Equations reloaded

Cyprien Mangin, Matthieu Sozeau

Equations Reloaded

Cyprien Mangin
Inria Paris & IRIF - Université Paris 7 Diderot
France
cyprien.mangin@m4x.org

Matthieu Sozeau
Inria Paris & IRIF - Université Paris 7 Diderot
France
matthieu.sozeau@inria.fr

Abstract

Equations is a plugin for the Coq proof assistant which provides a notation for defining programs by dependent pattern-matching and well-founded recursion and derives useful proof principles for demonstrating properties about them. We present here an improved design and rewriting of its implementation that avoids the use of axioms and reliance on $\mathcal{L}_{\text{tac}}$ programming, providing a feature-rich function definition package for Coq as a definitional extension to the Coq kernel. At the core of the system is a simplifier for dependent equalities that can be reused to define enhanced versions of the dependent elimination tactics of Coq, namely dependent `destruction` and `inversion`. We introduce verified optimizations of the simplifier that allow generating smaller and simpler Equations definitions and proof terms for these tactics in general. We demonstrate the applicability of the tool on a medium-sized example of a reflective decision procedure.

Keywords Coq, dependent pattern-matching, proof assistants

1 Introduction

Equations is a tool designed to help with the definition of programs in the setting of dependent type theory, as implemented in the Coq proof assistant. Equations provides a syntax for defining programs by dependent pattern-matching and well-founded recursion and compiles them down to the core type theory of Coq, using the primitive eliminators for inductive types, accessibility and equality. In addition to the definitions of programs, it also automatically derives useful reasoning principles in the form of propositional equations describing the functions, and an elimination principle for calls to this function. It realizes this using a purely *definitional* translation of high-level definitions to core terms, without changing the core calculus in any way. This is to contrast with *axiomatic* implementations of dependent pattern-matching like the one of Agda [16], where the justification of dependent-pattern matching definitions in terms of core rules is proven almost entirely internally as in [4] but the core system is extended with evidence-free higher-level rules directly, simplifying the implementation work substantially.

Equations Reloaded, Version of December 22, 2017, Draft

.

At the user level though, Equations definitions closely resemble Agda definitions, for example a typical definition is the following, where we first recall the inductive definitions of length-indexed vectors and numbers in a finite set indexed by its cardinality.

```
Inductive vector (A : Type) : nat → Type :=
| nil : vector A 0
| cons (a : A) n (v : vector A n) : vector A (S n).

Inductive fin : nat → Set :=
| fz : ∀ n, fin (S n)
| fs : ∀ n, fin n → fin (S n).

Equations nth {A n} (v : vector A n) (f : fin n) : A :=
nth (cons x _ _) (fz n) := x;
nth (cons _ ?(n) v) (fs n f) := nth v f.
```

The nth function implements a safe lookup in the vector v as fin n is only inhabited by valid positions in v. The conciseness provided by dependent pattern-matching notation includes the ability to elide impossible cases of pattern-matching: here there is no clause for the nil case of vectors as the type fin 0 is empty. Also notice the inaccessible ?(n) annotation for the argument of the cons constructor in the second clause: as it is uniquely determined to be equal to the n argument of the fs constructor, it *must* be written as an inaccessible pattern or a wildcard as in the first clause. From this definition, Equations will generate a function called nth which obeys the equalities given by the user as clauses, along with a few helper lemmas to work with this definition, notably its associated elimination principle.

An issue of trust While the difference of viewpoint between Agda and Equations might seem only aesthetic and of little pratical relevance, this has far reaching consequences. Software is subject to bugs, and any extension of the core calculus of a proof assistant should be done with the utmost care as the entirety of developements done with it rely on the correctness of its kernel. Simplicity is hence a big plus to have trust in a given proof assistant. There is not only the possibility of bugs that we want to avoid, but, in particular in the case of dependent pattern-matching, there are metatheoretical properties we want to ensure that are hard to check if the calculus is extended with new rules. One such property is compatibility with certain independent axioms like uniqueness of identity proofs (hereafter, UIP, a.k.a Streicher's axiom K [9]) or the univalence principle [19]. These two axioms are inconsistent in general.

1.1 A short history of dependent pattern-matching

The first version of dependent pattern-matching was introduced by Coquand in [7], axiomatically defining a notation for dependent pattern-matching programs, and later refined by McBride *et al.* [12], using a definitional translation, with both systems assuming the UIP principle from the start. In [8], dependent pattern-matching was explained in terms of simplification of heterogeneous equalities which were defined using the UIP principle (although, in his PhD [12], McBride already hinted at the fact that a version using equality of telescopes, potentially avoiding the use of UIP, would be possible as well). Agda implements by default this notion of dependent pattern-matching, assuming the K principle.

Comes the introduction of Homotopy Type Theory and Univalence [17], whose central principle contradicts directly the uniqueness of identity proofs principle. Informally, in Homotopy Type Theory, one is interested in the higher-dimensional structure of equality types, which are shown to form weak ∞-groupoids [10, 20]. This is in direct conflict with the UIP principle which states, in terms of HoTT, that every type is an HSet, otherwise said that the higher-dimensional structure of identity at any type is trivial.

To remedy this conflict, and give a meaning to dependent pattern-matching compatible with univalence, one has to move to a view of heterogeneous equality which does not rely on UIP. This can be done using telescopes, or the notion of "path over a path", easily encoded in pure type theory using sigma types. This was done in [6] for an "axiomatic" version implemented in Agda and [11] for a "definitional" translation in Coq, which clearly circumbscribed the cases where the UIP principle was necessary during compilation. At this point, UIP, or the assumption that some type is an HSet was necesary for the deletion rule (to dependently eliminate an equality $e : t = t$) and to simplify problems of injectivity between indexed inductive types.

Since then, Cockx [5] introduced an alternative solution to injectivity which can remove the later uses of the UIP principle, justified by reasoning on higher-dimensional equalities. This ought to bring a happy conclusion to the "--without-K" story of Agda, which should enforce that UIP is not provable and had a history of bug reports where proofs of UIP were found repeatedly, fix after fix. This result should settle these issues once and for all by providing a solid theoretical background to the axiomatic dependent pattern-matching implemented in Agda. However, note that even this last solution involves constructing "out of thin air" a substitution that should come from a chain of computationally-relevant type equivalences. While we were able to reproduce this result, and checked the reasoning used to build this substitution, any change to the core calculus implies a requirement of trust towards its implementation, whose burden we avoid in the case of Equations by providing a definitional translation.

1.2 UIP versus Univalence

In practice both the UIP and the Univalence principle have value. In a theory with UIP built-in, for example in a version of the calculus of constructions with a definitionally proof-irrelevant `Prop` (like in Lean [2]), one can formulate dependent pattern-matching compilation by working with equalities in `Prop` and freely use UIP to simplify any pattern-matching problems. Moreover, this compilation is guaranteed to have good computational behavior as all the decoration added by the compilation are proof manipulations that are guaranteed to be computationally irrelevant by construction. In the setting of Coq, this has an impact on extraction: extraction of definitions by Equations when using the equality in `Prop` removes all the proof manipulations involved, leaving only the computational content. This is important in case one wants to actually compute with these definitions or their extraction, e.g. through a certified compiler like CertiCoq [1] that does erasure of proofs.

In contrast, Univalence forces to move to a proof-relevant equality type (defined in `Type`) which cannot be erased, but provides additional proof principles, like the ability to transport theories by isomorphisms, and features like Higher Inductive Types. It is hence useful to design the system so that it is as agnostic as possible about the equality used.

1.3 Contributions

In its first version [18], the Equations tool was relying on heterogeneous equality (a.k.a. "John-Major" equality) to implement the so-called "specialization by unification" [8] necessary to witness dependent pattern-matching compilation. It was also implemented in a rather prototypical fashion, using large amounts of fragile $\mathcal{L}_{\text{tac}}$ definitions and tricks to implement simplification.

