Equations `1.0beta2` Reference Manual

Matthieu Sozeau

December 17, 2017

Introduction

EQUATIONS is a toolbox built as a plugin on top of the COQ proof assistant to program and reason on programs defined by full *dependent* pattern-matching and *well-founded* recursion. While the primitive core calculus of COQ allows definitions by *simple* pattern-matching on inductive families and *structural* recursion, EQUATIONS extends the set of easily definable constants by allowing a richer form of pattern-matching and arbitrarily complex recursion schemes. It can be thought of as a twin of the FUNCTION package for Isabelle that implements a *definitional* translation from partial, well-founded recursive functions to the HOL core logic. See Ana Bove and Alexander Krauss and Matthieu Sozeau (2015) for an overview of tools for defining recursive definitions in interactive proof assistants like (COQ, AGDA or ISABELLE).

The first version of the tool was described in Sozeau (2010), the most recent one is described in Mangin and Sozeau (2017). This manual provides an overview of the system by way of introductory examples (chapter 1), and a short documentation of the plugin commands (chapter 2).

This manual describes version `1.0beta2` of the package.

Installation

Equations is available through the `opam`[1] package manager as package `coq-equations`. To install it on an already existing `opam` installation with the COQ repository, simply input the command:

```
# opam install coq-equations
```

The development version and detailed installation instructions are available at http://mattam82.github.io/Coq-Equations.

[1] http://opam.ocaml.org

Contents

Chapter 1

A gentle introduction to Equations

The source of this chapter that can be run in Coq with Equations installed is available at:

`https://raw.githubusercontent.com/mattam82/Coq-Equations/master/doc/equations_intro.v`

Equations is a plugin for Coq that comes with a few support modules defining classes and tactics for running it. We will introduce its main features through a handful of examples. We start our Coq primer session by importing the Equations module.

```
Require Import Arith Omega.
From Equations Require Import Equations.
```

1.1 Inductive types

In its simplest form, Equations allows to define functions on inductive datatypes. Take for example the booleans defined as an inductive type with two constructors true and false:

```
Inductive bool : Set := true : bool | false : bool
```

We can define the boolean negation as follows:

```
Equations neg (b : bool) : bool :=
neg true := false ;
neg false := true.
Print All.
```

Equations declarations are formed by a signature definition and a set of *clauses* that must form a *covering* of this signature. The compiler is then expected to automatically find a corresponding case-splitting tree that implements the function. In this case, it simply needs to split on the single variable *b* to produce two new *programming problems* neg true and neg false that are directly handled by the user clauses. We will see in more complex examples that this search for a splitting tree may be non-trivial.

1.2 Reasoning principles

In the setting of a proof assistant like Coq, we need not only the ability to define complex functions but also get good reasoning support for them. Practically, this translates to the ability to simplify applications of functions appearing in the goal and to give strong enough proof principles for (recursive) definitions.

`Equations` provides this through an automatic generation of proofs related to the function. Namely, each defining equation gives rise to a lemma stating the equality between the left and right hand sides. These equations can be used as rewrite rules for simplification during proofs, without having to rely on the fragile simplifications implemented by raw reduction. We can also generate the inductive graph of any `Equations` definition, giving the strongest elimination principle on the function.

I.e., for neg the inductive graph is defined as:

```
Inductive neg_ind : bool → bool → Prop :=
| neg_ind_equation_1 : neg_ind true false
| neg_ind_equation_2 : neg_ind false true
```

Along with a proof of $\Pi\ b$, neg_ind b (neg b), we can eliminate any call to neg specializing its argument and result in a single command. Suppose we want to show that neg is involutive for example, our goal will look like:

```
b : bool
============================
neg (neg b) = b
```

An application of the tactic *funelim* (neg b) will produce two goals corresponding to the splitting done in neg: neg false = true and neg true = false. These correspond exactly to the rewriting lemmas generated for neg.

In the following sections we will show how these ideas generalize to more complex types and definitions involving dependencies, overlapping clauses and recursion.

