EQUATIONS 1.2 Reference Manual

Matthieu Sozeau

March 21, 2019

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 **?** 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 **?**, the most recent one is described in **?**. This manual provides a documentation of the plugin commands (chapter 1) followed by a tutorial using basic examples (chapter 2). More elaborate examples are available at http://mattam82.github.io/Coq-Equations/examples.

This manual describes version `1.2` 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

Manual

1.1 The Equations Vernacular

1.1.1 Syntax of programs

In the grammar, $\overrightarrow{t}$ denotes a possibly empty list of t, $\overrightarrow{t}^{+}$ a non-empty list. Concrete syntax is in `typewriter` font. The syntax allows the definition of

term, type	t, τ	$::=$	$x \mid \lambda x : \tau, t, R \mid \forall x : \tau, \tau' \mid \lambda\{\overrightarrow{\overrightarrow{up} := t}^{+}\} \cdots$
binding	d	$::=$	$(x : \tau) \mid (x := t : \tau)$
context	Γ, Δ	$::=$	$\overrightarrow{d}$
programs	$progs$	$::=$	$prog \ \overrightarrow{mutual}.$
mutual programs	$mutual$	$::=$	$\texttt{with}\ p \mid where$
where clause	$where$	$::=$	$\texttt{where}\ p \mid \texttt{where}\ not$
notation	not	$::=$	$"string" := t\ (: scope)?$
program	$p, prog$	$::=$	$\texttt{f}\ \Gamma : \tau\ (\texttt{by}\ annot)? := clauses$
annotation	$annot$	$::=$	$\texttt{struct}\ x? \mid \texttt{wf}\ t\ R?$
user clauses	$clauses$	$::=$	$\overrightarrow{cl}^{+} \mid \{\ \overrightarrow{cl}\ \}$
user clause	cl	$::=$	$\texttt{f}\ \overrightarrow{up}\ n?\ ; \mid \overrightarrow{\mid up}^{+}\ n?\ ;$
user pattern	up	$::=$	$x \mid \_ \mid \textsf{C}\ \overrightarrow{up} \mid \texttt{?(}\,t\,\texttt{)} \mid \texttt{!}$
user node	n	$::=$	$:= t\ \overrightarrow{where} \mid \texttt{with}\ t\ \overrightarrow{,t} := clauses$

Figure 1.1: Definitions and user clauses

toplevel mutual (`with`) and nested (`where`) structurally recursive definitions. Notations can be used globally to attach a syntax to a recursive definition, or locally inside a user node. A single program is given as a tuple of a (globally fresh) identifier, a signature and a list of user clauses (order matters), along with an optional recursion annotation (see next section). The signature is simply a list of bindings and a result type. The expected type of the function `f` is then

$\forall \, \Gamma, \tau$. An empty set of clauses denotes that one of the variables has an empty type.

Each user clause comprises a list of patterns that will match the bindings Γ and an optional right hand side. Patterns can be named or anonymous variables, constructors applied to patterns, the inaccessible pattern `?(t)` (a.k.a. "dot" pattern in AGDA) or the empty pattern `!` indicating a variable has empty type (in this case only, the right hand side must be absent). Patterns are parsed using COQ's regular term parser, so any term with implicit arguments and notations which desugars to this syntax is also allowed.

A right hand side can either be a program node returning a term t potentially relying on auxiliary definitions through local `where` clauses, or a `with` node. Local `where` clauses can be used to define nested programs, as in HASKELL or AGDA, or local notations. They depend on the lexical scope of the enclosing program. As programs, they can be recursive definitions themselves and depend on previous `where` clauses as well: they will be elaborated to dependent let bindings. The syntax permits the use of curly braces around a list of clauses to allow disambiguation of the scope of `where` and `with` clauses. The $\lambda\{$ syntax (using a unicode lambda attached to a curly brace) extends COQ's term syntax with pattern-matching lambdas, which are elaborated to local `where` clauses. A local `with` t node essentialy desugars to a program node with a local `where` clause taking all the enclosing context as arguments plus a new argument for the term t, and whose clauses are the clauses of the `with`. The `with` construct can be nested also by giving multiple terms, in which case the clauses should refine a new problem with as many new patterns.

1.1.2   Generated definitions

Upon the completion of an EQUATIONS definition, a few supporting lemmas are generated.

Equations

Each compiled clause of the program or one of its subprograms defined implicitly by `with` or explicitly by `where` nodes gives rise to an equation. Note that the clauses correspond to the program's splitting tree, i.e. to the expansion of pattern-matchings, so a single source clause catching multiple cases can correspond to multiple equations. All of these equations are registered as hints in a rewrite hint database named f, which can be used by the `simp` or `autorewrite` tactic afterwards. The `simp` f tactic is just an alias to `autorewrite with` f. The equation lemmas are named after the position they appear in in the program, and are of the form f_*clause_n_equation_k*.