In this paper, we present a new implementation of specialization based on dependent equalities which removes these limitations and introduce a handful of new features which make the tool more widely applicable and useful. Our main contributions are:

- An extended source language for Equations including `where` nodes for defining subprograms. This allows in particular the definition of nested well-founded recursive functions on inductive families.
- A cleaner elaboration of Equations definitions, avoiding the use of a construction of [8] which was forcing unnecessary applications of UIP. We also avoid the use of proof-irrelevance for proving unfolding lemmas of recursive definitions. The system is parameterized by a logic definition so that it can use the logic in `Prop`, to be used with the proof-irrelevance principle, or in `Type`, for univalent settings.
- A new dependent pattern-matching simplification algorithm, implemented in ML, and compatible with both the UIP principle and univalence. This algorithm

produces axiom-free proof terms to be checked by the Coq kernel, and can be used independently from the Equations elaboration algorithm.
- An optimized compilation: by doing a first phase of simplification of dependent pattern-matching problems before case-splitting, we produce smaller and simpler proof terms.
- A new dependent elimination tactic. Based on this compilation engine, which has a careful treatment of names, we define a new dependent elimination tactic that can advantageously replace the `inversion` and dependent `destruction` tactics, letting the user specify cleanly the naming and ordering of branches when applying eliminations on inductive families.

This new system is freely available[1] and has been tested on a variety of examples, including a proof of strong normalization for predicative System F [11] and a reflexive tactic for deciding equality of polynoms. Our personal experience shows that the notational facilities provided by Equations definitions and the proof principles that are automatically derived from them provide a comfortable and efficient framework for dealing with complex definitions in the Coq proof assistant. We are aiming to collect more opinions on the subject from more varied audiences through the next experimental release of Equations.

Structure of the paper The paper is organized as follows: in §2 we introduce the source language of Equations and present its architecture and its main features by way of examples. In §3 we recall the theory of dependent pattern-matching compilation using equality of sigma types. In §4 we present our ML compiler and focus on how to deal with the problematic rule of injectivity of constructors. In §5 we develop an optimization of simplification that can reduce term size and complexity of definitions, and in §6 we put these together to produce a new dependent elimination tactic. In §7, we take a new look at the treatment of structural and well-founded recursive definitions in the new setting. §8 provides a short walkthrough of an example of using the new system to develop a reflexive decision procedure. Finally we review related work in §9 and conclude.

2 Interpreting equations

The source language of Equations is a declarative syntax for stating the equations that a definition f should enjoy. The main difference with usual Coq definitions is that the pattern-matching construct (i.e. `match`) is implicit in the definition of clauses for a given function f.

Notations and terminology We will use the notation $\overline{\Delta}$ to denote the set of variables bound by an environment Δ, in the order of declarations, and also to denote lists in general, usually separated by semicolons ;. An *arity* is a term of the form $\Pi\ \Gamma, s$ where s is a sort. A sort (or kind) can be either Prop (categorizing propositions) or Type (categorizing computational types, like bool). An arity is hence always a type. We will ignore universe levels throughout, but the system works with Coq 8.6 featuring typical ambiguity and universe polymorphism, which we use to formalize our constructions. We consider inductive families to be defined in a (elided) global context by an arity $\mathsf{I} : \Pi\ \Delta, s$ and constructors $\mathsf{I}_i : \Pi\ \Gamma_i, \mathsf{I}\ \overline{t}$. Although CIC distinguishes between parameters and indices and our implementation does too, we will not distinguish them in the presentation for the sake of simplicity.

The compilation process starts from a signature and a list of clauses given by the user, as in the example nth above, constructed from the grammar given in figure 1.

`term, type`	$t,\ \tau$	::=	$x \mid \lambda x : \tau, t \mid \Pi x : \tau, \tau' \mid \ldots$
`binding`	d	::=	$(x\ :\ \tau) \mid (x\ :=\ t\ :\ \tau)$
`context`	Γ, Δ	::=	$\overline{d}$
`program`	$prog$	::=	$\mathsf{f}\ \Gamma\ :\ \tau\ :=\ \overline{c}$
`user clause`	c	::=	$\mathsf{f}\ \overline{up}\ n$
`user pattern`	up	::=	$x \mid \mathsf{C}\ \overline{up} \mid \mathtt{?}(t)$
`user node`	n	::=	$:=!\ x \mid \mathtt{with}\ t := \{\ \overline{c}\ \}$
		|	$:=\ t\ \mathtt{where}\ \overline{prog}$
		|	$\mathtt{by\ rec}\ x\ R := prog$

Figure 1. Definitions and user clauses

A program is given as a tuple of a (globally fresh) identifier, a signature and a list of user clauses (order matters). The signature is simply a list of bindings and a result type. The expected type of the function f is then $\Pi\ \Gamma, \tau$. Each user clause comprises a list of patterns that will match the bindings Γ and a right hand side which can either be an empty node ($:=!\ x$), a `with` node adding a pattern to the problem, scrutinizing the value of some term t, a program node returning a term t potentially relying on auxiliary definitions through `where` clauses or a by `rec` $x\ R$ node starting a well-founded recursion on variable x using relation R.

Nested recursion As a first example of a complex Equations definition, we define a function gathering the elements in a rose tree. We define rose trees as trees whose nodes contain lists of trees, i.e. forests.

```
Context {A : Type}.
Inductive rose : Type :=
| leaf (a : A) : rose | node (l : list rose) : rose.
```

This is a nested inductive type we can measure assuming a list_size function for measurings lists. Here we use the usual structural recursion guardness check of Coq that can check that this definition is terminating by unfolding the definition of list_size.

```
Equations size (r : rose) : nat :=
size leaf := 0; size (node l) := S (list_size size l).
```

[1] http://mattam82.github.io/Coq-Equations

However, if we want to program more complex recursions, or rearrange our terms slightly, the limited syntactic guardness check will quickly get in our way. Thanks to the `where` nodes and support of Equations for well-founded recursion, we can define the following function gathering the elements in a rose tree using nested recursion on the list. The function is nesting a well-founded recursion inside another one, based on the measure of rose trees and lists. No syntactic criterion is used to ensure the termination of this definition. Note that the auxilliary definitions type mentions the variable l bound by the enclosing pattern-matching, to pass around information on the size of arguments.

```
Equations elements (r : rose) : list A :=
elements l by rec r (MR lt size) :=
elements (leaf a) := [a];
elements (node l) := aux l _
  where aux x (H : list_size size x < size (node l)) : _ :=
  aux x H by rec x (MR lt (list_size size)) :=
  aux nil _ := nil;
  aux (cons x xs) _ := elements x ++ aux xs _.
```

This kind of nested pattern-matching and well-founded recursion was not supported by previous definition packages for Coq like Function or Program. For now we will focus on the compilation of pattern-matching only, coming back to recursion in §7.

Searching for a covering

The goal of the compiler is to produce a proof that the user clauses form an exhaustive covering of the signature, compiling away nested pattern-matchings to simple case splits. As we have multiple patterns to consider and allow overlapping clauses, there may be more than one way to order the case splits to achieve the same results. We use inaccessible patterns (noted $?(t)$) as in Agda to help recover a sense of what needs to be destructed and what is statically known to have a particular value, but overlapping clauses force the compilation to be phrased as a search procedure. As usual, we recover a deterministic semantics using a first-match rule when two clauses overlap.

program	$prog$	::=	$(\ell_p, \Gamma, rec?, spl)$
recursion	rec	::=	$\mathtt{wf}(t, R) \mid \mathtt{struct}\ x$
context map	c	::=	$\Delta \vdash \overline{p} : \Gamma$
pattern	p	::=	$x \mid \mathrm{C}\ \overline{p} \mid ?(t)$
splitting	spl	::=	$\mathrm{Split}(c, x, ((spl)?)^n)$
		|	$\mathrm{Compute}(c, \ell_c, t, prog^*)$
label	ℓ	::=	$\epsilon \mid \ell.n \quad (n \in \mathbb{N})$

Figure 2. Context mappings and splitting trees

The search for a covering works by gradually refining a *programming problem* $\Delta \vdash \overline{p} : \Gamma$ and building a splitting tree. A programming problem, or context mapping (fig. 2), is a substitution from Δ to Γ, associating to each variable in Γ a pattern p typable in Δ. We start the search with the problem $\Gamma \vdash \overline{\Gamma} : \Gamma$, i.e. the identity substitution on Γ and the list of user clauses. At each point during covering, we can compute the expected target type of the current subprogram by applying this substitution to its declared type τ. In [8] and [18], a specific $\mathrm{f}_{\mathsf{comp}}$ constant was defined to keep track of this type and communicate with tactics, however this introduces unnecessary dependencies and we can do without it. It makes the return type dependent on all arguments of f, which means they must all get specialized during dependent eliminations, while this is not necessary in general.

