

# Equations Reloaded

A definitional extension of Coq for dependent and recursive function definitions

ANONYMOUS AUTHOR(S)

EQUATIONS is a plugin for the Coq proof assistant which provides a notation for defining programs by dependent pattern-matching and structural or well-founded recursion and derives useful proof principles for demonstrating properties about them. We build on a first prototype of EQUATIONS that was presented in [30]. We present a much improved, general design and rewriting of the implementation that avoids the use of axioms and reliance on  $\mathcal{L}_{\text{tac}}$  programming, providing a robust, feature-rich function definition package for Coq as a definitional extension to the Coq kernel. At the core of the system is a new simplifier for dependent equalities that can be reused to define enhanced, axiom-free 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.

## 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 automatically derives useful reasoning principles in the form of propositional equations describing the functions, and elimination principles that ease reasoning on them. 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 [26], where the justification of dependent-pattern matching definitions in terms of core rules is proven almost entirely internally as in [10] but the core system is extended with evidence-free higher-level rules directly, simplifying the implementation work substantially.

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 : nat) (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 (a.k.a. “forced”) `?(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<sup>1</sup>. From this definition, `EQUATIONS` will generate a function called `nth` which obeys the equalities given by the user as clauses, using first-match semantics in case of overlap, and realizing the expanded clauses as definitional equalities in general (we will discuss the computational behavior of the generated definitions shortly). Along with the definition, `EQUATIONS` automatically generates propositional equalities for the defining equations of the function, its graph and associated elimination principle. The construction of these derived terms is entirely generic and based on the intermediate splitting tree representation of functions used during compilation. These provide additional assurance that the compilation is meaning-preserving. In the case of `nth`, the generated lemmas are<sup>2</sup>:

```

59 Check nth_equation_1 : ∀ (A : Type) (f : fin 0), ImpossibleCall (nth nil f).
60 Check nth_equation_2 : ∀ (A : Type) (n : nat) (a : A) (v : vector A n), nth (cons a v) (fz n) = a.
61 Check nth_equation_3 : ∀ A n a v f, nth (cons a v) (fs n f) = nth v f.
62
63 Check nth_elim : ∀ P : ∀ (A : Type) (n : nat), vector A n → fin n → A → Prop,
64   (∀ A n a v, P A (S n) (cons a v) (fz n) a) →
65   (∀ A n a v f, P A n v f (nth v f) → P A (S n) (cons a v) (fs n f) (nth v f)) →
66   ∀ A n v f, P A n v f (nth v f).

```

The first generated “equation” is actually a proof that `nth (nil A) f` is an impossible call (i.e. a proof of `False`), which can be used to discharge directly goals where such calls appear. Both `nth_equation_2` and `nth_equation_3` are proved by reflexivity (i.e. `eq_refl`). Finally the eliminator `nth_elim` provides an abstract view on `nth`, it witnesses that any proof about `nth v f` can be equivalently split in two cases: (i) one where the arguments are refined to `cons a v` and `fz` and the result of the call itself is refined to `a` and (ii) another for `cons a v` and `fs f`, where we get an induction hypothesis for the recursive call to `nth v f`. This provides an economic way to prove properties of functions as the recursion and pattern-matching steps involved in the function definition are entirely summarized by this principle.

## Issues of trust

While the difference of viewpoint between a core calculus extension and an elaboration might seem only aesthetic and of little practical 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 developments 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 and recursion, 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) or the univalence principle [32]. These two axioms are inconsistent in general.

**Thesis** We can construct a definitional extension that is by construction compatible with any consistent axiom and does not enlarge the trusted code base, while providing the benefits of a high-level abstract view on function definitions by pattern-matching and recursion. The following sections explain our design choices to achieve this goal.

<sup>1</sup>There must be only one binding occurrence for every variable in the pattern, all other occurrences appear under inaccessible annotations. In case the inaccessible / forced term is itself a variable the implementation could allow any of the occurrences to be the non-inaccessible one, while we currently force a particular one

<sup>2</sup>We declared the type argument `A` of `nil` and `cons` implicit, as well as the `n` argument of `cons` for conciseness, and generally elide unnecessary type annotations

## 1.1 The identity type

Both axioms pertain to the identity type of type theory, also known as propositional equality. We recall its definition as it is the central inductive family used in this work:

**Inductive**  $\text{eq} \{A : \text{Type}\} (x : A) : A \rightarrow \text{Prop} := \text{eq\_refl} : x = x \text{ where } "x = y" := (\text{eq } x \ y) : \text{type\_scope}.$

Equality is an equivalence relation and its elimination principle  $\text{eq\_rect\_dep}$  is a dependent version of the Leibniz substitution principle, coined the J rule in type theory jargon:

**Check**  $\text{eq\_rect\_dep} : \forall A \ x (P : \forall y : A, x = y \rightarrow \text{Type}) (p : P \ x (\text{eq\_refl } x)),$   
 $\forall (y : A) (e : x = y), P \ y \ e.$

Informally, this principle states that to prove a goal depending on a term  $y$  and a proof of equality  $x = y$ , it suffices to show the case where  $y$  is substituted by  $x$  and the equality by  $\text{eq\_refl } x : x = x$ .

