

Equations: A Dependent Pattern-Matching Compiler

Matthieu Sozeau

Harvard University
`mattam@eecs.harvard.edu`

Abstract. We present a compiler for definitions made by pattern matching on inductive families in the COQ system. It allows to write structured, recursive dependently-typed functions as a set of equations, automatically find their realization in the core type theory and generate proofs to ease reasoning on them. It provides a complete package to define and reason on functions in the proof assistant, substantially reducing the boilerplate code and proofs one usually has to write, also hiding the intricacies related to the use of dependent types and complex recursion schemes.

1 Introduction

In this paper, we present a new tool to define and reason on functions manipulating inductive families in the COQ system. At the core of the system is a compiler for dependent pattern-matching definitions given as a set of *equations* into vanilla COQ terms, inspired by the work of Goguen *et al.* [1]. Our system also incorporates with-clauses (as in EPIGRAM or AGDA) that can be used to add a pattern on the left-hand side for further refinement and supports structural and well-founded recursion on inductive families using a purely logical and efficient scheme.

The system provides proofs of the equations that can be used as rewrite rules to reason on calls to the function. It also automatically generates the inductive graph of the function and a proof that the function respects it, giving a useful elimination principle for it.

EQUATIONS¹ is implemented as an elaboration into the core COQ type theory, allowing the smallest trusted code base possible and ensuring the correctness of the compilation at each use. The whole system makes heavy use of type classes and the high-level tactic language of COQ for greater genericity and extensibility.

The paper is organized as follows: first, we present an implementation of a dependent pattern-matching compiler (§2) supporting with clauses and efficient recursion on inductive families (§3). Then, we show how we can derive support proofs and in particular a powerful elimination principle directly from the structure of our programs (§4). We finally discuss related work in section 5 and conclude (§6).

¹ Available at <http://mattam.org/research/coq/equations.en.html>

2 Dependent pattern-matching compilation redux

The idea of writing pattern-matching equations over inductive families goes back to Coquand [2]. He introduced the idea of checking that a set of equations formed an exhaustive *covering* of a signature. From this covering one can build an efficient case tree in the standard way [3].

The interesting addition of dependent pattern-matching over simply-typed pattern-matching is the fact that some constructors need not be considered because the type of the object being matched upon guarantees that it could not have been built with them. Moreover, as each constructor refines the indices of a filtered object and as we are considering equations that can have multiple patterns, refinement may have effects on the values or types of other matched objects. This means that each constructor adds static information to the problem, and this process can be used ad libitum, as exemplified by the definition of `diag` below:

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

We pattern match on a square matrix of size n by n and compute its diagonal. Only two cases need to be considered: either the matrix is empty and so is its diagonal, or the matrix has $n + 1$ rows made of vectors of size $n + 1$ and we can extract the element at the top left of the matrix and build the rest of the diagonal recursively.

Internal vs. external approaches There exist two main approaches to adding dependent pattern matching to a dependent type theory. One is to bake in the high-level pattern matching construct and make the associated coverage checking and unification procedure part of the core system. This is essentially a shallow approach: one works directly in the metalanguage of the system's implementation and avoids building witnesses for the covering and unification. The disadvantages of the external approach are that it makes the trusted code base larger and limits the extensibility of the system: adding a new pattern matching construct like with-clauses requires to modify the kernel's code. AGDA implements pattern-matching this way, and there is a proposal to extend COQ in a similar way [4].

The internal approach takes a different path. In this case we use the type theory itself to explain why a definition is correct, essentially building a witness of the covering in terms of the simpler existing constructs on inductive families. This is the path chosen by [1], and the way EPIGRAM implements pattern-matching. One advantage is that the compiler needs not to be trusted: it elaborates a program that can be checked independently in the core type theory. By taking an elaboration viewpoint, it is also much easier to extend the system with new features that can also be compiled away to the core type theory. Our mantra (after McBride) is that type theory is enough to explain high-level programming constructs.

Our implementation closely follows the scheme from [1], its originality comes mainly from a number of design choices that we will explain in detail. We will not present here the whole formal development of pattern-matching compilation as is done in [1] but we will introduce the main structures necessary to describe our contributions.

The compilation process starts from a signature and a set of clauses given by the user, constructed from the grammar given in figure 1.

term, type	$t, \tau ::= x \mid \lambda x : \tau, t \mid \Pi x : \tau, \tau' \mid \dots$
binding	$d ::= (x : \tau) \mid (x := t : \tau)$
context	$\Gamma, \Delta ::= \vec{d}$
program	$prog ::= f \Gamma : \tau := \vec{c}$
user clause	$c ::= f \vec{up} n$
user pattern	$up ::= x \mid C \vec{up} \mid ?(t)$
user node	$n ::= := t \mid :=! x \mid \text{with } t := \{ \vec{c} \}$

Fig. 1. Definitions and user clauses

A program is given as a tuple of a (globally fresh) identifier, a signature and a set of user clauses. The signature is simply a list of bindings and a result type. The purposed type of the function f is then $\Pi \Gamma, \tau$. Each user clause comprises a set of patterns that will match the bindings Γ and a right hand side which can either be a simple term (program node), an empty node indicating that the type of variable x is uninhabited or a refinement node adding a pattern to the problem, scrutinizing the value of t .

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. 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 consider inductive families to be defined in a (elided) global context by an arity $\mathbf{l} : \Pi \Delta, s$ and constructors $\mathbf{l}_i : \Pi \Gamma_i, \mathbf{l} \vec{t}$. 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.