The search for a covering and building of the splitting tree is entirely standard and unchanged from the previous version (one can refer to [18] for details), so we consider that we are directly given a splitting tree corresponding to our definition.

A splitting can either be:

- A $\mathrm{Split}(\Delta \vdash \overline{p} : \Gamma, x, (s?)^n)$ node denoting that the variable x is an object of an inductive type with n constructors and that splitting it in context Δ will generate n subgoals which are covered by the optional subcoverings s. When the type of x does not unify with a particular constructor's type the corresponding splitting is empty. Otherwise the substitution built by unification determines the context mapping used in each of the subcoverings.
- A $\mathrm{Compute}(\Delta \vdash \overline{p} : \Gamma, \ell, t, \overline{w})$ node, denoting a right-hand side whose definition is t (of type $\tau[\overline{p}]$) under some set of auxiliary local definitions $\overline{w}$. Both `with` and `where` clauses are compiled this way. We will not delve into the details of the compilation of `with` in this paper, the reader can refer to [18] for a detailed treatment. The `where` clauses directly translate to auxiliary local definitions in this representation.

For each (sub)program $(\ell_p, \Gamma, rec?, s)$, the optional rec annotation describes its recursive structure.

- A $\mathtt{wf}(t,\ R)$ annotation denotes an application of the well-founded fixpoint combinator to define the rest of the function ℓ_p described by s. The user has to specify an homogeneous order relation R on the type of the term t which can mention any of the variables in Γ. We will describe this in more detail in section 7.
- A $\mathtt{struct}\ x$ annotation denotes a usual structurally recursive fixpoint of Coq, where x is a single variable declared in the context Γ.

3 Dependent pattern-matching compilation recap

From the splitting tree representation of a program, we want to obtain an actual Coq definition. To do so, we follow the same schema as [8] and [18] with minor modifications. We recall the main construction here.

Packing inductives Before going further, we will simplify our development by considering only homogeneous relations between inductive families. Indeed we can define for any inductive type $\Pi\ \Delta,\ \mathsf{I}\ \Delta$ (any arity in general) a corresponding closed type by wrapping the indices Δ in a dependent sum and both the indices and the inductive type in another dependent sum.

Definition 3.1 (Telescope transformation). For any context Δ, we define packing $\Sigma(\Delta)$ and unpacking $\overline{\Sigma}(\Delta, s)$ by recursion on the context.

$$\Sigma(\epsilon) = \mathsf{unit} \qquad \Sigma(x : \tau, \Delta) = \Sigma x : \tau, \Sigma(\Delta)$$
$$\overline{\Sigma}(\epsilon, s) = \epsilon \qquad \overline{\Sigma}(x : \tau, \Delta, s) = \pi_1\ s, \overline{\Sigma}(\Delta, \pi_2\ s)$$

This is what [5] denote as $\bar{\mathsf{I}}$, and provides a definition of the "total space" described by a family in HoTT terms, using iterated sigma types. We can automatically derive this construction for any inductive type using the `Derive Signature for I` command. This provides in particular a trivial function to inject a value in the signature:

$$\mathsf{signature\_pack} : \forall \Delta (x : \mathsf{I}\ \Delta), \Sigma i : \overrightarrow{\Delta}, I\ \bar{i}$$

Deriving a few constructions

The dependent pattern-matching notation acts as a high-level interface to a unification procedure on the theory of constructors and uninterpreted functions. Our main building block in the compilation process is hence a mechanism to produce witnesses for the resolution of constraints in this theory, that is used to compile `Split` nodes. The proof terms will be formed by applications of simplification combinators dealing with substitution and proofs of injectivity and discrimination of constructors, their two main properties.

The design of this simplifier is based on the "specialization by unification" method developed in [13, 14]. The problem we face is to eliminate an object x of type $\mathsf{I}\ \bar{t}$ in a goal $\Gamma \vdash \tau$ potentially depending on x. We want the elimination to produce subgoals for the allowed constructors of this family instance. To do that, we generalize the goal by fresh variables $\Delta\ (x' : \mathsf{I}\ \overline{\Delta})$ and an equation between telescopes asserting that x' is equal to x, giving us a new, equivalent goal:

$$\Gamma, \Delta, x' : \mathsf{I}\ \overline{\Delta}, \Gamma \vdash (\overline{\Delta}, x') = (\overline{t_i}, x) \to \tau \tag{1}$$

We can apply the standard eliminator for I on x' in this goal to get subgoals corresponding to all its constructors, all starting with an equation relating the indices t of the original instance to the indices of the constructor. We will use the algorithm presented in sections 4 and 5 to simplify these equations.

Apart from uses of the eliminator for equality, the simplifier needs definitions for simplifying equalities between constructors. We need a tactic that can simplify any equality of the shape:

$$(\overline{ts}, \mathsf{C}\ \bar{t}) =_{\Sigma i : \Delta, I\ \bar{i}} (\overline{us}, \mathsf{D}\ \bar{u}) \tag{2}$$

Either giving us equalities between the arguments $\bar{t}$ and $\bar{u}$ (injectivity) that can be further simplified or deriving a contradiction if C is different from D (conflict). McBride *et al.* [14] describe a generic method to derive such an eliminator that can be adapted to work on telescopic equalities instead of heterogeneous equalities ([5] describes it in detail). We implement this construction as another `Derive` scheme in Coq. For any (computational) inductive type $\mathsf{I} : \Pi\ \Gamma, \mathsf{Type}$, we can use `Derive NoConfusion for I` to derive an instance of the type class $\mathsf{NoConfusionPackage}(\Sigma i : \overrightarrow{\Delta}, \mathsf{I}\ \bar{i})$ that provides a proof of isomorphism of the two types:

$$\forall\ x\ y, \mathsf{NoConfusion}\ (\Sigma i : \overrightarrow{\Delta}, \mathsf{I}\ \bar{i})\ x\ y \simeq x =_{\Sigma i : \overrightarrow{\Delta}, \mathsf{I}\bar{i}} y \tag{3}$$

When x and y are of the shape in equation (2), $\mathsf{NoConfusion}\ x\ y$ directly reduces to either True, False or an equality between the arguments $\bar{t}$ and $\bar{u}$. Note that we cannot derive such a principle for families in `Prop` as this would contradict proof-irrelevance: constructors of inductives in `Prop` cannot be discriminated.

4 Crafting terms for Coq

In a first iteration of Equations, the compilation of the splitting tree to a Coq term was mostly done using tactics. Specifically, the part about simplifying equations was implemented by repeatedly matching the shape of the goal against the conclusion of a list of lemmas.

Using $\mathcal{L}_{\mathrm{tac}}$ has the advantage of being simple and easily readable. However, $\mathcal{L}_{\mathrm{tac}}$ as a programming language is not really convenient for our purpose. This is a logic programming language with inherent backtracking capabilities which are useless for us, and with an uncomfortable handling of variables. Moreover, transmitting information between the OCaml code generating a splitting tree and the tactics compiling some parts of this tree is at best inelegant: the previous solution was to generate goals that contained superfluous hypotheses with the missing information (for instance, which variable we wish to eliminate). While this works, it also results in a proof term polluted with this additional context; we can also hope for an improvement in performances with a more direct solution.

