcons} = \lambda A^{\text{Type}}\, n^{\text{Nat}}\, a^{A}\, v^{\text{Vector}\,n\,a}.\, \text{inr}\,(n,(a,v))^{s\,n'}_{n':\text{Nat}.A\times V\,n'} \end{cases}$$

Contrary to proper indices, the types of non-uniform parameters are allowed to be in $\Box$, this allows the definition of types such as the binary lists [12]:

```
Inductive BList (A:Type) : Type :=
| one : A → BList A
| twice : BList (A*A) → BList A
| stwice : A → BList (A*A) → BList A
```

which are rendered in the algebraic presentation as:

$$\begin{cases} \text{BList} = \mu\text{BList}^{\text{Type}\to\text{Type}}.\, \lambda A^{\text{Type}}.\, A + (\text{BList}\,(A\times A) + A \times (\text{BList}\,(A\times A))) \\ \text{one} = \lambda A^{\text{Type}}\, x^{A}.\, \text{inl}\, x \\ \text{twice} = \lambda A^{\text{Type}}\, l^{\text{BList}\,(A\times A)}.\, \text{inr}\,(\text{inl}\, l) \\ \text{stwice} = \lambda A^{\text{Type}}\, x^{A}\, l^{\text{BList}\,(A\times A)}.\, \text{inr}\,(\text{inr}\,(a, l)) \end{cases}$$

In Coq, where there is a hierarchy of universe, types of proper indices can be in any sort. However, a proper index whose type is in Type_i constrains the final type to be in Type_{i+1} or higher. Uniform parameters, of any type, do not constrain the type they parametrise.

Inductive types are consumed by recursive fixed points. Using the implicit unfolding of inductive fixed points, we can pattern-match over the top constructor directly. The Coq function

```
Fixpoint add (x y:Nat) : Nat :=
match y with
| o ⇒ x
| s y' ⇒ s (add x y')
end
```

is rendered in the algebraic presentation as

$$\text{add} = \text{fix add}\ x{:}\text{Nat}\ y{:}\text{Nat} \Rightarrow \left|\begin{array}{l} \text{match}\ y\ \text{as}\ \_\ \text{return Nat with} \\ \text{inl}\,\_ \ \Rightarrow x \\ \text{inr}\, y' \Rightarrow s\,(\text{add}\, x\, y') \end{array}\right.$$

3.4 Co-induction

In addition to inductive fixed points, Coq also has support for co-inductive fixed points. Co-inductive fixed points are required to be strictly positive, like inductive fixed points. We choose in this section, a presentation of co-inductive data where fixed points are explicitly introduced with a constructor. Below we will use this explicit presentation to give a variation

on Coq's co-inductive fixed points.

$$\frac{\Gamma \vdash A : s \quad \Gamma, X{:}A \to \mathrm{Type} \vdash F : A \to \mathrm{Type} \quad \mathsf{sp}_X\, F}{\Gamma \vdash \nu X^{A \to \mathrm{Type}}.\, F : A \to \mathrm{Type}}$$

$$\frac{\Gamma \vdash \nu X^{A \to s}.\, F : A \to s \quad \Gamma \vdash i : A \quad \Gamma \vdash u : F\big[\, X \setminus \nu X^{A \to s}.\, F \,\big]\, i}{\Gamma \vdash \mathrm{forced}\, u : \big(\nu X^{A \to s}.\, F\big)\, i}$$

$$\frac{\begin{array}{ccc} \Gamma \vdash \nu X^{A \to s}.\, F : A \to s & \Gamma \vdash i : A & \Gamma \vdash u : \big(\nu X^{A \to s}.\, F\big)\, i \\ \Gamma, x{:}\big(\nu X^{A \to s}.\, F\big)\, i \vdash P : s' & \Gamma, y{:}F\big[\, X \setminus \nu X^{A \to s}.\, F \,\big]\, i \vdash v : P[\, x \setminus \mathrm{forced}\, y \,] & \end{array}}{\Gamma \vdash \begin{array}{l} \mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{forced}\, y \Rightarrow v \end{array} : P[\, x \setminus u \,]}$$

$$\begin{array}{l} \mathrm{match}\ \mathrm{forced}\, u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{forced}\, y \Rightarrow v \end{array} \rightsquigarrow v[\, y \setminus u \,]$$