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.

context map $c ::= \Delta \vdash \vec{p} : \Gamma$
pattern $p ::= x \mid \mathbf{C} \vec{p} \mid ?(t)$
splitting $spl ::= \mathbf{Split}(c, x, (spl?)^n) \mid \mathbf{Compute}(c, rhs)$
node $rhs ::= \mathbf{Program}(t) \mid \mathbf{Refine}(t, c, \ell, spl)$
label $\ell ::= \epsilon \mid \ell.n \quad (n \in \mathbb{N})$

Fig. 2. Context mappings and splitting trees

The search for a covering works by gradually refining a *programming problem* $\Delta \vdash \vec{p} : \Gamma$ 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 $\Gamma \vdash \vec{p} : \Gamma$, i.e. the identity substitution on Γ and the list of user clauses. For f with signature $\Pi \Delta, \tau$ we define the f -computation type $f_{\text{comp}} \Delta := \tau$ and consider the type of the function to be $\Pi \Delta, f_{\text{comp}} \Delta$ from now on. We use this computation type during compilation to precisely keep track of recursive calls and to interface with tactics. A splitting can either be:

- A $\mathbf{Split}(\Delta \vdash \vec{p} : \Gamma, x, (s?)^n)$ 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.
- A $\mathbf{Compute}(\Delta \vdash \vec{p} : \Gamma, rhs)$ node, where the right hand side can be either:
 - A $\mathbf{Program}(t)$ node denoting a leaf program t of type $f.l_{\text{comp}} \vec{p}$ in Δ .
 - A $\mathbf{Refine}(t, c', \ell, s)$ node corresponding to a with rule introducing a pattern for t with s covering the new problem c' . The label ℓ uniquely identifies the node and will be used to define auxiliary definitions.

$\text{MATCH}(x, p) ::= \uparrow \{x := p\}$	
$\text{MATCH}(\mathbf{C} \vec{p}, \mathbf{C} \vec{q}) ::= \text{MATCH}(\vec{p}, \vec{q})$	
$\text{MATCH}(\mathbf{C} -, \mathbf{D} -) ::= \Downarrow$	$\uparrow s \cup \uparrow s' ::= \uparrow (s \cup s')$
$\text{MATCH}(\mathbf{C} \vec{p}, y) ::= \Rightarrow \{y\}$	$\Rightarrow vs \cup \Rightarrow vs' ::= \Rightarrow (vs \cup vs')$
$\text{MATCH}(?(t), -) ::= \uparrow \emptyset$	$\Downarrow \cup - \mid - \cup \Downarrow ::= \Downarrow$
	$\Rightarrow vs \cup - ::= \Rightarrow vs$
	$- \cup \Rightarrow vs ::= \Rightarrow vs$
$\text{MATCH}(\epsilon, \epsilon) ::= \uparrow \epsilon$	
$\text{MATCH}(p_0; \vec{p}, q_0; \vec{q}) ::= \text{MATCH}(p_0, q_0) \cup \text{MATCH}(\vec{p}, \vec{q})$	

Fig. 3. Matching patterns

Recursively, we will try to match the user patterns of each clause with the current problem $\Delta \vdash \vec{p} : \Gamma$. Matching patterns \vec{q} from the a user clause and patterns \vec{p} from the current programming problem can either fail (\Downarrow), succeed ($\uparrow s$) returning a variable substitution s from \vec{q} to \vec{p} or get stuck ($\Rightarrow vs$) returning a set of variables from \vec{p} that needs further splitting to match the user patterns in \vec{q} .

- If the clause does not match a particular problem we try the next one, if there are no clauses left we have a non-exhaustive pattern-matching.
- If the problem is stuck on the clause, we try to recursively find a splitting after refining a stuck variable x , creating subproblems that correspond to an instantiation of the variable with each possible constructor. We build a $\text{Split}(c, x, s)$ node from the resulting set of splittings if all succeed, or try the next stuck variable.
- If the clause matches we get back a substitution from the user clause variables to Δ , so we can typecheck right-hand side terms in environment Δ . We look at the right-hand side and decide:
 - If it is a program user node, we simply typecheck the program and build a $\text{Program}(t)$ node.
 - If it is an empty node ($:=! x$), we refine x and check that this produces no subproblems, building a Split node.
 - If it is a with node ($\Leftarrow t \Rightarrow \{\bar{c}\}$), we typecheck t in Δ finding its type τ_Δ . We then strengthen the context Δ for t , giving us the minimal context Δ^t to typecheck t and the remaining context Δ_t . This strengthening is in fact a context mapping $\Delta^t, x_t : \tau_\Delta, \Delta_t \vdash str : \Delta, x_t : \tau_\Delta$. We can now abstract t from the remaining context to get a new context: $\Delta_{\ell.n} \triangleq \Delta^t, x_t : \tau_\Delta, \Delta_t[t/x_t]$. We check that this context is well-typed after the abstraction, which might not be the case. We also define an abstracted type for the subprogram $\text{f.l.n}_{\text{comp}} \Delta_{\ell.n} := \text{f.l}_{\text{comp}} \bar{p}[t/x_t]$ and search for a covering of the identity substitution of $\Delta_{\ell.n}$ using updated user clauses \bar{c} . The user clauses are actually transformed to match the strengthening: each c_i must be of the form $\bar{p}_i p_i^x$ where \bar{p}_i matches \bar{p} . The matching gives us a substitution from the variables of \bar{p} , the patterns at the with node, to new user patterns. We can easily make new user clauses matching the strengthened context $\Delta_{\ell.n}$ by associating to each variable of $\Delta_{\ell.n}$ its associated user pattern and using p_i^x for the new pattern. The result of the covering will then be a splitting s for the problem $c' = \text{idsubst}(\Delta_{\ell.n})$ from which we can build a $\text{Refine}(t, c', \ell.n, s)$ node with a fresh n . Compiling this term will give us a term of type $\Pi \Delta_{\ell.n}, \text{f.l.n}_{\text{comp}} \bar{\Delta}_{\ell.n}$. We can apply this term to $\bar{\Delta}^t, t, \bar{\Delta}_t$ to recover a term of type $\text{f.l}_{\text{comp}} \bar{p}$ in the original Δ context, providing a witness for the initial $\Delta \vdash \bar{p} : \Gamma$ problem. Consider for example the following definition:

```

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

```

When interpreting the **with** node, the patterns of the inner clauses are transformed to match the variables $A a l p$ bound at the **with** node and their additional pattern for $p a$.