The computation rule of  $\text{eq\_rect\_dep}$  is reducing to its single arm  $p : P \ x (\text{eq\_refl } x)$  when  $e$  is  $\text{eq\_refl } x$ , and the canonicity property of the theory ensures that if  $p : t = u$  in the empty context (i.e. when  $p, t$  and  $u$  are closed terms), then  $t$  and  $u$  are convertible and  $p$  is  $\text{eq\_refl}$ . Said otherwise, propositional equality *reflects* convertibility in the empty context. It is however a much larger relation under context: equality proofs can be built from induction principles, or be elaborate “lies” under inconsistent contexts. It is good to bear in mind these intuitions when working with the (seemingly trivial) identity type.

## 1.2 A short history of dependent pattern-matching

The first version of dependent pattern-matching was introduced by COQUAND in [14], axiomatically defining a notation for dependent pattern-matching programs, and later refined by MCBRIDE *et al.* [22], using a definitional translation, with both systems assuming the UIP principle from the start. In [15], dependent pattern-matching was explained in terms of simplification of heterogeneous equalities which were defined using the UIP principle (although, in his PhD [22], 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. Uniqueness of Identity Proofs states that all equality proofs at *any* type are equal.

$$\text{UIP} : \forall (A : \text{Type}) (x \ y : A) (p \ q : x = y), p = q \quad (1)$$

This axiom is consistent with but independent from Martin-Löf Type Theory and the Calculus of Inductive Constructions (CIC) [16], while it is easily derivable in Extensional Type Theory [21, p32] and Observational Type Theory [3]. It can easily be shown equivalent to the so-called K axiom which stipulates that all proofs of *reflexive* equality are equal to  $\text{eq\_refl}$ :

$$\text{K} : \forall (A : \text{Type}) (x : A) (p : x = x), p = \text{eq\_refl} \quad (2)$$

Enter Homotopy Type Theory (HoTT) and Univalence [28], whose central principle contradicts directly the uniqueness of identity proofs principle. Univalence proclaims that equality of types is equivalent to equivalence of types:

$$\text{univalence} : \forall (A \ B : \text{Type}), (A = B) = \text{Equiv } A \ B \quad (3)$$

Informally, in Homotopy Type Theory, one is interested in the higher-dimensional structure of equality types, which are shown to form weak  $\infty$ -groupoids [19, 33]. This is in direct conflict with the UIP principle which states, in terms of HoTT, that every type is an homotopy set, that is a discrete space where the only paths are identities/reflexivities on a point. UIP implies that the higher-dimensional structure of identity at any type is trivial. Already at the level of types, one can build two equivalences from booleans to booleans, the identity and the negation. The axiom allows

148 to derive that the equality of types  $\mathbb{B} = \mathbb{B}$  has two distinct elements, contradicting UIP. One can  
 149 however still show using a result of Hedberg [17] that usual datastructures with decidable equality  
 150 like natural numbers enjoy UIP, provably.

151 To remedy this apparent conflict, and give a meaning to dependent pattern-matching compatible  
 152 with univalence, one has to move to a view of heterogeneous equality which does not rely on UIP  
 153 at all types. This can be done using telescopes, or the notion of “path over a path”, easily encoded in  
 154 pure type theory using sigma types. This was done in [12] for an “axiomatic” version implemented  
 155 in AGDA and [20] for a “definitional” translation in Coq, which clearly circumscribed the cases  
 156 where the UIP principle was necessary during compilation. At this point, UIP, or the assumption  
 157 that some type is an HSet was necessary for the deletion rule (to dependently eliminate an equality  
 158  $e : t = t$ ) and to simplify problems of injectivity between indexed inductive types.