There are two more drawbacks that we wish to avoid:

- while manipulating the goal in $\mathcal{L}_{\mathrm{tac}}$, it is possible to lose track of the way the context evolved, and plug the rest of the proof incorrectly into the resulting term, by swapping two variables with the same type for instance. The $\mathsf{f}_{\mathsf{comp}}$ trick used to let us keep track of this evolution, but makes the goal more dependent than it needs to be;
- $\mathcal{L}_{\mathrm{tac}}$ and tactics in general do not allow a good enough control over the terms we are building. Sometimes,

we wish to eliminate a variable in a more subtle way, instead of generating an equality for each index in every case. We will develop this idea below.

In this section, we explain the simplification mechanism we implemented in OCaml. The next section will describe the elimination strategy that we implemented to produce smaller and simpler terms.

4.1 Simplification, from $\mathcal{L}_{\text{tac}}$ to OCaml

We implement simplification as a facility mostly independent from Equations, which is able to take a goal of shape $\overline{u} = \overline{t} \to P$ under a context Γ, and attempts to eliminate the equations $\overline{u} = \overline{t}$, either solving the goal or leaving P as a subgoal after simplification. We provide an interface that could be used from any other plugin, and a tactic which can be used on any goal with the correct shape. Currently it relies on inductive types and lemmas which are defined by Equations, such as sigma types, but it has been implemented so as to be modular and will be made independent in the future.

It works by applying a sequence of so-called *simplification steps*. To each simplification step corresponds one OCaml function which takes a goal $\Gamma \vdash \tau$ and, if it succeeds, returns a term c such that $\Gamma \vdash c : \tau$. Unless the goal was directly solved, for instance when simplifying an equality between two distinct constructors, the term c will contain exactly one existential variable, which is returned as a subgoal $\Gamma' \vdash \tau'$ along with c. Therefore, if we know which simplification steps to apply, it is just a matter of combining these simplification functions.

A note on K There are two simplification steps which make use of K on a given type: ***Deletion***, as expected, and ***NoConfusion*** which requires it on the indices of the inductive type through the ***Pack*** step presented below. In both cases, we do not actually require K directly, but decidable equality at a specific term, that is $\forall x, \{x = t\} + \{x \neq t\}$ for some term t. We can find a proof of this fact by using the typeclasses mechanism of Coq, and also derive it automatically using a `Derive EqDec for I` command. If we can't find such a proof, there is an option to either admit it as an axiom, or fail altogether. In all the examples presented here, we only, if ever, use defined proofs of (pointed) decidable equality. Definitions using K have a different computational behavior and can get stuck on open terms, but as usual we can still derive their equations and elimination principle.

Now we describe each simplification step in order. For each one, we provide the shape of the goals to which it applies, what the goal should look like after it is applied, and anything else which might be relevant. Note that we could also describe each step as an equivalence of telescopes, as was done in [4] for instance; instead, we choose here to show how it acts on a given goal, since we are directly manipulating terms.

Each of these simplification steps apply under a certain context Γ which we do not write most of the time because it will not change. It is also good to keep in mind the equivalence between an equality of telescopes, and a telescope of equalities. This equivalence is made obvious in practice by this first simplification step.

Remove sigma `∀ (e : (p, x) = (q, y)), P e` becomes `∀ (e' : p = q) (e : rew e' x = y), P (sigma_eq e' e)`

This step is trivial and only serves as a mechanical way to ensure that the other simplification steps do not need to deal with equality of telescopes but rather a (curried) telescope of equalities. The function sigma_eq combines the two previous equalities into one well-typed equality between `(p, x)` and `(q, y)`, while rew (a.k.a. ap or subst) rewrites in the type of `x` with the equality e', it is just an application of J.

Deletion `∀ (e : t = t), P e` becomes `P eq_refl`.

This step requires K on the type of `t`, as described above.

Solution `∀ Γ, ∀ (e : x = t), P x e` becomes `∀ Γ', P t eq_refl`.

Here x has to be a variable which does not occur in `t`. This step might require that we manipulate the environment through strengthening. Strengthening is implemented as a function which, from a context, a variable x and a term `t`, computes a context mapping such that the resulting context allows for a well-typed substitution of x by `t`, using J. This is the only case where we need to move variables around in the environment and doing it in OCaml allows us to correctly keep track of each variable thanks to this context mapping.

True and False `∀ (e : True), P e` becomes `P I`
`∀ (e : False), P e` becomes solved

This step is trivial, but is required to solve some goals produced by the ***NoConfusion*** step.
The last steps are all used to solve the problem of dealing with the injectivity and disjointness of constructors.

4.2 Focus on NoConfusion

Since the ***NoConfusion*** step is the trickier one, we show in more details what happens when the simplification mechanism encounters an equality between constructors. We will split this step in three parts: ***Pack***, ***NoConfusion*** and ***Unpack*** and will follow a simple example to explain it.

Let us consider the context $\Gamma = (A : \mathsf{Type})(n : \mathsf{nat})(x\ y : A)(v\ w : \mathsf{vect}\ A\ (\mathsf{S}\ n))$ and the goal:

$$\Gamma \vdash \mathsf{cons}\ x\ v = \mathsf{cons}\ y\ w \to \mathsf{vect}\ A\ n$$

where vect is the same inductive type of length-indexed lists as in the introduction. First, as we define NoConfusionPackage on a homogeneous type, we need to pack the values with their indices.

Pack `∀ (e : C t̄ = D ū), P e` becomes `∀ (e : (idx_t), C t̄) = (idx_u, D ū)), ind_pack_inv P e` where the type of `C t̄` is `I idx_t` and the type of `D ū` is `I idx_u`. In other words, $\overline{idx_t}$ and $\overline{idx_u}$ are the indices of $C\bar{t}$ and $D\bar{u}$.

This step requires K on the type of the indices of the inductive type. There is room for improvement by adapting the ideas of [4], as we underline in the last section of this paper.

The function ind_pack_inv is an opaque function which goes back to the original equality between the values in the inductive family; it will also serve as a marker that a ***NoConfusion*** step is in progress. This way, the equality does not get mixed in the goal and we can make sure to simplify ind_pack_inv properly once e is eliminated.

In our example, the index is in nat, which enjoys K. After this step the goal becomes:

$$\Gamma \vdash \forall (e : (\mathsf{S}\ n, \mathsf{cons}\ x\ v) = (\mathsf{S}\ n, \mathsf{cons}\ y\ w)),$$
$$\mathsf{ind\_pack\_inv}\ (\mathsf{vect}\ A\ n)\ e$$

NoConfusion `∀ (e : (idx_t), C t̄) = (idx_u, D ū)), P e` becomes:

- `∀ (e : t̄ = ū), P (noConf_inv e)` if C and D are the same constructor;
- solved if C and D are distinct constructors.

To implement this step, we use the NoConfusionPackage class that we are able to derive automatically (see section 3). The equality between $\bar{t}$ and $\bar{u}$ is, in general, an equality between telescopes, which will then be further simplified; when it is fully simplified, e and noConf_inv e will reduce to eq_refl.

Following the example, we get this goal:

$$\Gamma \vdash \forall (e : \mathsf{NoConfusion}\ (\mathsf{S}\ n, \mathsf{cons}\ x\ v)\ (\mathsf{S}\ n, \mathsf{cons}\ y\ w)),$$
$$\mathsf{ind\_pack\_inv}\ (\mathsf{vect}\ A\ n)\ (\mathsf{noConfusion\_inv}\ e)$$

As NoConfusion is applied to constructors, it is convertible to:

$$\Gamma \vdash \forall (e : (\mathsf{S}\ n, x, v) = (\mathsf{S}\ n, y, w)),$$
$$\mathsf{ind\_pack\_inv}\ (\mathsf{vect}\ A\ n)\ (\mathsf{noConfusion\_inv}\ e)$$

We end up with a telescopic equality at the head, which we can recursively simplify to unify its left- and right-hand sides. If the constructors were distinct, for instance cons and nil, then the application of NoConfusion would instead reduce to False, leading to a trivial absurdity.