The compiled term built from this covering will have a type convertible with:

$$\Pi A a l p (b : \text{bool}), \text{filter}_{\text{comp}} A (\text{cons } a l) p$$

We can then instantiate b with $p a$ to build a term in the initial A, a, l, p context.

This is a basic overview of the algorithm for type-checking pattern-matching definitions as described in [5], except for the treatment of inaccessible patterns we omitted for brevity. In our case however we not only check that pattern-matchings are well-formed, we also produce witnesses for this compilation in the core language, following [1]. Now that we have compiled the program to a simplified splitting tree, we just need to construct a mapping from splittings to COQ terms. We already explained how $\text{Refine}(c, x, \ell, s)$ nodes are compiled, and $\text{Program}(t)$ nodes are trivially compiled, so we just need to map Split nodes.

2.1 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, and use these 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 [6,7]. The problem we face is to eliminate an object x of type $\mathbb{1} \vec{t}$ 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 $\Delta (x' : \mathbb{1} \overline{\Delta})$ and a set of equations asserting that x' is equal to x , giving us a new, equivalent goal:

$$\Delta, x' : \mathbb{1} \overline{\Delta}, \Gamma \vdash \overline{\overline{\Delta}_i \simeq \vec{t}_i} \rightarrow x \simeq x' \rightarrow \tau$$

Note that the equations relate terms that may be in different types due to the fresh indices, hence we use heterogeneous equality \simeq to relate them. We can apply the standard eliminator for $\mathbb{1}$ on x' in this goal to get subgoals corresponding to all its constructors, all starting with a set of equations relating the indices t of the original instance to the indices of the constructor. We use a recursive tactic to simplify these equalities, solving the impossible cases automatically. Its completeness is asserted in [1]: at the end of specialization we get refined goals where the initial x has been substituted by the allowed constructors only.

Our tactic relies on a set of combinators for simplifying equations in the theory of constructors, most of which are just rephrasings of the substitution principles for Leibniz and heterogeneous equality. The only interesting bit is a simplifier for equalities between constructors. We need a tactic that can simplify any equality $\mathbb{C} \vec{t} = \mathbb{D} \vec{u}$, either giving us equalities between arguments \vec{t} and

\vec{u} that can be further simplified or deriving a contradiction if \mathbf{C} is different from \mathbf{D} . McBride *et al.* [7] describe a generic method to derive such an eliminator that we adapted to COQ. For any (computational) inductive type $\mathbf{l} : \Pi \Gamma, \mathbf{Type}$, we can derive a transformer $\mathbf{NoConfusion}_{\mathbf{l}} : \Pi \Gamma (P : \mathbf{Type}), \mathbf{l} \bar{\Gamma} \rightarrow \mathbf{l} \bar{\Gamma} \rightarrow \mathbf{Type}$ that describes how to simplify the goal P under the assumption that the two instances of $\mathbf{l} \bar{\Gamma}$ are equal. E.g., for natural numbers we define:

```

Equations NoConfusion_nat (P : Type) (x y : nat) : Type :=
  NoConfusion_nat P O O := P → P ;
  NoConfusion_nat P (S n) (S m) := (n = m → P) → P ;
  NoConfusion_nat P _ _ := P.

```

Suppose we have a goal P and a proof of $\mathbf{NoConfusion_nat} P x y$ for x and y in constructor form supposing $x = y$. The proof will always unfold to an implication ending in P , so we can apply it to our goal. Depending on the form of x and y , we will make the goal progress in different ways. If x and y are both \mathbf{O} , then we are left to prove the same goal unchanged, the equality is trivial ($P \rightarrow P$). If x and y are both of the form \mathbf{S}_- then we are left with a proof of the goal under the additional hypothesis that the arguments are equal ($(n = m \rightarrow P) \rightarrow P$). Finally, in all other cases, the goal is directly discharged, as we have a witness of P by contradiction of the equality of n and m .

We define a new type class [8] to register *NoConfusion* proofs for each type. Instances can be automatically derived for any computational inductive family. We can then build a generic tactic to simplify any equality hypothesis on a registered type using this construction, which subsumes the standard `discriminate` and `injection` tactics.

Dealing with K There is one little twist in our simplifier, due to the fact that COQ does not support the principle of “Uniqueness of Identity Proofs”, also referred to as Streicher’s K axiom [9], which is necessary to compile dependent pattern-matchings:

```

Axiom UIP_refl :  $\forall (U : \mathbf{Type}) (x : U) (p : x = x), p = \mathbf{eq\_refl}$ 

```

This principle allows us to simplify a goal depending on a proof p of $x = x$ by substituting the sole constructor `eq_refl` for p . As we are outside the kernel, we can easily make use of this axiom to do the simplifications, but this means that some of our definitions will not be able to reduce to their expected normal forms: they are not closed in the empty context anymore. We will tame this problem by providing the defining equations as rewrite rules once a function is accepted, making use of the axiom again to prove these.