159 Since then, Cockx [11] introduced an alternative solution to injectivity which can remove some  
 160 later uses of the UIP principle, justified by reasoning on higher-dimensional equalities. This ought  
 161 to bring a happy conclusion to the “--without-K” story of AGDA, which should enforce that UIP is  
 162 not provable and had a history of bug reports where proofs of UIP were found repeatedly, fix after  
 163 fix. This result should settle these issues once and for all by providing a solid theoretical background  
 164 to the axiomatic dependent pattern-matching implemented in AGDA. However, note that even this  
 165 last solution involves constructing “out of thin air” a substitution that should come from a chain  
 166 of computationally-relevant type equivalences. While we were able to reproduce this result<sup>3</sup>, and  
 167 checked the reasoning used to build this substitution, any change to the core calculus implies a  
 168 requirement of trust towards its implementation, whose burden we avoid in the case of EQUATIONS  
 169 by providing a definitional translation.

### 171 1.3 UIP versus Univalence

172 In practice both the UIP and the Univalence principle have value. In a theory with UIP built-in, for  
 173 example in a version of the calculus of constructions with a definitionally proof-irrelevant **Prop**  
 174 (like in LEAN [6]), one can formulate dependent pattern-matching compilation by working with  
 175 equalities in **Prop** and freely use UIP to simplify any pattern-matching problem. Moreover, this  
 176 compilation is guaranteed to have good computational behavior as all the decoration added by  
 177 the compilation are proof manipulations that are guaranteed to be computationally irrelevant by  
 178 construction. In the setting of Coq, this has an impact on extraction: extraction of definitions by  
 179 EQUATIONS when using the equality in **Prop** removes all the proof manipulations involved, leaving  
 180 only the computational content. This is important in case one wants to actually compute with these  
 181 definitions or their extraction, e.g. through a certified compiler like CertiCoq [5] that does erasure  
 182 of proofs.

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

### 188 1.4 Computational Behavior

189 Using a definitional translation, compilation of dependent pattern-matching introduces many proof-  
 190 manipulations to the implementations of definitions. It is actually the point of this elaboration  
 191 to relieve the user from having to witness reasoning on the theory of equality, constructors and  
 192 indexed inductive types to implement definitions by dependent pattern-matching. In section 5  
 193

195 <sup>3</sup>Anonymized: material/theories/telescopes.v, all .v files we refer to in the article are in the same directory.

we will show how we can minimize the decorations, but that is only a correctness-preserving optimization, which cannot remove all decorations in general.

Nonetheless, we can prove that the intuitive high-level computational behavior of a definition, looking at the clauses after compilation to a case tree (disambiguating overlapping patterns), is properly implemented by the compiled terms. That is a kind of computational soundness theorem, which relies on the condition that the compilation does not make use of a propositional K rule or an axiom. In case the compilation relies on a proof of K (e.g. derived using decidable equality of an index type), the system is still able to prove propositional equalities corresponding to the actual reduction rules of the definition *on closed terms* only. Finally, in case the compilation uses K as an axiom, the propositional equalities can be derived but we provide no guarantee about the computational behavior of the function inside Coq. We only know that its *extraction*, which removes all decorations, will have the expected computational behavior.

### 1.5 Pattern-Matching and Recursion

Dependently-typed programming involves not only pattern-matching on indexed families but also recursion. There are basically two ways to present inductive families in type theories:

- (1) The first is based on associating a dependent eliminator constant to each inductive family, with associated rewrite rules that enrich the definitional equality of the system, combining the structural recursion and pattern-matching constructs. This eliminator construction is usually justified from the construction of initial algebras in a categorical model.
- (2) Another way is to separate pattern-matching and recursion using two different language constructs, corresponding to ML's `match` and `let rec`. This is the solution adopted in Coq, where we have generic `match` and `fix` constructs that handle the computational behavior of any inductive type in the *shema* of inductive definitions accepted by the theory [27]. This provides more flexibility in the shape of definitions one can readily write in the language, e.g. allowing structural recursion on deep subterms of a recursive argument, and it allows to reuse the computational machinery associated to ML-like languages. For example, there is an obvious adaptation of the operational semantics of these constructs to abstract machines, and a more direct translation to other functional languages than using eliminators and rewrite rules.

The downside of the more expressive option chosen in Coq is that there is a complex syntactic guard-checking criterion that must be used to check that definitions are normalizing, as part of the type-checking algorithm implemented in the kernel. This algorithm is relatively concise and provides welcome flexibility but it has many drawbacks:

- (1) it has no up-to-date formal proof: most formal reasoning on CIC actually uses subtly different type-based variants of this check inspired by the theory of sized types [1].
- (2) it is inherently non-modular and the current implementation performs unsafe reductions during checking. The result is that it actually only checks normalization of definitions using a call-by-name reduction strategy, which is weaker than strong normalization.
- (3) it is a major source of critical bugs in the kernel.