In case the program is well-founded, EQUATIONS first generates an unfolded definition named f_unfold corresponding to the 1-unfolding of the recursive definition and shows that it is extensionally equal to f. This unfolding equation is used to generate the equations associated to f, which might also refer to the unfolded versions of subprograms. Well-founded recursive definitions can hence

generate a set of equations that is not terminating as an unconditional rewrite system.

Elimination principle

EQUATIONS also automatically generates a mutually-inductive relation corresponding to the graph of the programs, whose first inductive is named f_ind. It automatically shows that the functions respects their graphs (lemma f_ind_fun) and derives from this proof an elimination principle named f_elim. This eliminator can be used directly using the `apply` tactic to prove goals involving a call to the function(s). One has to provide predicates for each of the toplevel programs and the `where` subprograms (`with` subprograms's predicates follow from their enclosing programs).

In case the program has a single predicate, one can use the `funelim` *call* tactic to launch the elimination by specifying which call of the goal should be used as the elimination target. A most general predicate is inferred in this case.

1.1.3 Logic parameterization

EQUATIONS comes with two possible instances of its library, one where equality is COQ's standard equality **eq** in `Prop` and another where equality is proof-relevant and defined in `Type`. The first can be used simply by requiring `Equations.Equations`, while the later can be used by requiring `Equations.Type.All` instead. The two libraries are qualified by the `Equations.Prop` and `Equations.Type` prefixes. When refering to classes in the following, one can find their definition in the respective prefix. In other words, Classes.EqDec might refer to Equations.Prop.Classes.EqDEc or Equations.Type.Classes.EqDEc depending on the logic used.

1.1.4 Local Options

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

$$\text{Equations}(opts)\ \mathsf{f}\ \ldots$$

- `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`.

One can use the `Equations?` syntax to use the interactive proof mode instead of obligations to resolve holes in the term or obligations comming from well-founded recursive definitions. BEWARE that the use of the `abstract` tactical is not well-supported in this mode.

1.1.5   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 With Funext` (since v1.2): governs the use of the functional extensionality axiom to prove the unfolding lemma of well-founded definitions, which requires extensionality of the functional. By default *on*. When this flag is off, the unfolding lemmas of well-founded definitions might fail to be proven automatically and be left to the user as an obligation. To prove this obligation, the user is encouraged to use the `Equations.Init.unfold_recursor` tactic to help solve goals of the form

$$\mathsf{FixWf}\ x\ f = \mathsf{f\_unfold}\ x$$

.

- `Equations With UIP` (since v1.2): governs the use of instances of `Classes.UIP` derived by the user, or automatically from instances of the decidable equality class `Classes.EqDec`. By default *off*. When switched on, equations will look for an instance of **UIP** when solving equalities of the form

$$\forall (e : x = x :> A), P\,e$$

, i.e. to apply the deletion rule to such equations, or to unify indices of constructors for inductive families without a **NoConfusionHom** instance. It will report an error if it cannot find any. Note that when this option is on, the computational behavior of EQUATIONS definitions on open terms does not follow the clauses: it might block on the uip proof (for example if it is a decidable equality test). The rewriting equations and functional elimination principle can still be derived though and are the prefered way to reason on the definition.

- `Equations WithK` DEPRECATED. Use `With UIP` and declare your own version of the UIP axiom as a typeclass instance. Governs the use of the K axiom. By default *off*. The computational behavior of definitions using axioms changes entirely: their reduction will get stuck even on closed terms. It is advised to keep such definitions opaque and use the derived rewriting equations and functional elimination principle to reason on them.

- `Equations Derive Equations` (since v1.2) This sets the default for the generation of equations, governed by the local `eqns`/`noeqns` flags.

- `Equations Derive Eliminator` (since v1.2) This sets the default for the generation of the graph and functional elimination principle associated to a definition, governed locally by the `ind`/`noind` flags.

1.2 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:

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

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

  `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, as an heterogeneous relation. It embodies the discrimination and injectivity principles for the total space of the given inductive family: i.e. $\Sigma\Delta, \mathsf{I}\ \overline{\Gamma}\ \Delta$ for a family $\mathsf{I} : \forall\Gamma, \Delta \to \texttt{Type}$ where Γ are (uniform) parameters of the inductive and Δ its indices.

  It derives an instance of the class `Classew.NoConfusionPackage`.

- **NoConfusionHom**: generate the *homogeneous* no-confusion principle for the given family, which embodies the discrimination and injectivity principles for (non-propositional) inductive types. This principle can be derived if and only if the no-confusion property on the inductive family instance reduces to equality of the non-forced arguments of the constructors. In case of success it generates an instance of the class `Classes.NoConfusionPackage` for the type $\mathsf{I}\ \Delta\ \Gamma$ applicable to equalities of two objects in the *same* instance of the family I.