It is notorious that using rewriting instead of the raw system reduction during proofs is much more robust and lends itself very well to automation. Hence we only lose the ability to compute with these definitions inside COQ itself, for example as part of reflexive tactics. At least two proposed extensions to COQ allow to derive this principle without any axioms: an extension to make dependent pattern-matching more powerful with respect to indices [4] and the addition of proof-irrelevance. Having them would make EQUATIONS only more

useful. Note that extracted terms do not suffer from this fact as propositions like equality are erased.

3 Recursion

We now turn to the treatment of recursive definitions. A notorious problem with the COQ system is that it uses a syntactic check to verify that recursive calls are well-formed. Only structurally recursive functions making recursive calls on a single designated argument are allowed. The syntactic criterion is very restrictive, inherently non-modular and a major source of bugs in the core type checker. Its syntactic nature also precludes the use of some program transformations, for example uses of abstraction might turn a guarded program into an unguarded one. To avoid these pitfalls, we can use the same principle as for pattern-matching and *explain* the recursive structure of our programs using type theory itself.

To do so, we will use an elimination principle on the datatype we want to recurse on, that will give us a way to make recursive calls on any subterm. Instead of a syntactic notion of structural recursion, we will now use a logical one, which is compatible with the rest of the logical transformations happening during compilation.

3.1 The `Below` way

Goguen *et al.* [1] give a way to elaborate recursive definitions by building a memoizing structure. For any inductive type $I : III, Type$, we define a new type `BelowI` that captures all the recursive subterms of a given term, applied to an arity. For natural numbers, we define `Below_nat` as follows:

```
Equations Below_nat (P : nat → Type) (n : nat) : Type :=
  Below_nat P O := unit ;
  Below_nat P (S n) := (P n × Below_nat P n)%type.
```

The `Below_nat` definition uses the built-in structural recursion to build a tuple of all the recursive subterms of a number, applied to an arbitrary arity P . We can build this tuple for any $n : nat$ given a functional *step* that builds a $P n$ if we have P for all the strict subterms of n , and hence derive an eliminator:

```
Definition rec_nat (P : nat → Type)
  (step : II n : nat, Below_nat P n → P n) (n : nat) : P n :=
  step n (below_nat P step n).
```

Now suppose we want to define a function by recursion on $n : nat$. We can simply apply this recursor to get an additional `Below_nat P n` hypothesis in our context. If we then refine n , this `Below_nat P n` hypothesis will unfold at the same time to a tuple of $P n'$ for every recursive subterm n' of n . These hypotheses form the allowed recursive calls of the function.

This construction generalizes to inductive families and the predicate can also be generalized by equalities in a similar fashion as the dependent case construct

to allow recursion on subfamilies of a dependent inductive object. For example, consider defining `vlast`:

```
Equations vlast {A : Type} {n : nat} (v : vector A (S n)) : A :=
  vlast A n v by rec v :=
  vlast A ?(O) (Vcons a ?(O) Vnil) := a ;
  vlast A ?(S n) (Vcons a ?(S n) v) := vlast v.
```

Here we use recursion using `Below_vector`. When we encounter a recursion user node by `rec v` (witnessed as `Rec(v, s)` in the splitting tree), we apply the recursor for the type of `v`, after having properly generalized it. The recursion hypothesis is hence of the form:

$$\text{Below_vector } A \ (\lambda \ (n' : \text{nat}) \ (v' : \text{vector } A \ n'), \\ \Pi \ n \ (v : \text{vector } A \ (\text{S } n)), \ n' = \text{S } n \rightarrow v' \simeq v \rightarrow \text{vlast_comp } v) \ n \ v$$

When we use non-structural recursion, recursive calls are rewritten as applications of a trivial generic projection operator for the function:

$$\text{vlast_comp_proj} : \forall \ (A : \text{Type}) \ (n : \text{nat}) \ (v : \text{vector } A \ (\text{S } n)) \\ \{vcomp : \text{vlast_comp } v\} \rightarrow \text{vlast_comp } v$$

The last argument of the projection is implicit and will be filled either automatically by a proof search procedure or interactively by the user. When we typecheck a recursive call, the procedure will try to find a satisfying `vlast_comp` object in the context, simplifying and applying `Below_vector` hypotheses.

This method handles the structurally recursive definitions satisfactorily, but it is very inefficient. Indeed, if we try to reduce a program built with this recursor using a call-by-value reduction, there might be an exponential blowup as the object we are recursing on, that is the tuple of all possible recursive calls `below1`, will have to be computed for each call. This is not so important if we are using a lazy reduction strategy but it is prohibitive if we want to compute with a call-by-value strategy inside COQ, or compute with the extracted program in ML. Extraction removes the logical parts of a term (in `Prop`), like the manipulations on equality used during specialization by unification, but in this case the `Below` object is computational and must be kept.

To avoid this problem, we will use another way of witnessing the subterm relation that is entirely logical.

3.2 Generalized subterm relations

Our solution is to define the subterm relation on an inductive family and write functions by well-founded recursion on this relation. The solution is also restricted to inductive types in `Type`. Indeed we cannot define any irreflexive relation on inductives in `Prop`, as that would contradict the proof-irrelevance principle consistent with the calculus.

Definition 1 (Subterm relation). *Given a computational inductive type $\mathbb{I} : \Pi \Delta, \text{Type}$ with constructors $\overline{\mathbb{I}_i} : \Pi \Gamma_i, \mathbb{I} \overline{\vec{t}}$, we define the generalized subterm relation as an inductive type $\mathbb{I}^{sub} : \Pi \Delta_l \Delta_r, \mathbb{I} \overline{\Delta_l} \rightarrow \mathbb{I} \overline{\Delta_r} \rightarrow \text{Prop}$. For each*

constructor $l_i : \Pi \Gamma_i, l \vec{t}$ and for each binding of Γ_i of the form $(x : \Pi \Gamma_x, l \vec{u})$ we add a constructor to the relation: $l_n^{sub} : \Pi \Gamma_x \Gamma_i, l^{sub} \vec{u} \vec{t} (x \bar{\Gamma}_x) (l_i \bar{\Gamma}_i)$.