The most disturbing bug in recent times is instructive. It was discovered by researchers working in Homotopy Type Theory: the guardness check was too lax allowing pattern-matching (e.g. `match`) terms to be considered subterms if all their branches were subterms. This criterion results in an inconsistency in presence of Univalence, or even the weaker Propositional Extensionality axiom that was believed to be consistent with Coq since its inception. The size-change termination criterion of Agda, also basically based on syntax was also oblivious to this problem. The fix to this issue has

yet to see a completely formal justification, and actually weakens the guard checking in an drastic way, disallowing perfectly fine definitions in CoQ (§5.3 illustrates this).

Again, to avoid these subtle trust issues, our solution is simple: elaborate complex recursive definitions using the tools of the logic itself instead of trying to extend the core calculus. We will do so using the well-known, constructive accessibility characterization of well-founded recursion and provide high-level constructs so that it can readily apply to inductive families. Combined with our elimination principle generation machinery, this provides a powerful *definitional* framework for dealing with mutual, nested, and well-founded recursive definitions using dependent pattern-matching.

## 1.6 Contributions

In its first version [30], the EQUATIONS tool was relying on heterogeneous equality (a.k.a. “John-Major” equality) to implement the so-called “specialization by unification” [15] 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 global and local where clauses for defining mutual or nested recursive functions and nested well-founded programs respectively (§ 2). The eliminators derived for mutual and nested programs are the most general ones and allow to work much more comfortably with nested inductive definitions than in vanilla CoQ.
- A cleaner elaboration of EQUATIONS definitions, avoiding the use of a construction of [15] 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 record of constants representing logical connectives with their introduction and elimination principles, so that it can use the usual CoQ connectives in **Prop** or connectives defined 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 released, stable and freely available<sup>4</sup>. It has been tested on a variety of examples, including a proof of strong normalization for predicative System F [20] and a reflexive tactic for deciding equality of polynomials which we present in more detail in section 6. 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.

<sup>4</sup>Anonymized: material/equations.tgz

295 *Structure of the paper.* We will present the contributions in stages corresponding to the compiler's  
 296 stages: in §2 we present the user-level features of EQUATIONS in a tutorial way, focusing on the  
 297 treatment of structural and well-founded recursive definitions. In §3 we introduce the formal source  
 298 language of EQUATIONS and present its architecture, as well as a dependent elimination tactic  
 299 derived from it. In §4 we recall the theory of dependent pattern-matching compilation using equality  
 300 of sigma types and present our ML compiler. In §5 we develop an optimization of simplification  
 301 that can reduce term size and complexity of definitions. In §6 we provide a short walkthrough of  
 302 an example of using the new system to develop a reflexive decision procedure. Finally we review  
 303 related work in §7 and conclude.

## 304 2 FROM STRUCTURAL TO NESTED WELL-FOUNDED RECURSION

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

308 The most direct way to define a recursive function is to just reuse the name of the function in  
 309 any right-hand side of a clause. In this case, the user relies on COQ's guard condition to check that  
 310 the definition is terminating, as in the `nth` example in the introduction.

### 311 2.1 Mutual structural recursion (`mutrec.v`)

312 EQUATIONS supports mutual recursion on mutual inductive types at the top-level, using a syntax  
 313 close to vanilla COQ. We demonstrate it on an example formalization of a term-language for  $\lambda$ -  
 314 calculus with non-empty application spines and well-scoped variables. We define this datatype  
 315 as a mutual indexed inductive type of `terms` and `spines`. Technically, we use a non-recursively  
 316 uniform parameter here instead of a proper index for the number of free variables. That is, the  
 317 parameter varies in the arguments of the constructor, e.g. for `Lam` where we introduce a fresh free  
 318 variable, but not in their conclusions, which we even omit here for `term`. They are treated the same  
 319 by EQUATIONS.

320 `Inductive term (n : nat) : Set := Var (f : fin n) | Lam (t : term (S n)) | App (t : term n) (l : spine n)`  
 321 `with spine (n : nat) : Set := tip : term n → spine n | snoc : spine n → term n → spine n.`

322 Suppose we want to define capture-avoiding substitution for this language. We first need to  
 323 define lifting of a well-scoped term with  $n$  variables into a well-scoped term with  $n+1$  free variables,  
 324 shifting variables above or equal to  $k$  by 1. We handle separately the case of variables:

325 `Equations lift_fin {n} (k : nat) (f : fin n) : fin (S n) :=`  
 326 `lift_fin 0 f := fs f; lift_fin (S k) (fz n) := fz; lift_fin (S k) (fs n f) := fs (lift_fin k f).`

327 Lifting a `fin` changes the index to make space for the new free variable. Using a toplevel `where`  
 328 clause, one can define the mutually recursive lifting function on terms and spines. The definition is  
 329 accepted by the guard checker easily.