- **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 `Classes.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 wf` clause of equations. It derives an instance of the class `Classes.WellFounded`.

1.3 `dependent elimination`

The `dependent elimination` tactic can be used to do dependent pattern-matching during a proof, using the same engine as Equations.

Its syntax is:

`dependent elimination` *ident* `as` [up | .. | up].

It takes a list of patterns (see figure 1.1) that should cover the type of *ident* and generates the corresponding subgoals.

1.4 `simp`

The `simp` $f_1 \ldots f_n$ tactic is an alias to

```
autorewrite with f₁ ... fₙ;
try typeclasses eauto with Below subterm_relation f₁ ... fₙ
```

It can be used to simplify goals involving equations definitions $f_1 \ldots f_n$, by rewriting with the equations declared for the constants in the associated rewrite hint database and trying to solve the goal using the hints declared in the associated "auto" hint database, both named f.

1.5 Functional elimination

The `funelim` t tactic can be used to launch a functional elimination proof on a call t of an Equations-defined function $t = f$ *args* (the eliminator is named f_elim). By default, it will generalize the goal by an equality betwee `f args` and a fresh call to the function `f args'`, keeping information about initial arguments of the function before doing elimination. This ensures that subgoals do not become unprovable due to generalization. This might produce complex induction hypotheses that are guarded by dependent equalities between the initial arguments and the recursive call arguments. These can be simplified by instantiating the induction hypotheses sufficiently and providing reflexive equality proofs to instantiate the equalities.

A variant `apply_funelim` t simply applies the eliminator without generalization, avoiding the generation of (dependent) equalities. Note that in this case (as when using Coq's built-in `induction` tactic) one may have to explicitely

generalize the goal by equalities (e.g. using the `remember` tactic) if the function call being eliminated is not made of distinct variables, otherwise it can produce unprovable subgoals.

Finally, for mutual or nested programs, no automation is provided yet. The user has to invoke the functional elimination directly, e.g. using

$$\texttt{eapply } (\mathsf{f\_elim}\ P_1 \ldots P_n)$$

providing predicates for each of the nested or mutual function definitions in f (use `About` $\mathsf{f\_elim}$ to figure out the predicates to be provided).

Chapter 2

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 **?** 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 Program.
From Equations Require Import Equations.
```

2.1 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.
```

`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.

2.2   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.

2.3   Building up

2.3.1   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 explicitly one can use the syntax $(A{:=}B)$, renaming that implicit argument to B in this particular case

2.3.2 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:

$app\_elim :$
$\forall\ P : \forall\ (A : \mathtt{Type})\ (l\ l' : \mathsf{list}\ A),\ \mathsf{list}\ A \to \mathtt{Prop},$
$(\forall\ (A : \mathtt{Type})\ (l' : \mathsf{list}\ A),\ P\ A\ \mathsf{nil}\ l'\ l') \to$
$(\forall\ (A : \mathtt{Type})\ (a : A)\ (l\ l' : \mathsf{list}\ A),$
$P\ A\ l\ l'\ (\mathsf{app}\ l\ l') \to P\ A\ (a :: l)\ l'\ (a :: \mathsf{app}\ l\ l')) \to$
$\forall\ (A : \mathtt{Type})\ (l\ l' : \mathsf{list}\ A),\ P\ A\ l\ l'\ (\mathsf{app}\ l\ l')$

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 **?** that derives induction principles from function definitions.

2.3.3 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 **?** community and was first presented and implemented in the Epigram language **?**.

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 with 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) with 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.

2.4   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 **?**). In this case the automatic tactic is able to derive by itself that $n \neq m \rightarrow$ 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 with pf eq_refl := { | x :=! x };
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 by applying it to reflexivity. 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 (**??**).

  However, in this case, one could also write an empty set of clauses for the `with` subprogram, as Equations applies a heuristic in case of an empty set of clause: it tries to split each of the variables in the context to find an empty type.

- In the second case, we simply return the head of the list, disregarding the proof.

2.4.1 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 **?**, which are a similar concept to the generalization of algebraic datatypes to GADTs in functional languages like Haskell **?**. 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 : Type) (x : A) : A → 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:

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

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". When they are variables the inaccessibility annotation is optional.

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 $(A : \text{Type})$: `nat` $\to$ `Type` := | Vnil : vector A O | Vcons : A $\to \forall\ n$: `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 (Vcons a 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 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.

2.4.2   Derived notions, No-Confusion

For this to work smoothlty, the package requires some derived definitions on each (indexed) family, which can be generated automatically using the generic *Derive* command. Here we ask to generate the signature, heterogeneous no-confusion and homogeneous no-confusion principles for vectors:

Derive NoConfusion for nat.
Derive Signature NoConfusion *NoConfusionHom* for vector.