Before going further, we will simplify our development by considering only homogeneous relations. Indeed we can define for any inductive type $\Pi \Delta, l \Delta$ (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 2 (Telescope transformation). For any context Δ , we define packing $\Sigma(\Delta)$ and unpacking $\bar{\Sigma}(\Delta, s)$ by recursion on the context²:

$$\begin{aligned} \Sigma(\epsilon) &= \mathbf{unit} & \Sigma(x : \tau, \Delta) &= \Sigma x : \tau, \Sigma(\Delta) \\ \bar{\Sigma}(\epsilon, s) &= \epsilon & \bar{\Sigma}(x : \tau, \Delta, s) &= \pi_1 s, \bar{\Sigma}(\Delta, \pi_2 s) \end{aligned}$$

The heterogeneous subterm relation can hence be uncurried to form an homogeneous relation on $\Sigma i : \Sigma(\Delta), l \bar{\Sigma}(\Delta, i)$.

The traditional notion of well-founded relation as found in the Coq standard library is restricted to homogeneous relations and based on the following notion of accessibility:

Inductive `Acc` $\{A\} (R : A \rightarrow A \rightarrow \mathbf{Prop}) (x : A) : \mathbf{Prop} :=$
`Acc_intro` : $(\forall y : A, R y x \rightarrow \mathbf{Acc} R y) \rightarrow \mathbf{Acc} R x$.

An element of `Acc A R x` contains a proof that any preceding element of x by R (if any) is also accessible. As objects of `Acc` are inductive, there has to a finite proof for the accessibility of x , hence all possible chains $\dots R x_{i-1} x_i, R x_i x$ have to be finite. A relation is said to be well-founded if all elements of its support are accessible for it. This corresponds (classically) to the descending chain condition. We make a class to register well founded relations:

Class `WellFounded` $\{A : \mathbf{Type}\} (R : \mathbf{relation} A) := \mathbf{wellfounded} : \forall a, \mathbf{Acc} R a$.

It is then trivial to derive a fixpoint combinator by recursion on the accessibility proof, given a step function as before:

Definition `FixWf` $\{WF : \mathbf{WellFounded} A R\} (P : A \rightarrow \mathbf{Type})$
 $(step : \Pi x : A, (\Pi y : A, R y x \rightarrow P y) \rightarrow P x) : \Pi x : A, P x$.

Obviously, we can prove that the direct subterm relation defined above is well-founded. It follows by a simple induction on the object and inversion on the subterm proof relating the subterms and the original term. We still need to take the transitive closure of this relation to get the complete subterm relation. Again it is easily shown that transitive closure preserves well-foundedness.

Using this recursion scheme produces more efficient programs, as only the needed recursive calls have to be computed along with the corresponding proofs of the subterm relation. Extraction of `FixWf` is actually a general fixpoint.

We can use the same technique as before to use this fixpoint combinator in `Equations` definitions, we just need to deal with the currying when applying

² We omit type annotations for the construction of sums and the projections, they can be easily inferred.

it to an object in an inductive family. Consider the application of the fixpoint combinator for `vlast` again, our initial problem was:

$$\forall A\ n\ (v : \text{vector } A\ (\mathbb{S}\ n)), \text{vlast_comp } A\ n\ v$$

To apply our recursion operator over vectors, we must first prepare for a dependent elimination on v packed with its index n . To do so, we simply generalize by an equality between the packed object and a fresh variable of the packed type, giving us an equivalent goal:

$$\begin{aligned} & A : \text{Type } v' : \{index : \text{nat} \ \& \ \text{vector } A\ index\} \\ & \text{=====} \\ & \forall n\ (v : \text{vector } A\ (\mathbb{S}\ n)), v' = \text{existT } (\mathbb{S}\ n)\ v \rightarrow \text{vlast_comp } A\ n\ v \end{aligned}$$

We can now directly use the fixpoint combinator on the subterm relation for packed vectors with v' . This results in a new goal with an additional induction hypothesis expecting a packed vector and a proof that it is smaller than the initial packed v . Using currying, unpacking of existentials and the dependent elimination simplification tactic, we get back a goal refining the initial problem with the same patterns $A\ n\ v$.

The last step is to provide a proof search procedure to automatically build proofs of the subterm relation, filling the witnesses that appear at recursive calls. We can easily do so using a hint database with the constructors of the `!sub` relation and lemmas on the transitive closure relation that only allow to use the direct subterm relation on the right to guide the proof search by the refined v , emulating the unfolding strategy of `Below`.

Measures We need not restrict ourselves to the subterm relation for building well-founded definitions, we can also use any other available well-founded relation at our hands. A common one is provided by the inverse image relation produced by a function on a given relation, often referred to as a measure when the relation is the less-than order on natural numbers. We leave this generalization for future work.

4 Reasoning support

We now turn to the second part of the `EQUATIONS` package: the derivation of support definitions to help reasoning on the generated implementations.

4.1 Building equations

The easiest step is constructing the proofs of the equations as propositional equalities.

Definition 3 (Equations statements). *We recurse on the splitting tree, book-keeping the current label ℓ , initially ϵ , and for each `Compute`($\Delta \vdash \vec{p} : \Gamma, rhs$) node we inspect the right-hand side and generate a statement:*

- `Program`(t): the equation is simply $\Pi\ \Delta, f.\ell\ \vec{p} = t$.