Let us assume we have performed the unification of the equality at the head and get, for instance:

$$(A : \mathsf{Type})(m : nat)(y : nat)$$
$$\vdash \mathsf{ind\_pack\_inv}\ (\mathsf{vect}\ A\ m)(\mathsf{noConfusion\_inv}\ \mathsf{eq\_refl})$$

Unpack ind_pack_inv P eq_refl becomes P eq_refl

A simple step, closing the overall process of ***NoConfusion***.

In practice, we now rely on the fact that noConfusion_inv applied to eq_refl reduces to eq_refl to conclude and get the final goal after unification:

$$(A : \mathsf{Type})(m : nat)(y : nat) \vdash \mathsf{vect}\ A\ m$$

4.3 A simplification tactic

During the compilation of the splitting tree, when we split a variable, we end up with an equality of telescopes to simplify. We can run this simplification machinery on it, by using an inference mechanism which looks at the head of the current goal to choose which step to apply. The simplification part of Equations is modular, and can be easily plugged into an independent tactic which performs simplification on an arbitrary goal, either by choosing automatically a step to apply, or by letting the user specify which sequence of steps should apply.

For instance, with a goal such as `S n = S m → n = m`, one could use this simplification tactic and write `simplify ${→}` to end up with `m = m` as goal through a ***NoConfusion*** step and a ***Solution*** step. The current syntax, which is subject to change, uses → and ← for a ***Solution*** step to the right or to the left, – for ***Deletion*** and ${...} for ***NoConfusion***.

Since the ***NoConfusion*** step requires to simplify the equations it generates, it is natural to have a nested syntax for this step. It is also possible to let the module decide which step to apply by using one of the inference rules: ? to let it infer one step, and ∗ to let it fully simplify a telescopic equality.

5 Smart case

One of the core mechanisms of Equations is the ability to eliminate properly a dependently-typed variable. While a mechanical way to do it has been explained by Goguen et al., we might want to be more clever if we are producing an actual proof term to be used by Coq.

The goal of this section is to explain how, from a context $\Gamma_0(x : \mathsf{I}\ \bar{t})\Gamma_1$, we produce a Coq term which eliminates x. The result will look like the following:

```
match y as y' in I ū' return Δ → t̄ = ū' → T
with ...
end Δ eq_refl
```

where $(y : \mathsf{I}\ \bar{u})$ is some fresh variable that is introduced to generalize x and Δ is a list of variables from the initial context which need to be *cut*.

To simplify the presentation, we will be quite liberal with the notations in the following paragraphs, freely using a context as the telescope of its types, the telescope of its variables, or just a list of its variables.

From telescopes to elimination Let us consider some context $\Gamma_0(x : \mathsf{I}\ \bar{t})\Gamma_1$, where $\bar{t} : \bar{\tau}$ are the indices of the variable that we wish to eliminate. The straightforward way to do so, as explained with equation (1), is the following:

1. Introduce fresh binders for new indices, a new inductive value and an equality.
$$\Gamma_0(x : \mathsf{I}\ \overline{t})\Gamma_1(\overline{u} : \overline{\tau})(y : \mathsf{I}\ \overline{u})(e : (\overline{t}, x) = (\overline{u}, y))$$
2. Eliminate the fresh variable y using its dependent eliminator. This produces one branch for each constructor of the inductive type I. In the branch for constructor $\mathsf{C_i} : \forall \overline{T_i}, \mathsf{I}\ \overline{u_i}$, the indices $\overline{u}$ and the variable y are instantiated according to the arity of $\mathsf{C_i}$, and fresh constructors arguments $\overline{a}$ are introduced.
$$\Gamma_0(x : \mathsf{I}\ \overline{t})\Gamma_1(\overline{a} : \overline{T_i})(e : (\overline{t}, x) = (\overline{u_i}, \mathsf{C_i}\,\overline{a}))$$
3. Simplify the equality e, effectively unifying x with $\mathsf{C_i}\ \overline{a}$ if it succeeds positively, or directly solving the goal if it succeeds negatively.

The elimination of y is performed, in Coq, by the production of a `match` which would look like this:

```
match y as y' in I u' return (t, x) = (u', y) → T
with ...
end eq_refl
```

Therefore, we need to eliminate each equality one by one, while we could leverage the way Coq type-checks a `match` to remove some of these equalities. The goal is to produce a smaller and simpler term when possible, and especially to have less rewriting on types in our terms.

To do so, we will start from the context after generalization
$$\Gamma_0(x : \mathsf{I}\ \overline{t})\Gamma_1(\overline{u} : \overline{\tau})(y : \mathsf{I}\ \overline{u})(e : (\overline{t}, x) = (\overline{u}, y))$$
Recall that an equality of telescopes is equivalent to a telescope of equalities where each one can depend on the previous ones.
$$\Gamma_0(x : \mathsf{I}\ \overline{t})\Gamma_1(\overline{u} : \overline{\tau})(y : \mathsf{I}\ \overline{u})(e_1 : t_1 = u_1)e_2 \dots e_n$$
We will then perform a ***Solution*** step on some of the equalities e_j that will be chosen according to two criteria explained later, while maintaining this general shape for the telescope:
$$\Gamma(\overline{u} : \overline{\tau})(y : \mathsf{I}\ \overline{u})\Delta e$$
where e is a telescopic equality (possibly empty) which is some subset of the initial equality, and Δ is a list of variables from the initial context called the *cuts*. All these ***Solution*** steps are performed in OCaml directly on the telescope, and will keep producing telescopes which are equivalent to the first one.

Finally, from a telescope with this shape where we have maintained y fully generalized, we can eliminate y by producing a `match` which has the announced shape:

```
match y as y' in I u' return Δ → e → T
with ...
end Δ eq_refl
```

where the cuts are applied to the `match`. In each branch of the `match`, we still have to simplify the remaining telescopic equality to finish the unification.

There are two kinds of equalities that we will simplify early in this fashion: homogeneous equalities and equalities on variables on which nothing depends.

Homogeneous solutions Firstly, we consider in order each element of $(\overline{t}, x)$ to see if we can directly resolve the corresponding equality in e. We will do so for some term t_j in the telescope if the following are true:

- t_j is a variable;
- t_j did not appear anywhere in the previous indices nor in the parameters of the inductive type (linearity criterion);
- the type of t_j does not depend on an index that we cannot remove from the telescope (dependency criterion).

If all these criteria are true, then we can apply a ***Solution*** step to the selected equalities from left to right. Indeed, each equality will be homogeneous by the time we want to resolve it – thanks to the dependency criterion – and the term on the left will be a variable from the initial indices – thanks to the linearity criterion.

Note that it is always possible to apply a ***Solution*** step as long as one of the sides of the equality is a free variable, but we only want to do so on some selected equalities that will make our term simpler. In this case, as we are just manipulating a telescope, we can directly perform the ***Solution*** step on the telescope.

Recall that we start from a telescope with the following shape:
$$\Gamma(\overline{u} : \overline{\tau})(y : \mathsf{I}\ \overline{u})\Delta e_0(e : x = u_j)e_1$$
If we have chosen to solve the equality $(e : x = u_j)$, then we move x and anything that depends on it *after* y and its indices. We can perform this permutation because the linearity criterion ensures that x is a variable not in $(\overline{u}, y)$.
$$\Gamma'(\overline{u} : \overline{\tau})(y : \mathsf{I}\ \overline{u})(x : \tau_j)\Delta_x\Delta e_0(e : x = u_j)e_1$$
It is possible that x was already in Δ, for instance if it depended on a previously solved equality. In this case we just don't need to move it.