1.3 Building up

1.3.1 Polymorphism

Coq's inductive types can be parameterized by types, giving polymorphic datatypes. For example the list datatype is defined as:

```
Inductive list {A} : Type := nil : list | cons : A → list → list.

Arguments list : clear implicits.
Notation "x :: l" := (cons x l).
```

No special support for polymorphism is needed, as type arguments are treated like regular arguments in dependent type theories. Note however that one cannot match on type arguments, there is no intensional type analysis. We can write the polymorphic tail function as follows:

```
Equations tail {A} (l : list A) : list A :=
tail nil := nil ;
tail (cons a v) := v.
```

Note that the argument $\{A\}$ is declared implicit and must hence be omitted in the defining clauses. In each of the branches it is named A. To specify it explicitely one can use the syntax $\{A{:=}B\}$, renaming that implicit argument to B in this particular case

1.3.2 Recursive inductive types

Of course with inductive types comes recursion. Coq accepts a subset of the structurally recursive definitions by default (it is incomplete due to its syntactic nature). We will use this as a first step towards a more robust treatment of recursion via well-founded relations. A classical example is list concatenation:

```
Equations app {A} (l l' : list A) : list A :=
app nil l' := l' ;
app (cons a l) l' := cons a (app l l').
```

Recursive definitions like app can be unfolded easily so proving the equations as rewrite rules is direct. The induction principle associated to this definition is more interesting however. We can derive from it the following *elimination* principle for calls to app:

```
app_elim :
∀ P : ∀ (A : Type) (l l' : list A), list A → Prop,
(∀ (A : Type) (l' : list A), P A nil l' l') →
(∀ (A : Type) (a : A) (l l' : list A),
P A l l' (app l l') → P A (a :: l) l' (a :: app l l')) →
∀ (A : Type) (l l' : list A), P A l l' (app l l')
```

Using this eliminator, we can write proofs exactly following the structure of the function definition, instead of redoing the splitting by hand. This idea is already present in the Function package Barthe et al. (2006) that derives induction principles from function definitions.

1.3.3 Moving to the left

The structure of real programs is richer than a simple case tree on the original arguments in general. In the course of a computation, we might want to scrutinize intermediate results (e.g. coming from function calls) to produce an answer. This literally means adding a new pattern to the left of our equations made available for further refinement. This concept is know as with clauses in the Agda Norell (2007) community and was first presented and implemented in the Epigram language McBride and McKinna (2004).

The compilation of with clauses and its treatment for generating equations and the induction principle are quite involved in the presence of dependencies, but the basic idea is to add a new case analysis to the program. To compute the type of the new subprogram, we actually abstract the discriminee term from the expected type of the clause, so that the type can get refined in the subprogram. In the non-dependent case this does not change anything though.

Each `with` node generates an auxiliary definition from the clauses in the curly brackets, taking the additional object as argument. The equation for the with node will simply be an indirection to the auxiliary definition and simplification will continue as usual with the auxiliary definition's rewrite rules.

```
Equations filter {A} (l : list A) (p : A → bool) : list A :=
filter nil p := nil ;
filter (cons a l) p ⇐ p a ⇒ {
  filter (cons a l) p true := a :: filter l p ;
  filter (cons a l) p false := filter l p }.
```

By default, equations makes definitions opaque after definition, to avoid spurious unfoldings, but this can be reverted on a case by case basis, or using the global `Set Equations Transparent` option. `Global Transparent` filter.

A common use of with clauses is to scrutinize recursive results like the following:

```
Equations unzip {A B} (l : list (A × B)) : list A × list B :=
unzip nil := (nil, nil) ;
unzip (cons p l) ⇐ unzip l ⇒ {
  unzip (cons (pair a b) l) (pair la lb) := (a :: la, b :: lb) }.
```

The real power of with however comes when it is used with dependent types.