- $\text{Refine}(t, \Delta' \vdash \overrightarrow{v}^x, x, \overrightarrow{v}_x : \Delta^x, x : \tau, \Delta_x, \ell', s)$: We know that the new programming problem is just a reordering of the variables in Δ after having inserted a declaration for the refined object and abstracted the remaining Δ_x context. The auxiliary definition $\mathbf{f}.\ell'$ produces an object refining this context, we can hence generate an indirection equation for the helper function: $\Pi \Delta, \mathbf{f}.\ell \overrightarrow{p} = \mathbf{f}.\ell' \overrightarrow{v}^x t \overrightarrow{v}_x$. We continue the generation of equations, considering the new programming problem and setting the current label to ℓ' .

All of these goals are solvable by simply unfolding the definition of the function and simplifying the goal: the constructor forms in the leaf patterns direct the reduction. If we didn't use any axioms during the definition, then these follow definitionally. When we encounter axioms in these proofs we simply rewrite using their expected computational behavior.

We create a database of rewrite rules named \mathbf{f} with the proofs of these equations, to allow simplification by rewriting, abstracting many steps of reasoning and computation in a single rewrite.

4.2 Induction principle

The next step is to build the inductive graph of the function. Considering $\mathbf{f} : \Pi \Gamma, \mathbf{f}_{\text{comp}} \overline{\Gamma}$, we want to build an inductive relation $\mathbf{f}_{\text{ind}} : \Pi \Gamma, \mathbf{f}_{\text{comp}} \overline{\Gamma} \rightarrow \text{Prop}$ that relates arguments $\overline{\Gamma}$ to results $\mathbf{f}_{\text{comp}} \overline{\Gamma}$.

We first define a function that finds the occurrences of recursive calls in a right-hand side term and abstracts them by variables. This is easy to do given that all the recursive calls are labeled by the trivial $\mathbf{f}_{\text{comp_proj}}$ projection.

Definition 4 (Abstracting recursive calls). *The $\text{ABSREC}(f, t)$ operator is defined by recursion on the term t , under a local context Δ , initially empty. The operator builds a context representing the abstracted recursive calls and a new term using these abstracted calls. By case on t :*

- $\mathbf{f}_{\text{comp_proj}} \overrightarrow{t} p$: We recursively compute the abstractions in \overrightarrow{t} giving us a new context Δ' and terms \overrightarrow{t}' . We extend Δ' with a fresh declaration $\text{res} := \lambda \Delta, \mathbf{f} \overrightarrow{t}' : \Pi \Delta, \mathbf{f}_{\text{comp}} \overrightarrow{t}'$ and the term becomes $\text{res } \overline{\Delta}$.
- $\lambda x : \tau, b$: Let the result of abstracting b in an extended context $\Delta, x : \tau$ be (Δ', b') , we return $(\Delta', \lambda x : \tau, b')$.
- $f e$: We simply combine the results of abstracting f and e separately.
- $\text{let } x := t \text{ in } b$: We do the abstractions in t resulting in Δ', t' and recursively call the abstraction on b in a context extended with $(x := t')$. We simply combine the resulting contexts and terms.
- Otherwise we return the empty context and the term unchanged.

Once we get the recursive calls abstracted, we will need to add induction hypotheses to the context.

Definition 5 (Induction hypotheses generation). *Given a context Δ of results produced by the $\text{ABSREC}(f, t)$ operator, we define the induction hypotheses context by a simple map on Δ , denoted $\text{HYPS}(\Delta)$. For each binding $\text{res} : \Pi \Delta, \mathbf{f}_{\text{comp}} \overrightarrow{t}$ we build a new binding $\text{resind} : \Pi \Delta, \mathbf{f}_{\text{ind}} \overrightarrow{t} (\text{res } \overline{\Delta})$.*

We are now ready to build the inductive graph. We will actually be building graphs for both the toplevel definition and each auxiliary definition, resulting in a mutual inductive type $\text{f.l.ind} : \Pi \Delta_{\ell.n}, \text{f.l.comp} \overline{\Delta}_{\ell.n} \rightarrow \text{Prop}$.

Definition 6 (Inductive graph). We compute the constructors of the f.l.ind relation by recursion on the splitting tree:

- $\text{Split}(c, x, s)$: Again splitting nodes are basically ignored, we just collect the constructor statements for the splittings s , if any.
- $\text{Rec}(v, s)$: Recursion nodes are also ignored when we compute the inductive graph, after all they just produce different ways to build f.comp objects. We just recurse on the splitting s .
- $\text{Compute}(\Delta \vdash \overline{p} : \Gamma, \text{rhs})$: By case on rhs:
 - $\text{Program}(t)$: We abstract the recursive calls of the term using the function $\text{ABSREC}(f, t)$ which returns a context ψ and a new term t' . We return the statement

$$\Pi \Delta \Psi \text{HYPS}(\Psi), \text{f.l.ind} \overline{p} t'$$

- $\text{Refine}(t, \Delta' \vdash \overline{v}^x, x, \overline{v}_x : \Delta^x, x : \tau, \Delta_x, \ell.n, s)$: As for the equation, we just have to do an indirection to the inductive graph of the auxiliary function, but we have to take into account the recursive calls of the refined term too. We compute $\text{ABSREC}(f, t) = (\psi, t')$ and return:

$$\begin{aligned} \Pi \Delta^x \Delta_x \Psi \text{HYPS}(\Psi) (\text{res} : \text{f.l.n.comp} \overline{\Delta}^x t' \overline{\Delta}_x) \\ \text{f.l.n.ind} \overline{v}^x t' \overline{v}_x \text{res} \rightarrow \text{f.l.ind} \overline{p} \text{res} \end{aligned}$$

We continue with the generation of the f.l.n.ind graph.