Just like inductive data is *destructed* by a recursive fixed point operation, co-inductive data is *constructed* by a co-recursive fixed point operation, allowing co-inductive data to be infinite. The guard condition on co-recursive fixed points ensures that a finite number of unfolding will eventually produce a forced value.

$$\frac{\begin{array}{cc} \Gamma \vdash \prod_{x_1:A_1, \ldots, x_n:A_n} \big(\nu X^{A \to s}.\, F\big)\, i : s & \mathrm{coguarded}\, f\, x_1 \ldots\, x_n\, u \\ \Gamma, f{:}\prod_{x_1:A_1, \ldots, x_n:A_n} \big(\nu X^{A \to s}.\, F\big)\, i, x_1{:}A_1, \ldots, x_n{:}A_n \vdash u : \big(\nu X^{A \to s}.\, F\big)\, i & \end{array}}{\Gamma \vdash \mathrm{cofix}\ f\ x_1{:}A_1\ \ldots\ x_n{:}A_n \Rightarrow u : \prod_{x_1:A_1, \ldots, x_n:A_n} \big(\nu X^{A \to s}.\, F\big)\, i}$$

Co-recursive fixed-point are meant to represent infinite data: they cannot be unfolded eagerly, lest they would fail to terminate. They are unfolded only when they appear at the head of a pattern-matching expression:

$$\rightsquigarrow \begin{array}{l} \mathrm{match}\ (\mathrm{cofix}\ f\ x_1{:}A_1\ \ldots\ x_n{:}A_n \Rightarrow u)\, v_1 \ldots\, v_n\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{forced}\, y \Rightarrow v \\ \mathrm{match}\ u[\, f \setminus (\mathrm{cofix}\ f\ x_1{:}A_1\ \ldots\ x_n{:}A_n \Rightarrow u)\ ,\ x_i \setminus v_i \,]\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{forced}\, y \Rightarrow v \end{array}$$

The dependent elimination rule for co-inductive fixed points asserts, in essence, that every co-inductive data is of the form forced u. Even though it would be fine for inductive fixed points – this is why we could leave the unrolling to the conversion – this does not reflect well the computational aspects of co-inductive data: suspended co-recursive fixed points are values, and won't be evaluated until the context demands it. The fact that the elimination for co-inductive data claims that all values are forced gives rise to undesirable behaviour.

Take for instance the following simple co-inductive type, and data:

$$\begin{cases} T = \nu X.\, X \\ i = \mathrm{cofix}\ i \Rightarrow \mathrm{forced}\, i \end{cases}$$

So that i is effectively an infinite sequence of forced. Using the elimination principle above, it is possible to give a *closed* proof that $\mathrm{Eq}\,T\,i\,(\mathrm{forced}\,i)$:

$$\begin{array}{l}\text{match } i \text{ as } x \text{ return } \mathrm{Eq}\,T\,x\left(\text{forced}\left(\begin{array}{l}\text{match } x \text{ with}\\ \mid\ \text{forced}\,y \Rightarrow y\end{array}\right)\right) \text{ with}\\ \mid\ \text{forced}\,y \Rightarrow \mathrm{eq}_{\mathrm{refl}}\,T\,y\end{array}$$

However, i and $(\mathrm{forced}\,i)$ are not convertible, yet, as every closed proof of equality does, this proof reduces to $\mathrm{eq}_{\mathrm{refl}}$, hence should relate convertible terms. The dependent elimination rule of co-inductive fixed points compromises the type safety of the logic.

Coq uses the above dependent elimination rule for co-inductive fixed points. It was a deliberate decision made for practical purposes. Nonetheless, one may want to weaken it to avoid the incompatibility between equality and conversion. To do so, it suffices to erase the dependency of the return predicate over the matched term:

$$\frac{\Gamma\vdash \nu X^{A\to s}.\,F : A\to s \quad \Gamma\vdash i : A \quad \Gamma\vdash u : \left(\nu X^{A\to s}.\,F\right) i \quad \Gamma\vdash P : s' \quad \Gamma\vdash v : P}{\Gamma\vdash \begin{array}{l}\text{match } u \text{ return } P \text{ with}\\ \mid\ \text{forced}\,y \Rightarrow v\end{array} : P}$$