1.4 Dependent types

Coq supports writing dependent functions, in other words, it gives the ability to make the results *type* depend on actual *values*, like the arguments of the function. A simple example is given below of a function which decides the equality of two natural numbers, returning a sum type carrying proofs of the equality or disequality of the arguments. The sum type $\{\ A\ \} + \{\ B\ \}$ is a constructive variant of disjunction that can be used in programs to give at the same time a boolean algorithmic information (are we in branch A or B) and a *logical* information (a proof witness of A or B). Hence its constructors `left` and `right` take proofs as arguments. The eq_refl proof term is the single proof of $x = x$ (the x is generaly infered automatically).

```
Equations equal (n m : nat) : { n = m } + { n ≠ m } :=
equal O O := left eq_refl ;
equal (S n) (S m) with equal n m := {
```

  equal (S n) (S $?(n)$) (left eq_refl) := left eq_refl ;
  equal (S n) (S m) (right p) := right _ } ;
equal x y := right _.

Of particular interest here is the inner program refining the recursive result. As equal n m is of type $\{ n = m \} + \{ n \neq m \}$ we have two cases to consider:

- Either we are in the `left` p case, and we know that p is a proof of $n = m$, in which case we can do a nested match on p. The result of matching this equality proof is to unify n and m, hence the left hand side patterns become S n and S $?(n)$ and the return type of this branch is refined to $\{ n = n \} + \{ n \neq n \}$. We can easily provide a proof for the left case.
- In the right case, we mark the proof unfilled with an underscore. This will generate an obligation for the hole, that can be filled automatically by a predefined tactic or interactively by the user in proof mode (this uses the same obligation mechanism as the Program extension Sozeau (2007)). In this case the automatic tactic is able to derive by itself that $n \neq m \to$ S $n \neq$ S m.

Dependent types are also useful to turn partial functions into total functions by restricting their domain. Typically, we can force the list passed to head to be non-empty using the specification:

```
Equations head {A} (l : list A) (pf : l ≠ nil) : A :=
head nil pf :=! pf;
head (cons a v) _ := a.
```

We decompose the list and are faced with two cases:

- In the first case, the list is empty, hence the proof pf of type nil $\neq$ nil allows us to derive a contradiction. We make use of another category of right-hand sides, which we call *empty* nodes to inform the compiler that a contradiction is derivable in this case. In general we cannot expect the compiler to find by himself that the context contains a contradiction, as it is undecidable (Oury (2007); Goguen et al. (2006)).
- In the second case, we simply return the head of the list, disregarding the proof.

1.4.1 Inductive families

The next step is to make constraints such as non-emptiness part of the datatype itself. This capability is provided through inductive families in Coq Paulin-Mohring (1993), which are a similar concept to the generalization of algebraic datatypes to GADTs in functional languages like Haskell Schrijvers et al. (2009). Families provide a way to associate to each constructor a different type, making it possible to give specific information about a value in its type.

Equality

The alma mater of inductive families is the propositional equality eq defined as:

Inductive eq $(A : \mathtt{Type})$ $(x : A) : A \to \mathtt{Prop} :=$
eq_refl : eq $A\ x\ x$.

Equality is a polymorphic relation on A. (The Prop sort (or kind) categorizes propositions, while the Set sort, equivalent to $\star$ in Haskell categorizes computational types.) Equality is *parameterized* by a value x of type A and *indexed* by another value of type A. Its single constructor states that equality is reflexive, so the only way to build an object of eq $x\ y$ is if $x\ \tilde{}= y$, that is if x is definitionaly equal to y.

Now what is the elimination principle associated to this inductive family? It is the good old Leibniz substitution principle:

$\forall\ (A : \mathtt{Type})\ (x : A)\ (P : A \to \mathtt{Type}),\ P\ x \to \forall\ y : A,\ x = y \to P\ y$

Provided a proof that $x = y$, we can create on object of type $P\ y$ from an existing object of type $P\ x$. This substitution principle is enough to show that equality is symmetric and transitive. For example we can use pattern-matching on equality proofs to show:

Equations eqt $\{A\}$ $(x\ y\ z : A)$ $(p : x = y)$ $(q : y = z) : x = z :=$
eqt x $?(x)$ $?(x)$ eq_refl eq_refl := eq_refl.

Let us explain the meaning of the non-linear patterns here that we slipped through in the equal example. By pattern-matching on the equalities, we have unified x, y and z, hence we determined the *values* of the patterns for the variables to be x. The $?(x)$ notation is essentially denoting that the pattern is not a candidate for refinement, as it is determined by another pattern. This particular patterns are called “inaccessible”.