We can now prove that the function (and its helpers) corresponds to this graph by proving the following lemma:

Theorem 1 (Graph lemma). We prove $\Pi \Delta_\ell, \text{f.l.ind} \overline{\Delta}_\ell (\text{f.l} \overline{\Delta}_\ell)$ by following the splitting tree.

- $\text{Rec}(c, s)$: We replay recursion nodes, giving us new ways to prove f.ind that we will use to prove the goals corresponding to induction hypotheses.
- $\text{Split}(c, x, s)$: Each split is simply replayed, empty ones solve the goal directly.
- $\text{Compute}(\Delta \vdash \overline{p} : \Gamma, \text{rhs})$: At computation nodes our goal will necessarily be simplifiable by an equation because we replayed the whole splitting, i.e. it will have the form $\Pi \Delta, \text{f.l.ind} \overline{p} (\text{f.l} \overline{p})$. By case on rhs:
 - $\text{Program}(t)$: We rewrite with the equation for this node and apply one of the constructors for the graph. We will optionally get subgoals for induction hypotheses here if t had recursive calls in it. They are solved by a proof search, in exactly the same way as the proofs for the recursive calls were found.
 - $\text{Refine}(t, \Delta' \vdash \overline{v} : \Gamma', \ell.n, s)$: Here we can rewrite with the indirection equation and apply the indirection constructor for the inductive graph, then solve potential induction subgoals. We will be left with a subgoal for f.l.n.ind in which we must abstract the refined term t . We can then recurse on the goal: $\Pi \Delta_{\text{f.l.n}}, \text{f.l.n.ind} \overline{v} (\text{f.l.n} \overline{v})$.

□

4.3 Deriving an eliminator

Once we have the proof of the graph lemma $\Pi \Delta, \mathbf{f}_{\text{ind}} \overline{\Delta} (\mathbf{f} \overline{\Delta})$, we can specialize the eliminator of \mathbf{f}_{ind} which is of the form:

$$\begin{array}{l} \Pi (P : \Pi \Delta, \mathbf{f}_{\text{comp}} \overline{\Delta} \rightarrow \mathbf{Prop}) \overline{(P_{\mathbf{f},\ell} : \Pi \Delta_{\mathbf{f},\ell}, \mathbf{f}_{\text{comp}} \overline{\Delta}_{\mathbf{f},\ell} \rightarrow \mathbf{Prop})} \\ (f : \Pi \Gamma, \mathbf{f}_{\text{ind}} \overrightarrow{t} (\mathbf{f} \overline{u}) \rightarrow P \overrightarrow{t} (\mathbf{f} \overline{u}) \rightarrow \dots) \\ \vdots \\ \Pi \Delta (r : \mathbf{f}_{\text{comp}} \overline{\Delta}), \mathbf{f}_{\text{ind}} \overline{\Delta} r \rightarrow P \overline{\Delta} r \end{array}$$

This eliminator expects not only one proposition depending on the arguments of the initial call to \mathbf{f} but also a proposition for each one of the helpers corresponding to refinement nodes. We can easily define the proposition for $\mathbf{f}_{\text{f.l.n}}$ in terms of the proposition for $\mathbf{f}_{\text{f.l}}$. Each $P_{\mathbf{f},\ell,\mathbf{n}}$ is defined as:

$$\begin{array}{l} \lambda \Delta^x (x : \tau) \Delta_x (r : \mathbf{f}_{\text{f.l.n,comp}} \overline{\Delta}_{\mathbf{f},\ell,\mathbf{n}}), \Pi \Psi \text{HYPS}(\Psi)\{\mathbf{f}_{\text{ind}} := P\}, \\ \Pi H : x = t', P_{\mathbf{f},\ell} \overrightarrow{p} (\text{cast } r \ x \ H) \quad \text{where } \text{ABSREC}(f, t) = \Psi, t' \end{array}$$

We abstract on the context of the refinement node with its distinguished variable x and on a result r for the subprogram. As we know that this subprogram is called with a particular refined value t , we can assert the equality $x = t$ and cast the result with this equality to get back a term of type $\mathbf{f}_{\text{f.l,comp}} \overline{\Delta}^x \overline{\Delta}_x$: we are simply doing the inverse of the abstraction of t that happened during the typechecking of the refinement node. Of course, if t itself contains recursive calls, we must also abstract by the corresponding P hypotheses and use t' instead.

We want to eliminate not any $\mathbf{f}_{\text{comp}} \overline{\Delta}$ object but specifically \mathbf{f} calls. Using the graph lemma proof we can trivially specialize the conclusion to $\Pi \Delta, P \overline{\Delta} (\mathbf{f} \overline{\Delta})$

We can also remove the unnecessary hypotheses of the form $\mathbf{f}_{\text{ind}} \overrightarrow{t} (\mathbf{f} \overline{u})$ appearing in the methods, as they are all derivable from the graph lemma proof.

Finally, we can get rid of all the indirection methods as they are of the form:

$$\Pi \Delta_{\mathbf{f},\ell,\mathbf{n}} (r : \mathbf{f}_{\text{f.l,comp}} \overline{\Delta}_{\mathbf{f},\ell,\mathbf{n}}) \Psi \text{HYPS}(\Psi)\{\mathbf{f}_{\text{ind}} := P\}, P_{\mathbf{f},\ell,\mathbf{n}} \overline{\Delta}^x t' \overline{\Delta}_x r \rightarrow P_{\mathbf{f},\ell} \overrightarrow{p} r$$

These are readily derivable given the definitions of the $\mathbf{P.f.l.n}$ above: the equality hypothesis for the refinement is instantiated by a reflexivity proof, making the cast reduce directly. We are left with a tautology.