4 Prop

With all the common baggage for predicative sorts set in place, we can add impredicative sorts to the algebraic presentation. The main such sort in Coq is the sort **Prop** of propositions. The design of **Prop** is guided by *proof irrelevance*: even if it is not actually provable in Coq, different proofs of a proposition are thought of as being equal. This property is useful for program extraction: only the *computationally relevant* parts of a program need to be executed to get the final result. In other words: propositions are considered as *static* data. It is why, with disjunction and existential defined as:

```
Inductive Or (A B:Prop) : Prop :=
| or_introl : A → A ∨ B
| or_intror : B → A ∨ B.
Inductive Ex (A:Type) (P:A→Prop) : Prop :=
| ex_intro : forall x:A, P x → Ex A P.
```

the following terms are refused by type-checking:

```
match x with
| or_introl _ ⇒ true
| or_intror _ ⇒ false
end.
```

and

```
match x with
| ex_intro x _ ⇒ x
end.
```

On the other hand, it is not the case of every inductive type defined in **Prop**, that they cannot be eliminated into **Type**. Conjunction and falsity are two counter-examples:

```
Inductive False : Prop := .
Inductive And (A B:Prop) : Prop :=
| conj : A → B → A ∧ B.
```

Coq allows elimination over these two propositions into **Type**, and both following terms are well-typed:

```
match x return Bool with end.
```

and

```
match x with
| conj _ _ ⇒ true
end.
```

The object of this section is to make syntactically explicit what happens when an inductive type of Coq is declared to be of sort **Prop**. The description elaborated in this section has strong similarities with the system of bracket-types proposed by Awodey & Bauer [2]. They describe the propositions as the subset of types with at most one element, and introduce a left adjoint, written as brackets, to the inclusion of propositions into types. We will reuse their notation, even though, in our intensional setting, T:**Prop** does not enforce that T has a most one element, and the bracketing operation does not properly form an adjunction with the inclusion from **Prop** to **Type**.

4.1 Impredicativity

Let us start by introducing the new sort Prop in the algebraic presentation:

$$\frac{}{\Gamma \vdash \mathrm{Prop} : \mathrm{Type}}$$

As in [2], propositions form a subset of types. Coq has a subtyping rule (also known as *cumulativity*) to make the inclusion transparent. We will, however, render it with a syntactic construct:

$$\frac{\Gamma \vdash A : \mathrm{Prop}}{\Gamma \vdash \{A\} : \mathrm{Type}} \qquad \frac{\Gamma \vdash u : A}{\Gamma \vdash \mathrm{prf}\, u : \{A\}}$$

$$\frac{\Gamma \vdash u : \{A\} \quad \Gamma, x{:}\{A\} \vdash P : s \quad \Gamma, y{:}A \vdash v : P[\, x \setminus \mathrm{prf}\, y\,]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{prf}\, y \Rightarrow v\end{array} : P[\, x \setminus u\,]}$$

$$\begin{array}{l}\mathrm{match}\ \mathrm{prf}\, u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{prf}\, y \Rightarrow v\end{array} \rightsquigarrow v[\, y \setminus u\,]$$

$$\frac{}{\Gamma \vdash 0 : \mathrm{Prop}} \qquad \frac{}{\Gamma \vdash 1 : \mathrm{Prop}}$$

$$\frac{\Gamma \vdash A : \mathrm{Prop} \quad \Gamma \vdash B : \mathrm{Prop} \quad \Gamma, x{:}A \vdash T : \mathrm{Prop} \quad \Gamma, x{:}A \vdash f : B}{\Gamma \vdash \sum^{f}_{x:A} T : B \to \mathrm{Prop}}$$

Figure 5 Singleton rules

This definition simply makes $\{A\}$ a synonym of A, except of sort Type. It is strictly positive in A:

$$\frac{\mathsf{sp}_X\, A}{\mathsf{sp}_X\, \{A\}}$$

The fact that Prop is impredicative – *i.e.* supports the following product formation rules:

$$\frac{\Gamma \vdash A : s \quad \Gamma, x{:}A \vdash B : \mathrm{Prop}}{\Gamma \vdash \prod_{x:A} B : \mathrm{Prop}}$$

is easily understood in terms of proof irrelevance. Indeed, if for all x, B has at most one element, so has the product over x. Even though it uses functional extensionality, which is not provable.