Then we move the equality $(e : x = u_j)$ right after the declaration of x. We can do so because the dependency criterion ensures that this equality is homogeneous, that is it does not depend on any equality in e_0.
$$\Gamma'(\overline{u} : \overline{\tau})(y : \mathsf{I}\ \overline{u})(x : \tau_j)(e : x = u_j)\Delta_x\Delta e_0 e_1$$
Finally, we can apply a ***Solution*** step on e.
$$\Gamma'(\overline{u} : \overline{\tau})(y : \mathsf{I}\ \overline{u})\Delta_x[x := u_j]\Delta e_0 e_1[x := u_j, e := \mathsf{eq\_refl}]$$
We end up again with a telescope which has the shape that we want to maintain, and can keep on solving the other equalities that we selected.

Clearing variables The second kind of equalities that we wish to solve early are equalities on variables in the telescope on which nothing depends. We will solve these equalities even if they are not homogeneous yet. Indeed, since nothing

depends on them, this will just be like removing the corresponding variable from the context, without introducing any complication.

Again, we start from a telescope with the following shape:

$$\Gamma(\overline{u} : \overline{\tau})(y : \mathsf{I}\ \overline{u})\Delta e_0(e : \mathrm{rew}\ e_0\ x = u_j)e_1$$

where rew e_0 x performs rewriting in the type of x through the equality of telescopes e_0. Indeed, this time, we allow e to be dependent on the previous equalities; the only condition is that x is a variable and nothing depends on x and e in e_1 or the (elided) goal.

Since nothing depends on x, we can move its declaration right before e.

$$\Gamma'(\overline{u} : \overline{\tau})(y : \mathsf{I}\ \overline{u})\Delta e_0(x : \tau_j)(e : \mathrm{rew}\ e_0\ x = u_j)e_1$$

If x was in Δ instead of Γ, it does not change anything.

Finally, we can perform a ***Solution*** step on e.

$$\Gamma'(\overline{u} : \overline{\tau})(y : \mathsf{I}\ \overline{u})\Delta e_0 e_1$$

This effectively clears x from the context at no cost. We end up again with a telescope which has the shape that we want to maintain, and can keep on clearing other variables in the same way if possible.

Implementation details First of all, the order in which we solve equalities by applying these two optimizations is not a pure left-to-right order as when Equations is building a splitting tree. As a consequence, it can happen that the context that we get after elimination of a variable is different in the compiled term and in the splitting tree. More precisely, since we have still performed the same unification steps, just in a different order, the two contexts will be a permutation of each other.

To recover a correct term, we compute a context mapping corresponding to the operations performed during the smart case and the simplification – which has the added benefit of making sure everything we do is well-typed – and match it to the context mapping from the splitting tree. We can deduce from these a permutation of context that we can apply in the term being produced.

We do not use the same strategy to produce the splitting tree and to compile a term for two main reasons:

- the implementation used to build the splitting tree is kept very close to the simple and mechanical way of eliminating a variable that was originally described;
- the second optimization relies not only on the current context, but also analyzes the current goal to determine if a variable has dependencies.

It is also necessary to underline a drawback of the first optimization. It is responsible for the presence of the cuts Δ, which were empty originally. These are variables whose type is "rewritten" definitionally, through the use of the `return` clause of the `match`. This is not an actual problem as long as the guard condition of Coq is able to track well enough these variables. On this subject, this work helped uncover a case where the guard condition was not liberal enough in tracking the subterm relationship. For more details, see the relevant pull-request[2].

6 A dependent elimination tactic

Since eliminating a dependently-typed variable is at the core of Equations, we can reuse its splitting mechanism to provide a dependent elimination tactic which allows a fine-grained control over the depth of the elimination, the names of any bound variable and the order of the clauses.

Consider that the current goal is $\Gamma \vdash \tau$, and we want to eliminate a variable x of type $\mathsf{I}\ \overline{t}$ from context Γ. We start with a list of patterns, corresponding to the different cases of this elimination. The user can provide such patterns, or the tactic can generate some default patterns as follows.

Each pattern is meant to correspond to one possible case of the elimination. In the simplest case, there is one pattern for each constructor of the type I, with one variable for each argument of the constructor. In some cases, there can be less patterns if some branches are impossible, or more if the user wants to do deep pattern-matching on some arguments of a constructor.

In any case, for each pattern p, the tactic produces a simple list of patterns:

- for each variable in context Γ which is not x, the associated pattern is the variable itself;
- for the variable x, the associated pattern is p.

To each list of patterns, we associate a right-hand side made of a program node := ?e with a hole ?e as a term.

We have a type and a list of clauses, this a problem that we can give to Equations, resulting in a term which has the correct type, and produces a subgoal for each hole that we put as right-hand sides of clauses. The user can then go on with proving each subgoal.

Below we provide a simple example using this dependent elimination tactic to prove the transitivity of the $\leq$ relation on the type fin n of sets $\{1 \ldots n\}$.

We define $\leq$ by fz $\leq i$ and $i \leq j \rightarrow$ fs $i \leq$ fs j.

```
Inductive fle : ∀ n, fin n → fin n → Set :=
| flez : ∀ n (j : fin (S n)), fle fz j
| fles : ∀ n (i j : fin (S n)), fle i j → fle (fs i) (fs j).
```

We will need a no-confusion principle for fin, which we derive automatically.

```
Derive NoConfusion for fin.
```

We could prove the transitivity of the relation fle by defining a recursive function with Equations, but here we will instead define a `Fixpoint` and use the dependent `elimination` tactic that we provide.

```
Fixpoint fle_trans {n : nat} {i j k : fin n}
  (p : fle i j) (q : fle j k) {struct p} : fle i k.
```

[2] https://github.com/coq/coq/pull/920

```
Proof.
```

We use the dependent `elimination` tactic to eliminate p, providing a pattern for each case. We could also let Equations generate names for the bound variables.

```
dependent elimination p as [flez n' j | fles n' i j p].
1: apply flez.
```

We know that q has type fle (fs _) k. Therefore, it cannot be flez and we must only provide one pattern for the single relevant branch.

```
dependent elimination q as [fles _ i' j' q].
```

The end of the proof is straightforward. We can check that this definition does not make use of any axiom, contrary to what we would obtain by using dependent `destruction`.

7 Recursion

Equations allows the user to define recursive functions either through the use of structural recursion, or by providing a well-founded relation for which an argument decreases, through the `by rec` t R annotation.

Structural or well-founded The most direct way to define a recursive function is to just reuse the name of the function in any right-hand side of a clause. In this case, the user relies on Coq's guard condition to check that the definition is correct. While the implementation of the guard condition has been adapted over the years to try and allow as many safe cases as possible, it is still obviously an approximation and may fail in some legitimate cases. This is aggravated by the fact that Equations uses a lot of rewriting, making the job of the guard condition checker that much more difficult. In many cases everything works as expected, but sometimes it (apparently) diverges because it must unfold definitions or it fails to track the subterm relation correctly in the term due to rewritings (applications of J) or *cuts*. Indeed a problematic case appears when the subterm is abtracted in branches of a match, as mentionned earlier. Equations provides an automatic derivation of the well-foundedness of the Subterm relation to handle these cases where the guardness check fails. It can also be the case that the recursion is simply not structural and the user wants to use well-founded recursion.

Well-founded functions are defined as usual, except Equations will afterwards ask the user to prove some obligations about the well-foundedness of the relation, and that the arguments decrease according to the given order for each recursive call, like Program or Function.