The cleaned up eliminator can be applied directly to any goal depending on \mathbf{f} , possibly after another generalization by equalities if the call has concrete, non-variable arguments. The elimination will give as many goals as there are $\mathbf{Program}()$ nodes in the splitting tree (possibly more than the number of actual user nodes due to overlapping patterns). The context will automatically be enriched by equalities witnessing all the refinement information available and of course induction hypotheses will be available for every recursive call appearing in these and the right hand sides. This gives rise to a very powerful tool to write proofs on our programs, and a lightweight one at the same time: all the details of the splitting, refinement and recursion are encapsulated in the eliminator.

5 Related Work

5.1 Dependent pattern-matching

The notions of dependent pattern-matching and coverage building were first introduced by Coquand in his seminal article [2] and the initial ALF implementation. It was studied in the context of an extension of the Calculus of Constructions by Cornes [10] who started the work on inversion of dependent inductive types that was later refined and expanded by McBride [11]. Subsequent work with McKinna and Goguen around EPIGRAM [7,12,1] led to the compilation scheme for dependent pattern-matching definitions which is also at the basis of EQUATIONS. Using the alternative external approach, Norell developed the AGDA 2 language [5], an implementation of Martin-Löf Type Theory that internalizes not only Streicher’s axiom K but also the injectivity of inductive types. In a similar spirit, Barras *et al* [4] propose to extend COQ’s primitive elimination rule to handle the unification process.

5.2 Recursion in type theory

Our treatment of recursion is comparable to the FUNCTION tool by Barthe *et al.* [13] which supports well-founded recursion and also generates an inductive graph and a functional induction principle. Our implementation is however more robust as the input program is sufficiently structured to give a complete procedure to generate the graph, and more powerful in its handling of dependent pattern-matching. It does not however remedy the combinatorial explosion due to the use of catch-all clauses in programs as it also uses an expansion strategy to compile pattern-matching.

Another powerful way to handle non-structural recursion in type theory was developed by Bove et Capretta [14]. The technique, based on the ability to first define the inductive domain of a function and delay the termination argument might now be adaptable in our setting.

5.3 Elaborations into type theory

The PROGRAM [15] extension of COQ which permits elaboration of COQ programs by separating programming and proving lacked the support for reasoning on definitions after the fact. We hope to combine the subset coercions system of PROGRAM inside EQUATIONS to get the best of both tools.

The EPIGRAM language also incorporates “views” and the application of arbitrary eliminators with `by` in addition to the `with` construct [12] which were not considered here.

6 Conclusion

Future Work Our setup should allow to extend the language easily with features like first-class patterns or views and support the application of custom tactics

during elaboration. We will also need to consider the general case of mutual (co)-inductive definitions. We also want to evaluate the effectiveness of this approach on a large example, comparing it to other function definition tools like `FUNCTION` or the `HOL` function package.

We have presented a new tool for defining programs using dependent-pattern matching in the `Coq` system, automatically generating a supporting theory to ease post-hoc reasoning on them. The system has a safe architecture, living entirely outside the kernel and allowing easy extension thanks to its reliance on the high-level tactic language and type classes constructs. We hope it provides a more robust, accessible and powerful user interface to the calculus.

References

1. Goguen, H., McBride, C., McKinna, J.: [Eliminating Dependent Pattern Matching](#). In Futatsugi, K., Jouannaud, J.P., Meseguer, J., eds.: *Essays Dedicated to Joseph A. Goguen*. Volume 4060 of LNCS, Springer (2006) 521–540
2. Coquand, T.: [Pattern Matching with Dependent Types](#) (1992) Proceedings of the Workshop on Logical Frameworks.
3. Augustsson, L.: [Compiling Pattern Matching](#). In: *FPCA*. (1985) 368–381
4. Barras, B., Corbineau, P., Grégoire, B., Herbelin, H., Sacchini, J.L.: [A New Elimination Rule for the Calculus of Inductive Constructions](#). In Berardi, S., Damiani, F., de'Liguoro, U., eds.: *TYPES*. Volume 5497 of LNCS., Springer (2008) 32–48
5. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (2007)
6. McBride, C.: [Elimination with a Motive](#). In Callaghan, P., Luo, Z., McKinna, J., Pollack, R., eds.: *TYPES*. Volume 2277 of LNCS., Springer (2000) 197–216
7. McBride, C., Goguen, H., McKinna, J.: [A Few Constructions on Constructors](#). *Types for Proofs and Programs* (2004) 186–200
8. Sozeau, M., Oury, N.: [First-Class Type Classes](#). In Otmane Ait Mohamed, C.M., Tahar, S., eds.: *Theorem Proving in Higher Order Logics, 21th International Conference*. Volume 5170 of LNCS. Springer (2008) 278–293
9. Streicher, T.: *Semantics of Type Theory*. Springer-Verlag (1991)
10. Cornes, C.: Conception d'un langage de haut niveau de représentation de preuves: Récurrence par filtrage de motifs, Unification en présence de types inductifs primitifs, Synthèse de lemmes d'inversion. PhD thesis, Université Paris 7 (1997)
11. McBride, C.: [Inverting Inductively Defined Relations in LEGO](#). In Giménez, E., Paulin-Mohring, C., eds.: *TYPES*. Volume 1512 of LNCS., Springer (1996) 236–253
12. McBride, C., McKinna, J.: [The view from the left](#). *J. Funct. Program.* **14** (2004) 69–111
13. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: [Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant](#). *Functional and Logic Programming* (2006) 114–129
14. Bove, A., Capretta, V.: [Modelling general recursion in type theory](#). *Mathematical Structures in Computer Science* **15** (2005) 671–708
15. Sozeau, M.: Un environnement pour la programmation avec types dépendants. PhD thesis, Université Paris 11, Orsay, France (2008)