4.2 Singleton rules

The types which (ideally) preserve the proof irrelevance property are sometimes called singleton types in the setting of Coq. In our algebraic presentation, they correspond to inductive type family constructors with extra formation rules to make them preserve propositions. The rules are shown in Figure 5. This elucidates why Coq allows elimination over False and And into arbitrary type: False is implemented as 0 and And A B and $A \times B$. The elimination rules being unchanged, pattern-matching over proofs of False and And are unrestricted. Because restricted pattern-matching is often seen as the default, singleton types are said to enjoy *singleton elimination*.

Remark that, proofs of propositions being uninformative, there is essentially nothing to be gained from depending on, or being indexed over a proposition. In consequence, the type formation rule for proper indexing in Figure 5 is only useful, in practice, for the subcase of cartesian product.

Coq actually implements two other singleton rules. The first one is for inductive fixed points. In our algebraic presentation:

$$\frac{\Gamma \vdash A : s \quad \Gamma, X{:}A \to \mathrm{Prop} \vdash F : A \to \mathrm{Prop} \quad \mathsf{sp}_X\, F}{\Gamma \vdash \mu X^{A \to \mathrm{Prop}}.\, F : A \to \mathrm{Prop}}$$

It allows to type the accessibility predicate Acc in Prop. This rule is sound in that fixed points indeed preserve proof irrelevance in presence of functional extensionality. It is also very useful for extraction: structural recursion over Acc allows the definition of functions

whose termination cannot be proved automatically by the guard condition. However, the proof is no longer needed to ensure termination in the target languages of extraction. In this sense, at least, it is static data.

The last singleton rule allows properly indexed families in Prop (not how it is stronger than the rule dependent sum of Figure 5):

$$\frac{\Gamma \vdash A : \mathrm{Prop} \quad \Gamma \vdash B : \mathrm{Type} \quad \Gamma, x{:}A \vdash T : \mathrm{Prop} \quad \Gamma, x{:}A \vdash f : B}{\Gamma \vdash \sum^{f}_{x:A} T : B \to \mathrm{Prop}}$$

It turns the identity type Eq into a proposition. It is known to be sound to accept that Eq is proof irrelevant [6]. It is also useful for extraction, as equal types, in a closed environment, are extracted to the same type. Hence a program may safely eliminate over Eq knowing that it will not affect the performances of the extracted code. In Coq, the index B in the rule above can be of any sort Type_i, however, this wisdom has been challenged in recent years with the formulation of the *univalence principle* [15], of which a simple consequence is that Eq is *not* proof irrelevant at every type. Indeed, some extracted Coq programs written assuming the univalence principle crash.

To correct for the univalence principle, the singleton rule for proper indices can be simply dropped; but it can also be restricted to the lowest sort: $B : \mathrm{Type}_0$. More precisely the conjunction of the univalence principle and the proof irrelevance principle is consistent as long as the singleton rule of proper indices is restricted to sorts s such that there is no sort s' other than Prop such that $s' : s$. Because, if such a sort s' exists, $\mathbb{B} : s'$ and by univalence, $\mathrm{Eq}\,\mathbb{B}$ has two distinct elements contradicting proof irrelevance.

For types which do not enjoy singleton elimination, turning them into propositions means restricting their elimination. We achieve this effect by adding a single type construction coercing from Type to Prop:

$$\frac{\Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash [A] : \mathrm{Prop}} \qquad \frac{\Gamma \vdash u : A}{\Gamma \vdash \langle u \rangle : [A]}$$

$$\frac{\Gamma \vdash u : [A] \qquad \Gamma, x{:}\,[A] \vdash P : \mathrm{Prop} \qquad \Gamma, y{:}A \vdash v : P[\, x \setminus \langle y \rangle \,]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \langle y \rangle \Rightarrow v\end{array} : P[\, x \setminus u \,]}$$

$$\begin{array}{l}\mathrm{match}\ \langle u \rangle\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \langle y \rangle \Rightarrow v\end{array} \rightsquigarrow v[\, y \setminus u \,]$$