Unfolding When a well-founded recursive function f is defined, Equations also builds an *unfolded* version of the function called f_unfold, whose equations are the same as f, with any recursive call replaced by a call to f. Hence, f_unfold represents the 1-unfolding of f. Equations then proves automatically, by following the structure of the splitting tree, that f and f_unfold coincide at any point. The content of f_unfold is easier to manipulate than f because the "recursive" calls do not need to include the proofs that the recursive arguments decrease and it does not include an application of the well-founded recursion combinator: i.e. it is really non-recursive. The unfolding lemma for any function f has the following type, where f is directly an application of the well-founded recursion combinator FixWf:

$$\forall \Delta, \mathrm{f}\ \overline{\Delta} = \mathrm{f\_unfold}\ \overline{\Delta} \tag{4}$$

Using this lemma, we can also express cleanly the elimination principle of f, abstracting away from the proofs used to prove its termination. See [18] for details on this construction.

Accessibility is proof-irrelevant Using the folklore result that constructive accessibility as defined in Coq is proof-irrelevant, we can prove equation 4 using only the functional extensionality axiom (this is required to show that Acc is proof-irrelevant already). Hence, our proofs of unfolding for well-founded recursive definitions must rely on the functional extensionality axiom. This is the only axiom used by Equations now. Note that in a system with *definitional* proof irrelevance, this axiom would disappear. Finally, the presence of this axiom is not problematic in practice, as anyway the unfolding behavior of well-founded fixpoints on open terms, even assuming a closed well-foundedness proof, makes it difficult to handle during proofs and users tend to resort to unfolding lemmas instead. Note also that the unfolding lemma and the elimination principle allow reasoning on a recursive function even if its termination proof has not been provided, i.e. if it was itself admitted as an axiom.

8 Evaluation and examples

As an example of a successful use of well-founded recursion with Equations, we formalized the normalization of Leivant's predicative System F through hereditary substitution. The details can be found in [11].

Equations allows us to properly separate the computational content of the function, and the logical parts such as the proof of its termination and the proof of its correctness. As such, the extracted version of the normalization function ends up being very close to what we would have written naturally in OCaml. The function, written in the source language of Equations, is also easy to read and understand, with all the logical components being proved afterwards through the use of Coq's obligations.

8.1 A reflexive tactic using dependent types

To showcase the fact that the system allows to define *computational* functions on dependent types, we present here excerpts of the development of a reflexive tactic for solving polynomial equations. This example is derived from a solution provided by a student to a Coq project assignment. The original solution used an earlier version of Equations already but not all its features which we present here.

The goal of this exercise was to develop a reflexive tactic for proving boolean tautologies, by reflecting them into the set of multivariate polynomials over $\mathbb{Z}$. The complete annotated proof script is available on the web[3], among other examples, we encourage readers to follow the rest of the section by running it, we extracted this section of the article from that file.

8.1.1 Multivariate polynomials

Using an indexed inductive type, we ensure that polynomials of $\mathbb{Z}[(X_i)_{i\in\mathbb{N}}]$ have a unique representation. The first index indicates that the polynom is null. The second index gives the number of free variables.

```
Inductive poly : bool → nat → Type :=
| poly_z : poly true O
| poly_c (z : Z) : IsNZ z → poly false O
| poly_l {n b} (Q : poly b n) : poly b (S n)
| poly_s {n b} (P : poly b n) (Q : poly false (S n)) :
    poly false (S n).
```

- poly_z represents the null polynomial.
- poly_c c represents the constant polynomial c where c is non-zero (i.e. has a proof of IsNZ c).
- poly_l n Q represents the injection of Q, a polynomial on n variables, as a polynomial on n+1 variables.
- Finally, poly_s P Q : poly _ (S n) represents $P + X_n * Q$ where P cannot mention the variable X_n but Q can mention the variables up to and including X_n, and the multiplication is not trivial as Q is non-null.

These indices enforce a canonical representation by ordering the multiplications of the variables. A similar encoding is actually used in the `ring` tactic of Coq.

```
Derive Signature NoConfusion Subterm for poly.
```

In addition to the usual eliminators of the inductive type generated by Coq, we automatically derive a few constructions on this poly datatype, and the mono datatype that follows, that will be used by the Equations command:

- Its Signature: as described earlier (§3), this is the packing of a polynomial with its two indices, a boolean and a natural number in this case.
- Its NoConfusion property used to simplify equalities between constructors of the poly type (equation 3).
- Finally, its Subterm relation, to be used when performing well-founded recursion on poly.

8.1.2 Monomials

Monomials represent parts of polynoms, and one can compute the coefficient constant by which each monomial is multiplied in a given polynom. Again the index of a mono gives the number of its free variables.

```
Inductive mono : nat → Type :=
| mono_z : mono O
| mono_l : ∀ {n}, mono n → mono (S n)
| mono_s : ∀ {n}, mono (S n) → mono (S n).
```

Our first interesting definition computes the coefficient in Z by which a monomial m is multiplied in a polynomial p.

```
Equations get_coef {n} (m : mono n) {b} (p : poly b n) : Z
  := get_coef m p by rec (pack m) mono_subterm :=
get_coef mono_z poly_z := 0%Z;
get_coef mono_z (poly_c z _) := z;
get_coef (mono_l m) (poly_l p) := get_coef m p;
get_coef (mono_l m) (poly_s p _) := get_coef m p;
get_coef (mono_s m) (poly_l _) := 0%Z;
get_coef (mono_s m) (poly_s p1 p2) := get_coef m p2.
```

The definition can be done using either the usual structural recursion of Coq or well-founded recursion. If we use structural recursion however, the guardness check will not be able to verify the automatically generated proof that the function respects its graph, as it involves too much rewriting due to dependent pattern-matching. We could prove it using a dependent induction instead of using the raw fixpoint combinator as the recursion is on direct subterms of the monomial, but in general it could be arbitrarily complicated, so we present a version allowing deep pattern-matching and recursion. Note that this means we lose the definitional behavior of get_coef during proofs on open terms, but this can advantageously be replaced using explicit `rewrite` calls, providing much more control over simplification than the reduction tactics, especially in presence of recursive functions. The get_coef function still uses no axioms, so it can be used to compute as part of a reflexive tactic for example.

We want to do recursion on the (dependent) m : mono n argument, using the derived mono_subterm relation, which expects an element in the signature of mono, { n : nat & mono n }, so we use pack m to lift m into its signature type (pack is just an abbreviation for the signature_pack overloaded constant defined in §3).

The rest of the definition is standard: to fetch a monomial coefficient, we simultaneously pattern-match on the monomial and polynomial. Note that many cases are impossible due to the invariants enforced in poly and mono. For example mono_z can only match polynomials built from poly_z or poly_c, etc.

8.1.3 Two detailed proofs

The monomial decomposition is actually a complete characterization of a polynomial: two polynomials with the same coefficients for every monomial are the same.

```
Theorem get_coef_eq {n} b1 b2
  (p1 : poly b1 n) (p2 : poly b2 n) :
  (∀ (m : mono n), get_coef m p1 = get_coef m p2) →
  (b1 ; p1) = (b2 ; p2) :> { null : _ & poly null n}.
Proof with (simp get_coef in *; auto).
```

[3] http://mattam82.github.io/Coq-Equations/examples

Throughout the proof, we use the `simp` tactic defined by Equations which is a wrapper around `autorewrite` using the hint database associated to the constant get_coef: the database contains the defining equations of get_coef as rewrite rules that can be used to simplify calls to get_coef in the goal.

```
intros Hcoef.
induction p1 as [ | z Hz | n b p1 | n b p1 IHp q1 IHq ]
  in b2, p2, Hcoef ⊢ *;
[dependent elimination p2 as [poly_z | poly_c z i] |
 dependent elimination p2 as [poly_z | poly_c z i] |
 dependent elimination p2 as
     [poly_l n b' p2 | poly_s n b' p2 q2] ..].
```

We first do an induction on *p1* and then eliminate (dependently) *p2*, the first two branches need to consider variable-closed *p2*s while the next two branches have *p2* : poly _ (S *n*), hence the poly_l and poly_s patterns. The elided rest of the tactic solves simple subgoals.