Indexed datatypes

Functions on vectors provide more stricking examples of this situation. The vector family is indexed by a natural number representing the size of the vector: [Inductive vector $(A : \mathtt{Type}) : \mathsf{nat} \to \mathtt{Type} :=$ | Vnil : vector A O | Vcons : $A \to \forall\ n : \mathsf{nat}$, vector $A\ n \to$ vector A (S n)]

The empty vector Vnil has size O while the cons operation increments the size by one. Now let us define the usual map on vectors: Notation Vnil := Vector.nil.

Notation Vcons := Vector.cons.

Equations vmap $\{A\ B\}$ $(f : A \to B)$ $\{n\}$ $(v :$ vector $A\ n)$:
vector $B\ n :=$
vmap f $\{n{:=}?(0)\}$ Vnil := Vnil ;
vmap f $\{n{:=}?(\mathsf{S}\ n)\}$ (Vcons $a\ n\ v$) := Vcons $(f\ a)$ (vmap $f\ v$).

Here the value of the index representing the size of the vector is directly determined by the constructor, hence in the case tree we have no need to eliminate

n. This means in particular that the function vmap does not do any computation with n, and the argument could be eliminated in the extracted code. In other words, it provides only *logical* information about the shape of v but no computational information.

The vmap function works on every member of the vector family, but some functions may work only for some subfamilies, for example vtail:

```
Equations vtail {A n} (v : vector A (S n)) : vector A n :=
vtail (Vcons a n v') := v'.
```

The type of v ensures that vtail can only be applied to non-empty vectors, moreover the patterns only need to consider constructors that can produce objects in the subfamily vector A (S n), excluding Vnil. The pattern-matching compiler uses unification with the theory of constructors to discover which cases need to be considered and which are impossible. In this case the failed unification of 0 and S n shows that the Vnil case is impossible. This powerful unification engine running under the hood permits to write concise code where all uninteresting cases are handled automatically.

Of course the equations and the induction principle are simplified in a similar way. If we encounter a call to vtail in a proof, we can use the following elimination principle to simplify both the call and the argument which will be automatically substituted by an object of the form Vcons _ _ _:

```
∀ P : ∀ (A : Type) (n : nat), vector A (S n) → vector A n → Prop,
(∀ (A : Type) (n : nat) (a : A) (v : vector A n),
  P A n (Vcons a v) v) →
∀ (A : Type) (n : nat) (v : vector A (S n)), P A n v (vtail v)
```

As a witness of the power of the unification, consider the following function which computes the diagonal of a square matrix of size $n \times n$.

```
Equations diag {A n} (v : vector (vector A n) n) : vector A n :=
diag {n:=O} Vnil := Vnil ;
diag {n:=(S ?(n))} (Vcons (Vcons a n v) ?(n) v') :=
  Vcons a (diag (vmap vtail v')).
```

Here in the second equation, we know that the elements of the vector are necessarily of size S n too, hence we can do a nested refinement on the first one to find the first element of the diagonal.

1.4.2 Recursion

Notice how in the diag example above we explicitely pattern-matched on the index n, even though the Vnil and Vcons pattern matching would have been enough to determine these indices. This is because the following definitions fails:

```
Fail Equations diag' {A n} (v : vector (vector A n) n) : vector A n :=
diag' Vnil := Vnil ;
diag' (Vcons (Vcons a n v) n v') :=
  Vcons a (diag' (vmap vtail v')).
```

Indeed, Coq cannot guess the decreasing argument of this fixpoint using its limited syntactic guard criterion: vmap vtail *v'* cannot be seen to be a (large) subterm of *v'* using this criterion, even if it is clearly "smaller". In general, it can also be the case that the compilation algorithm introduces decorations to the proof term that prevent the syntactic guard check from seeing that the definition is structurally recursive.

To aleviate this problem, Equations provides support for *well-founded* recursive definitions which do not rely on syntactic checks.