The precise specification of these derived definitions can be found in the manual section (§1.1). Signature is used to "pack" a value in an inductive family with its index, e.g. the "total space" of every index and value of the family. This can be used to derive the heterogeneous no-confusion principle for the family, which allows to discriminate between objects in potentially different instances/fibers of the family, or deduce injectivity of each constructor. The *NoConfusionHom* variant derives the homogeneous no-confusion principle between two objects in the *same* instance of the family, e.g. to simplify equations of the form Vnil = Vnil :> vector A 0. This last principle can only be defined when pattern-matching on the inductive family does not require the K axiom and will otherwise fail.

2.4.3   Unification and indexed datatypes

Back to our example, 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 _ _ _:

$\forall\ P : \forall\ (A : \mathsf{Type})\ (n : \mathsf{nat}),\ \text{vector}\ A\ (\mathsf{S}\ n) \rightarrow \text{vector}\ A\ n \rightarrow \mathsf{Prop},$
$(\forall\ (A : \mathsf{Type})\ (n : \mathsf{nat})\ (a : A)\ (v : \text{vector}\ A\ n),$
$\quad P\ A\ n\ (\text{Vcons}\ a\ v)\ v) \rightarrow$
$\forall\ (A : \mathsf{Type})\ (n : \mathsf{nat})\ (v : \text{vector}\ A\ (\mathsf{S}\ n)),\ P\ A\ n\ v\ (\text{vtail}\ v)$

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 $\{A\ n\}$ $(v : \text{vector}\ (\text{vector}\ A\ n)\ n) : \text{vector}\ A\ n :=$
diag $(n{:=}\mathsf{O})$ Vnil $:=$ Vnil ;
diag $(n{:=}\mathsf{S}\ \_)$ (Vcons (Vcons $a\ v$) v') $:=$
  Vcons a (diag (vmap vtail v')).

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

2.4.4   Recursion

Notice how in the diag example above we explicitly 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 definition also fails:

Fail Equations diag' $\{A\ n\}$ $(v : \text{vector}\ (\text{vector}\ A\ n)\ n) : \text{vector}\ A\ n :=$
diag' Vnil $:=$ Vnil ;
diag' (Vcons (Vcons $a\ v$) 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:

Equations id $(n : \mathsf{nat}) : \mathsf{nat}$ by wf n lt $:=$
  id $\mathsf{0} := 0$;
  id $(\mathsf{S}\ n') := \mathsf{S}\ (\text{id}\ n')$.

Here id is defined by well-founded recursion on lt on the (only) argument n using the by wf annotation. At recursive calls of id, obligations are generated to show that the arguments effectively decrease according to this relation. Here the proof that $n' < \mathsf{S}\ n'$ 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 they must relate 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. This provides course-of-values or so-called "mathematical" induction on these objects, reflecting the structural recursion criterion in the logic.

```
Derive 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}. The default obligation tactic defined in Equations.Init includes a proof-search for *subterm* proofs which can resolve the recursive call obligation automatically in this case.

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

For the diagonal, it is easier to give n as the decreasing argument of the function, even if the pattern-matching itself is on vectors:

```
Equations diag' {A n} (v : vector (vector A n) n) : vector A n by wf n :=
diag' Vnil := Vnil ;
diag' (Vcons (Vcons a v) v') :=
```

Vcons a (diag' (vmap vtail v')).

One can check using `Extraction` diag' that the computational behavior of diag' is indeed not dependent on the index n.

Pattern-matching and axiom K

To use the K axiom or UIP with `Equations`, one *must* first set an option allowing its use during dependenet pattern-matching compilation.

`Module` KAxiom.

By default we disallow the user of UIP, but it can be set.

`Set Equations` *With UIP*.

`Module` WithAx.

The user must declare this axiom itself, as an instance of the UIP class.

`Axiom` *uipa* : $\forall$ A, UIP A.
`Local` *Existing* `Instance` *uipa*.

In this case the following definition uses the UIP axiom just declared.

`Equations` K $\{A\}$ $(x : A)$ $(P : x = x \rightarrow \texttt{Type})$ $(p : P$ `eq_refl`$)$
        $(H : x = x) : P\ H :=$
  K x P p `eq_refl` $:= p$.

`End` WithAx.

Note that the definition loses its computational content: it will get stuck on an axiom. We hence do not recommend its use.

Equations allows however to use constructive proofs of UIP for types enjoying decidable equality. The following example relies on an instance of the *EqDec* typeclass for natural numbers, from which we can automatically derive a UIP nat instance. 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, and the definition of K' is axiom-free.

`Equations` K' $(x : \texttt{nat})$ $(P : x = x \rightarrow \texttt{Type})$ $(p : P$ `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