We now focus on the case for poly_l on both sides. After some simplifications of the induction hypothesis using the *Hcoef* hypothesis, we get to the following goal:

```
(b, b' : bool) (n : nat) (p1 : poly b n) (p2 : poly b' n)
IHp1 : (b; p1) = (b'; p2)
============================
(b; poly_l p1) = (b'; poly_l p2)
```

The *IHp1* hypothesis, as a general equality between dependent pairs can again be eliminated dependently to substitute *b'* by *b* and *p2* by *p1* simultaneously, using `dependent elimination IHp1 as [eq_refl]`, leaving us with a trivial subgoal. The next step is to give an evaluation semantics to polynomials. We program `eval p v` where *v* is a valuation in Z for all the variables in *p* : poly _ *n*.

```
Equations eval {n} {b} (p : poly b n) (v : Vector.t Z n) : Z
  := eval p v by rec (pack p) poly_subterm :=
  eval poly_z nil := 0%Z;
  eval (poly_c z _) nil := z;
  eval (poly_l p) (cons _ _ xs) := eval p xs;
  eval (poly_s p1 p2) (cons y _ ys) :=
    (eval p1 ys + y × eval p2 (cons y ys))%Z.
```

Again we are using well-founded recursion on *p* using the subterm order. It is quite clear that two equal polynomials should have the same value for any valuation. To show this, we first need to prove that evaluating a null polynomial always computes to 0, whichever valuation is used. This is a typical case where the proof directly follows the definition of `eval`. Instead of redoing the same case splits and induction that the function performs, we can directly appeal to its elimination principle using the `funelim` tactic.

```
Lemma poly_z_eval {n} (p : poly true n) v : eval p v = 0%Z.
Proof.
  funelim (eval p v); [ reflexivity | assumption ].
Qed.
```

This leaves us with two goals as the true index in *p* implies that the poly_c and poly_s clauses do not need to be considered. We have to show 0 = 0 for the case *p* = poly_z and `eval q v` = 0 for the poly_l recursive constructor, in which case the conclusion directly follows from the induction hypothesis correspondinng to the recursive call. The second subgoal is hence discharged with an `assumption` call.

Using Equations, we can define more complex definitions on our polynomials, ultimately equipping poly with the structure of a ring. We stop our presentation here, but the reader can refer to the online version to finish the example. It culminates in a reflexive tactic that can solve tautologies in Heyting or Classical boolean algebras.

9 Related and Future Work

Cockx and Devriese [5] present an improvement on the simplification of unification constraints for indexed datatypes avoiding more uses of K that we are hoping to integrate in our algorithm. We reproduced its proof in Coq and are looking at ways to integrate it during simplification.

The technique of small inversions [15] is an alternative way to implement dependent eliminations, that is restricted to linear cases.

The equation compiler of Lean [2] is similar to our system. As mentioned in the introduction, pattern-matching compilation is simplified by using definitional proof-irrelevance. It also supports well-founded recursion using a fixpoint combinator and inference of the well-founded relation (op. cit. §8.4), but does not derive the corresponding elimination principle, only the unfolding equation.

The Function package [3] of Coq also provides support for deriving an eliminator from a well-founded definition and also proves the completeness of the graph that we currently lack. It is also clever about handling overlapping or default branches in pattern-matchings, providing a graph that corresponds more closely to the shape of the definition entered by the user. We leave to future work a refinement of the splitting tree structure to handle a similar optimization when typing constraints allow it. The main advantage of Equations is that it allows definitions by dependent pattern-matching and recursion schemes that Function cannot handle.

Conclusion

This article presents an enhanced design and implementation of the prototype described in [18]. While the interface has only been extended to support `where` clauses, the most crucial changes are in the way dependent pattern-matching is compiled, moving from heterogeneous equality to equalities of telescopes, and implementation-wise the development of a mostly independent optimized simplifier that can be reused for implementing a dependent elimination tactic. We hope to have demonstrated its usefulness on a medium-sized example.

References

[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *CoqPL - The Third International Workshop on Coq for Programming Languages*. Paris, France. http://conf.researchr.org/event/CoqPL-2017/main-certicoq-a-verified-compiler-for-coq

[2] Jeremy Avigad, Gabriel Ebner, and Sebastian Ullrich. 2017. The Lean Reference Manual, release 3.3.0. (October 2017). Available at https://leanprover.github.io/reference/lean_reference.pdf.

[3] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. 2006. Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant. *Functional and Logic Programming* (2006), 114–129. https://doi.org/10.1007/11737414_9

[4] Jesper Cockx. 2017. *Dependent Pattern Matching and Proof-Relevant Unification*. Ph.D. Dissertation. Katholieke Universiteit Leuven, Belgium. https://lirias.kuleuven.be/handle/123456789/583556

[5] Jesper Cockx and Dominique Devriese. 2017. Lifting proof-relevant unification to higher dimensions. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 173–181. https://doi.org/10.1145/3018610.3018612

[6] Jesper Cockx, Dominique Devriese, and Frank Piessens. 2014. Pattern matching without K. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 257–268. https://doi.org/10.1145/2628136.2628139

[7] Thierry Coquand. 1992. Pattern Matching with Dependent Types. (1992). http://www.cs.chalmers.se/~coquand/pattern.ps Proceedings of the Workshop on Logical Frameworks.

[8] Healfdene Goguen, Conor McBride, and James McKinna. 2006. Eliminating Dependent Pattern Matching. In *Essays Dedicated to Joseph A. Goguen (Lecture Notes in Computer Science)*, Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer (Eds.), Vol. 4060. Springer, 521–540. http://www.cs.st-andrews.ac.uk/~james/RESEARCH/pattern-elimination-final.pdf

[9] Martin Hofmann and Thomas Streicher. 1994. A Groupoid Model Refutes Uniqueness of Identity Proofs. In *LICS*. IEEE Computer Society, 208–212. http://www.tcs.informatik.uni-muenchen.de/~mhofmann/SH.dvi.gz

[10] Peter LeFanu Lumsdaine. 2010. Weak omega-categories from intensional type theory. *Logical Methods in Computer Science* 6, 3 (2010).

[11] Cyprien Mangin and Matthieu Sozeau. 2015. Equations for Hereditary Substitution in Leivant's Predicative System F: A Case Study. In *Proceedings Tenth International Workshop on Logical Frameworks and Meta Languages: Theory and Practice (EPTCS)*, Vol. 185. https://doi.org/10.4204/EPTCS.185 LFMTP'15.

[12] Conor McBride. 1999. *Dependently Typed Functional Programs and Their Proofs*. Ph.D. Dissertation. University of Edinburgh. http://citeseer.ist.psu.edu/mcbride99dependently.html

[13] Conor McBride. 2000. Elimination with a Motive. In *TYPES (Lecture Notes in Computer Science)*, Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack (Eds.), Vol. 2277. Springer, 197–216.

[14] Conor McBride, Healfdene Goguen, and James McKinna. 2004. A Few Constructions on Constructors. *Types for Proofs and Programs* (2004), 186–200. https://doi.org/10.1007/11617990_12

[15] Jean-François Monin and Xiaomu Shi. 2013. *Handcrafted Inversions Made Operational on Operational Semantics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 338–353. https://doi.org/10.1007/978-3-642-39634-2_25

[16] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden. http://www.cs.chalmers.se/~ulfn/papers/thesis.html

[17] Álvaro Pelayo and Michael A. Warren. 2012. Homotopy type theory and Voevodsky's univalent foundations. (10 2012). arXiv:1210.5658 http://arxiv.org/abs/1210.5658

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

[19] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations for Mathematics*. Institute for Advanced Study. http://homotopytypetheory.org/book

[20] Benno van den Berg and Richard Garner. 2011. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society* 102, 2 (2011), 370–394. https://doi.org/10.1112/plms/pdq026