The important rule is the elimination rule, where the return clause is limited to be of sort Prop, whereas every other type construction can be eliminated to any sort. Apart from this restriction $[A]$ is a synonym of A, except in Prop. In [2], the type theory is extensional, in that the identity type and the conversion relation coincide. The elimination rules for bracket is much finer and reflects precisely the fact that propositions are proof-irrelevant. In an intensional type theory, restricting with respect to sorts approximates this behaviour: even if we constrained propositions to be proof-irrelevant, not every proof irrelevant type will have

type Prop. The bracketing construction is also strictly positive:

$$\frac{\mathsf{sp}_X\, A}{\mathsf{sp}_X\, [A]}$$

It is actually possible, using only the impredicative dependent product to define a bracketing operation: $\prod_{P:\mathrm{Prop}} (A \to P) \to P$. Like $[A]$ it behaves as A except it can only be used to form a proposition. However, the impredicative encoding is positive but not strictly, which motivates the introduction of the extra construction.

4.3 Examples

The logical connectives can be defined as follows:

$$\begin{cases} \mathrm{False} = 0 \\ \mathrm{And} = \lambda A^{\mathrm{Prop}}\, B^{\mathrm{Prop}}.\, A \times B \\ \mathrm{pair} = \lambda A^{\mathrm{Prop}}\, B^{\mathrm{Prop}}\, x^A\, y^B.\, (x, y) \\ \mathrm{Or} = \lambda A^{\mathrm{Prop}}\, B^{\mathrm{Prop}}.\, [A + B] \\ \mathrm{or}_{\mathrm{introl}} = \lambda A^{\mathrm{Prop}}\, B^{\mathrm{Prop}}\, x^A.\, \langle \mathrm{inl}\, x \rangle \\ \mathrm{or}_{\mathrm{intror}} = \lambda A^{\mathrm{Prop}}\, B^{\mathrm{Prop}}\, y^B.\, \langle \mathrm{inr}\, y \rangle \\ \mathrm{Ex} = \lambda A^{\mathrm{Type}}\, P^{A \to \mathrm{Prop}}.\, \left[\sum_{x:A} P\, x \right] \\ \mathrm{ex}_{\mathrm{intro}} = \lambda A^{\mathrm{Type}}\, P^{A \to \mathrm{Prop}}\, x^A\, p^{P\, x}.\, \langle (x, p)_{x:A.P} \rangle \end{cases}$$

Note how, because of the brackets, existentials and disjunctions are prohibited from being eliminated to non-propositional types. Thanks to the singleton rules, however, conjunction and falsity do not require brackets.

As a final example, consider the type Ascending n p of ascending sequences of integers between p and n defined by mutual recursion with the proposition Ge m p which stands from m is greater than or equal to p:

```
Inductive Ascending : Nat → Nat → Type :=
| top : forall n, Ascending n n
| up : forall n p m, Ge m (s p) → Ascending n m → Ascending n p
with Ge : Nat → Nat → Prop :=
| ascend : forall m p, Ascending m p → Ge m p.
```

As Ascending has type **Type**, whereas Ge has type **Prop**, the translation to a single inductive type is not as straightforward as Even and Odd. The translation requires the use of brackets around the recursive calls:

$$\begin{cases} \mathrm{AscendingGe} = \mu X^{(\mathrm{Nat}\times\mathrm{Nat})+(\mathrm{Nat}\times\mathrm{Nat})\to\mathrm{Type}}.\, \lambda i^{(\mathrm{Nat}\times\mathrm{Nat})+(\mathrm{Nat}\times\mathrm{Nat})}. \\ \qquad \left(\sum_{n:\mathrm{Nat}}^{\mathrm{inl}\,(n,n)} 1\right) i \\ \qquad + \left(\sum_{j:\mathrm{Nat}\times\mathrm{Nat}}^{\mathrm{inl}\, j} \sum_{m:\mathrm{Nat}} [X\,(\mathrm{inr}\,(m, s\,(\pi_2\, j)))] \times X\,(\mathrm{inl}\,(\pi_1\, j, m))\right) i \\ \qquad + \left(\sum_{j:\mathrm{Nat}\times\mathrm{Nat}}^{\mathrm{inr}\, j} X\,(\mathrm{inl}\, j)\right) i \\ \mathrm{Ascending} = \lambda n\, p.\, \mathrm{AscendingGe}\,(\mathrm{inl}\,(n, p)) \\ \mathrm{Ge} = \lambda m\, p.\, [\mathrm{AscendingGe}\,(\mathrm{inr}\,(m, p))] \end{cases}$$

5 Impredicative Set

In addition to the impredicative sort Prop, Coq has a sort Set which is predicative by default but can be turned impredicative with a flag. Where Prop is meant to be used in the context of separating static and dynamic information, the spirit of the impredicative sort Set is to be as powerful as possible without being inconsistent. In the algebraic presentation, that means being stable by every construction except dependent sums with the first projection in an arbitrary sort (*strong sums*).