The simplest example of this is using the lt order on natural numbers to define a recursive definition of identity:

```
Require Import Equations.Subterm.

Equations id (n : nat) : nat :=
  id n by rec n lt :=
  id 0 := 0;
  id (S n') := id n'.
```

Here id is defined by well-founded recursion on lt on the (only) argument *n* using the by *rec* node. At recursive calls of id, obligations are generated to show that the arguments effectively decrease according to this relation. Here the proof that *n'* ¡ S *n'* is discharged automatically.

Wellfounded recursion on arbitrary dependent families is not as easy to use, as in general the relations on families are *heterogeneous*, as the must related inhabitants of potentially different instances of the family. Equations provides a *Derive* command to generate the subterm relation on any such inductive family and derive the well-foundedness of its transitive closure, which is often what's required. This provides course-of-values or so-called "mathematical" induction on these objects, mimicking the structural recursion criterion in the logic.

```
Derive Signature Subterm for vector.
```

For vectors for example, the relation is defined as:

```
Inductive t_direct_subterm (A : Type) :
  ∀ n n0 : nat, vector A n → vector A n0 → Prop :=
    t_direct_subterm_1_1 : ∀ (h : A) (n : nat) (H : vector A n),
      t_direct_subterm A n (S n) H (Vcons h H)
```

That is, there is only one recursive subterm, for the subvector in the Vcons constructor. We also get a proof of:

```
Check well_founded_t_subterm : ∀ A, WellFounded (t_subterm A).
```

The relation is actually called t_subterm as vector is just a notation for *Vector.t*. t_subterm itself is the transitive closure of the relation seen as an homogeneous one by packing the indices of the family with the object itself. Once this is derived, we can use it to define recursive definitions on vectors that the guard condition couldn't handle. The signature provides a signature_pack function to pack a vector with its index. The well-founded relation is defined on the packed vector type.

```
Module UnzipVect.
  Context {A B : Type}.
```

We can use the packed relation to do well-founded recursion on the vector. Note that we do a recursive call on a substerm of type vector A n which must be shown smaller than a vector A (S n). They are actually compared at the packed type { n : nat & vector A n}.

```
  Equations unzip {n} (v : vector (A × B) n) : vector A n × vector B n
:=
  unzip v by rec (signature_pack v) (@t_subterm (A × B)) :=
  unzip Vnil := (Vnil, Vnil) ;
  unzip (Vector.cons (pair x y) n v) with unzip v := {
    | pair xs ys := (Vector.cons x xs, Vector.cons y ys) }.
End UnzipVect.
```

While this was just mimicking simple structural recursion, we can of course use this for more elaborate termination arguments. We put ourselves in a section to parameterize a *skip* function by a predicate:

```
Section Skip.
  Context {A : Type} (p : A → bool).
  Equations skip_first {n} (v : vector A n) : &{ n : nat & vector A n } :=
  skip_first Vnil := &(0 & Vnil);
  skip_first (Vcons a n v') ⇐ p a ⇒ {
                              | true ⇒ skip_first v';
                              | false ⇒ &(_ & Vcons a v') }.
```

It is relatively straitforward to show that *skip* returns a (large) subvector of its argument

```
  Lemma skip_first_subterm {n} (v : vector A n) : clos_refl _ (t_subterm _)
(skip_first v) &(_ & v).
  Proof.
    funelim (skip_first v).
    constructor 2.
    depelim H.
    constructor 1.
    eapply clos_trans_stepr. simpl.
    apply (t_direct_subterm_1_1 _ _ _ (&(_ & t).2)). apply H.
    rewrite H. constructor. eauto with subterm_relation.
    constructor 2.
  Qed.

End Skip.
```

This function takes an unsorted vector and returns a sorted vector corresponding to it starting from its head a, removing all elements smaller than a and recursing.

```
Equations sort {n} (v : vector nat n) : &{n' : _ & vector nat n'} :=
sort v by rec (signature_pack v) (t_subterm nat) :=
```

```
sort Vnil := &( _ & Vnil );
sort (Vcons a n v) := let sk := skip_first (fun x ⇒ Nat.leb x a) v in &(_ &
Vcons a (sort sk.2).2).
```