To mirror the optional nature of the impredicativity of Set, the rules for a predicative sort Set are given in Figure 6; to turn impredicativity on, the rules of Figure 7 must be used *in addition* to those of predicative Set. This presentation makes immediately apparent that impredicative Set is an extension of predicative Set, in that every program of the latter typechecks in the former.

The rules of Set are the same as those of Prop, with the exception of $A + B$ which is in Set when both A and B are – even with predicative Set. Hence, there are types in Set with several elements – *e.g.* $\mathbb{B}$. As a consequence, the bracketing operation which coerces types in Type to Set does not enjoy an explanation in terms of proof irrelevance, as was the case in Prop. As a matter of fact, there is no clear set-theoretical description at all. A close cousin of Set bracketing, however, can be found in homotopy type theory [15], where, roughly, groupoids are *truncated* to sets through a quotient of their homsets by the total relation.

6 Conclusion

The algebraic presentation of Coq makes the conversion between sorts explicit. The toplevel inductive definitions of Coq can be understood as implicitly inserting canonical bracketing operations when an inductive type is declared inside an impredicative sort but should be of a different sort due to its form; and inserting type coercion from a smaller sort to a bigger sort when applying a cumulativity rule.

Monolithic type definitions like in Coq have a number of advantages over the algebraic presentation, they boil down to better type errors due to naming, better type inference and better memory representation due to n-ary sums and products. However, the value of the implicit coercions between sorts is less clear. In particular, the bracketing operation to impredicative sorts is probably a better guide for program extraction than the current method of figuring whether or not a given type is a proposition, which interacts badly with universe polymorphism [7]. Explicit coercions for extraction are also in the spirit of [5].

All of the algebraic type constructors can actually be defined in Coq, except the two fixed-points because there is no way to abstract over strictly positive type families. So is it clear that expressions of the algebraic presentation which do not use inductive or co-inductive fixed points can be translated into Coq. Occurrences of the fixed points in a type must be λ-lifted and given a toplevel name. Some care must be given to avoiding the duplication of such definitions otherwise types which must be convertible for the expression to typecheck, might be seen as different in the Coq translation. Apart from this technicality, translation from the algebraic presentation to Coq is straightforward. We claim that, at least if we extend the algebraic presentation to a hierarchy of universes and the strict positivity condition is made a bit more fine-grained, Coq terms can be, conversely, translated into the algebraic presentation.

$$\frac{}{\Gamma \vdash \mathrm{Set} : \mathrm{Type}} \qquad \frac{\Gamma \vdash A : \mathrm{Set} \quad \Gamma, x{:}A \vdash B : \mathrm{Set}}{\Gamma \vdash \prod_{x:A} B : \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : \mathrm{Set} \quad \Gamma, x{:}A \vdash P : \mathrm{Set}}{\Gamma \vdash \sum_{x:A} P : \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : \mathrm{Set} \quad \Gamma \vdash B : \mathrm{Set}}{\Gamma \vdash A + B : \mathrm{Set}} \qquad \frac{}{\Gamma \vdash 1 : \mathrm{Set}} \qquad \frac{}{\Gamma \vdash 0 : \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma, X{:}A \to \mathrm{Set} \vdash F : A \to \mathrm{Set} \quad \mathsf{sp}_X\, F}{\Gamma \vdash \mu X^{A \to \mathrm{Set}}.\, F : A \to \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : \mathrm{Set} \quad \Gamma \vdash B : \mathrm{Set} \quad \Gamma, x{:}A \vdash T : \mathrm{Set} \quad \Gamma, x{:}A \vdash f : B}{\Gamma \vdash \sum^{f}_{x:A} T : B \to \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma, X{:}A \to \mathrm{Set} \vdash F : A \to \mathrm{Set} \quad \mathsf{sp}_X\, F}{\Gamma \vdash \nu X^{A \to \mathrm{Set}}.\, F : A \to \mathrm{Set}}$$