Here we prove that the recursive call is correct as skip preserves the size of its argument

```
Next Obligation.
  red. simpl.
  eapply clos_trans_stepr_refl.
  simpl. apply (t_direct_subterm_1_1 _ _ _ (&(_ & v).2)).
  refine (skip_first_subterm _ _).
Qed.
```

To prove it we need a few supporting lemmas, we first write a predicate on vectors equivalent to *List.forall*.

```
Equations forall_vect {A} (p : A → bool) {n} (v : vector A n) : bool :=
forall_vect _ Vnil := true;
forall_vect p (Vcons x n v) := p x && forall_vect p v.
```

```
Require Import Bool.
```

By functional elimination it is easy to prove that this respects the implication order on predicates

```
Lemma forall_vect_impl {A} p p' {n} (v : vector A n)
      (fp : ∀ x, p x = true → p' x = true) :
  forall_vect p v = true → forall_vect p' v = true.
Proof.
  funelim (forall_vect p v). auto.
  simp forall_vect. rewrite !andb_true_iff; intuition auto.
Qed.
```

We now define a simple-minded sorting predicate

```
Inductive sorted : ∀ {n}, vector nat n → Prop :=
| sorted_nil : sorted Vnil
| sorted_cons x n (v : vector nat n) :
    forall_vect (fun y ⇒ Nat.leb x y) v = true →
    sorted v → sorted (Vcons x v).
```

Again, we show this by repeat functional eliminations.

```
Lemma fn_sorted n (v : vector nat n) : sorted (sort v).2.
Proof.
  funelim (sort v).
```

The first elimination just gives the two sort cases.

```
- constructor.
  - constructor; auto.
```

Here we have a nested call to skip_first, for which the induction hypothesis holds:

```
  H : sorted (sort (skip_first (fun x : nat ⇒ x ≤? h) t).2).2
  ============================
```

forall_vect (fun y : nat ⇒ h ≤? y) (sort (skip_first (fun x : nat ⇒ x ≤? h) t).2).2 = true

We can apply functional elimination likewise, even if the predicate argument is instantiated here.

funelim (skip_first (fun x : nat ⇒ Nat.leb x h) t); *simp* *sort* *forall_vect* in *; simpl in *.

After further simplifications, we get:

```
Heq : (h0 ≤? h) = false
H : sorted (Vcons h0 (sort (skip_first (fun x : nat ⇒ x ≤? h0) t).2).2)
============================
(h ≤? h0) && forall_vect (fun y : nat ⇒ h ≤? y) (sort (skip_first (fun x
: nat ⇒ x ≤? h0) t).2).2 = true
```

This requires inversion on the sorted predicate to find out that, by induction, $h0$ is smaller than all of fn (skip_first ...), and hence h is as well. This is just regular reasoning. Just note how we got to this point in just two invocations of *funelim*.

```
depelim H.
rewrite andb_true_iff.
enough (h ≤? h0 = true). split; auto.
eapply forall_vect_impl in H.
apply H.
intros x h0x. simpl. rewrite Nat.leb_le in *. omega.
rewrite Nat.leb_le, Nat.leb_nle in *. omega.
Qed.
```

Pattern-matching and axiom K

```
Module KAxiom.
```

By default we allow the K axiom, but it can be unset.

```
Unset Equations WithK.
```

In this case the following definition fails as K is not derivable on type A.

```
Fail Equations K {A} (x : A) (P : x = x → Type) (p : P eq_refl) (H : x
= x) : P H :=
  K x P p eq_refl := p.
```

However, types enjoying a provable instance of the K axiom are fine. This relies on an instance of the *EqDec* typeclass for natural numbers. Note that the computational behavior of this definition on open terms is not to reduce to p but pattern-matches on the decidable equality proof. However the defining equation still holds as a *propositional* equality.

```
Equations K (x : nat) (P : x = x → Type) (p : P eq_refl) (H : x = x) : P
H :=
  K x P p eq_refl := p.
```

```
  Print Assumptions K. (* Closed under the global context *)

End KAxiom.
```

Going further More examples are available at http://mattam82.github.io/Coq-Equations/examples

Chapter 2

Manual

2.1 Vernacular Commands

2.1.1 Equations

The `Equations` command takes a few options using the syntax

`Equations`(*opts*) `f` ...

- `noind`: Do not generate the inductive graph of the function and the derived eliminator.
- `noeqns`: Do not generate the equations correponding to the (expanded) clauses of the program. This implies `noind`.
- `struct` x: Declare the function structurally recursive on variable x.

The syntax of Equations itself is described in this paper Mangin and Sozeau (2017).

2.1.2 Global Options

The `Equations` command obeys a few global options:

- `Equations Transparent`: governs the opacity of definitions generated by `Equations`. By default this is off and means that definitions are declared *opaque* for reduction, avoiding spurious unfoldings when using the `simpl` tactic for example. The `simp c` tactic is favored in this case to do simplifications using the equations generated for `c`.
- `Equations WithK`: governs the use of the `K` axiom. By default on. When switched off, equations will look for a provable instance of the `K` axiom on the types it needs (through `EqDec` instances for example), and report an error if it cannot find any.

2.1.3 Derive

Equations comes with a suite of deriving commands that take inductive families and generate definitions based on them. The common syntax for these is:

$$\texttt{Derive } \mathbf{C}_1 \ldots \mathbf{C}_n \texttt{ for } \mathbf{ind}_1 \ldots \mathbf{ind}_n.$$

Which will try to generate an instance of type class **C** on inductive type **Ind**. We assume $\mathbf{ind}_i : \Pi\Delta.s$. The derivations provided by Equations are:

- **DependentEliminationPackage**: generates the dependent elimination principle for the given inductive type, which can differ from the standard one generated by Coq. It derives an instance of the class

  `Equations.DepElim.DependentEliminationPackage`.

- **Signature**: generate the signature of the inductive, as a sigma type packing the indices Δ (again as a sigma type) and an object of the inductive type. This is used to produce homogeneous constructions on inductive families, by working on their packed version (total space in HoTT lingo). It derives an instances of the class `Equations.Signature.Signature`.

- **NoConfusion**: generate the no-confusion principle for the given family, which embodies the discrimination and injectivity principles for (non-propositional) inductive types. It derives an instance of the class

  `Equations.DepElim.NoConfusionPackage`.

- **EqDec** This derives a decidable equality on C, assuming decidable equality instances for the parameters and supposing any primitive inductive type used in the definition also has decidable equality. If successful it generates an instance of the class (in `Equations.EqDec`):

  ```
  Class EqDec (A : Type) :=
    eq_dec : forall x y : A, { x = y } + { x <> y }.
  ```

- **Subterm**: this generates the direct subterm relation for the inductive (asuming it is in `Set` or `Type`) as an inductive family. It then derives the well-foundedness of this relation and wraps it as an homogeneous relation on the signature of the datatype (in case it is indexed). These relations can be used with the `by rec` clause of equations. It derives an instance of the class `Equations.Classes.WellFounded`.

Bibliography

Ana Bove and Alexander Krauss and Matthieu Sozeau. *Partiality and recursion in interactive theorem provers – an overview*. *Mathematical Structures in Computer Science*, FirstView:1–51, 2 2015. ISSN 1469-8072.

Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. *Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant*. *Functional and Logic Programming*, pages 114–129, 2006.

Coq. The Coq proof assistant. `coq.inria.fr`.

Healfdene Goguen, Conor McBride, and James McKinna. Eliminating Dependent Pattern Matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.

Cyprien Mangin and Matthieu Sozeau. Equations Reloaded. Submitted, 2017.

Conor McBride and James McKinna. *The view from the left.* *J. Funct. Program.*, 14(1):69–111, 2004.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Nicolas Oury. Pattern matching coverage checking with dependent types using set approximations. In Aaron Stump and Hongwei Xi, editors, *PLPV*, pages 47–56. ACM, 2007.

Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.

Tom Schrijvers, Simon P. Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 341–352, New York, NY, USA, 2009. ACM.

Matthieu Sozeau. Program-ing Finger Trees in Coq. In *ICFP'07*, pages 13–24, Freiburg, Germany, 2007. ACM Press.

Matthieu Sozeau. Equations: A Dependent Pattern-Matching Compiler. In *First International Conference on Interactive Theorem Proving*. Springer, July 2010.