$$\frac{\Gamma \vdash A : \mathrm{Set}}{\Gamma \vdash \{A\}_{\mathrm{Set}} : \mathrm{Type}} \qquad \frac{\Gamma \vdash u : A}{\Gamma \vdash \mathrm{elt}\, u : \{A\}_{\mathrm{Set}}}$$

$$\frac{\Gamma \vdash u : \{A\}_{\mathrm{Set}} \quad \Gamma, x{:}\{A\}_{\mathrm{Set}} \vdash P : s \quad \Gamma, y{:}A \vdash v : P[\,x \setminus \mathrm{elt}\, y\,]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{elt}\, y \Rightarrow v\end{array} : P[\,x \setminus u\,]}$$

$$\begin{array}{l}\mathrm{match}\ \mathrm{elt}\, u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \mathrm{elt}\, y \Rightarrow v\end{array} \rightsquigarrow v[\,y \setminus u\,]$$

$$\frac{\mathsf{sp}_X\, A}{\mathsf{sp}_X\, \{A\}_{\mathrm{Set}}}$$

Figure 6 Rules for predicative Set

$$\frac{\Gamma \vdash A : s \quad \Gamma, x{:}A \vdash B : \mathrm{Set}}{\Gamma \vdash \prod_{x:A} B : \mathrm{Set}} \qquad \frac{\mathsf{sp}_X\, A}{\mathsf{sp}_X\, [A]_{\mathrm{Set}}}$$

$$\frac{\Gamma \vdash A : \mathrm{Type}}{\Gamma \vdash [A]_{\mathrm{Set}} : \mathrm{Set}} \qquad \frac{\Gamma \vdash u : A}{\Gamma \vdash \langle u \rangle_{\mathrm{Set}} : [A]_{\mathrm{Set}}}$$

$$\frac{\Gamma \vdash u : [A]_{\mathrm{Set}} \quad \Gamma, x{:}[A]_{\mathrm{Set}} \vdash P : \mathrm{Set} \quad \Gamma, y{:}A \vdash v : P[\, x \setminus \langle y \rangle_{\mathrm{Set}}\,]}{\Gamma \vdash \begin{array}{l}\mathrm{match}\ u\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \langle y \rangle_{\mathrm{Set}} \Rightarrow v\end{array} : P[\, x \setminus u\,]}$$

$$\begin{array}{l}\mathrm{match}\ \langle u \rangle_{\mathrm{Set}}\ \mathrm{as}\ x\ \mathrm{return}\ P\ \mathrm{with} \\ \mid\ \langle y \rangle_{\mathrm{Set}} \Rightarrow v\end{array} \rightsquigarrow v[\, y \setminus u\,]$$

Figure 7 Rules for impredicative Set

References

1. Andreas Abel. *A polymorphic lambda-calculus with sized higher-order types*. PhD thesis, 2006.
2. Steve Awodey and Andrej Bauer. Propositions as [Types]. *Journal of Logic and Computation*, 14(4):447–471, August 2004.
3. Henk Barendregt. Lambda calculus with types. *Handbook of logic in computer science*, 1992.
4. Bruno Barras. Semantical Investigations in Intuitionistic Set Theory and Type Theories with Inductive Families. *Thèse d'Habilitation*, 2013.
5. Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. *Foundations of Software Science and Computational Structures*, 4962:365–379, 2008.
6. Gyesik Lee and Benjamin Werner. Proof-irrelevant model of CC with predicative induction and judgmental equality. 2011.
7. Pierre Letouzey and Bas Spitters. Implicit and noncomputational arguments using monads. pages 1–16, 2005.
8. Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, 2007.
9. Peter Morris and Thorsten Altenkirch. Constructing strictly positive families. *CATS '07 Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 111–121, 2007.
10. Peter Morris, Thorsten Altenkirch, and Neil Ghani. A universe of strictly positive families. *IInternational journal of foundations of computer science*, pages 83–107, 2009.
11. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
12. Chris Okasaki. *Purely functional data structures*. 1999.
13. Christine Paulin-Mohring. *Définitions inductives en théorie des types d'ordre supérieur*. PhD thesis, Université Claude Bernard-Lyon I, 1996.
14. The Coq development team. The Coq Proof Assistant.
15. The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*.