330 `Equations lift {n} (k : nat) (u : term n) : term (S n) := {`  
 331 `lift k (Var f) ⇒ Var (lift_fin k f); lift k (Lam t) ⇒ Lam (lift (S k) t);`  
 332 `lift k (App f ts) ⇒ App (lift k f) (lift_spine k ts) }`  
 333 `where lift_spine {n : nat} (k : nat) (t : spine n) : spine (S n) := {`  
 334 `lift_spine k (tip t) ⇒ tip (lift k t);`  
 335 `lift_spine k (snoc ts t) ⇒ snoc (lift_spine k ts) (lift k t) }.`

336 This representation lends itself naturally to a definition of parallel substitution. To define it at  
 337 the case of abstraction, one must explain how to lift a substitution of  $k$  variables to a substitution of

344  $k+1$  variables by preserving the  $0$ th variable and lifting the result of the substitution otherwise. This  
 345 pattern-matching involves the no-confusion principle on the natural number index (constructors  
 346 are distinct and injective), but does not require UIP.

```
347 Equations extend_var (k : nat) (u : fin (S k) → term k) (f : fin (S (S k))) : term (S k) :=
348   extend_var k u (fz ?(S k)) ⇒ Var fz ; extend_var k u (fs ?(S k) f) ⇒ lift 0 (u f).
```

349 Again, using a toplevel `where` clause, we can define this parallel substitution, structurally on the  
 350 term and spine.

```
351
352 Equations subst_term {k : nat} (u : fin (S k) → term k) (t : term (S k)) : term k := {
353   subst_term u (Var v) ⇒ u v ; subst_term u (Lam t) ⇒ Lam (subst_term (extend_var k u) t) ;
354   subst_term u (App t l) ⇒ App (subst_term u t) (subst_spine u l) }
355 where subst_spine {k : nat} (u : fin (S k) → term k) (t : spine (S k)) : spine k := {
356   subst_spine u (tip t) ⇒ tip (subst_term u t) ;
357   subst_spine u (snoc ts t) ⇒ snoc (subst_spine u ts) (subst_term u t) }.
```

358 *Testing.* We can run this program inside Coq to test it. First we need to represent a single  
 359 substitution of a variable by a closed term, i.e. a function of type `fin 1 → term 0`, and a substitution  
 360 into terms with 1 free variable `subst1`. This is definable by pattern-matching on the `fin _` arguments.

```
361 Equations subst0 (t : term 0) (f : fin 1) : term 0 := subst0 t (fz _) := t.
362 Equations subst1 (t : term 1) (f : fin 2) : term 1 := subst1 t (fz _) := t ; subst1 t (fs _ f) := Var fz.
```

363 The definition of `subst0` has a single case for `fz` as the `fs _ f'` case is ruled out automatically by  
 364 EQUATIONS which can infer that `f' : fin 0` is uninhabited using a single dependent case analysis (this  
 365 heuristic could be parameterized). We can now check that substitution is indeed capture-avoiding.

```
366 Definition id {n} : term n := Lam (Var fz). Definition idfree : term 1 := Lam (Var (fs fz)).
367 Definition Ω_body : term 1 := App (Var fz) (tip (Var fz)). Definition Ω := Lam Ω_body.
```

368 We define the identity function  $\lambda x. x$ , which can be lifted transparently to any number of free  
 369 variables,  $\lambda x. y$  as a term with one free variable, along with `Ω_body`, self-application of a variable  
 370 to itself, and the corresponding term  $\lambda x. x x$  and test:

```
371 Check eq_refl : subst_term (subst0 Ω) id = id. — (λx. x)[Ω/0] = λx. x
```

```
372 Check eq_refl : subst_term (subst0 Ω) Ω_body = App Ω (tip Ω). — (x x)[Ω/0] = Ω Ω
```

373 Finally, we check that substituting into a non-closed term does not capture bound variables:

```
374 Check eq_refl : subst_term (subst1 (Var fz)) (lift 1 idfree) = idfree. — (λ x. y)[z/y] = λx. z
```

## 375 2.2 Nested structural recursion (nestedrec.v)

376 Mutual recursion can be seen as a special form of *nested* recursion, where an inductive type is  
 377 defined mutually with a previously defined inductive type taking it as a parameter. Coq natively  
 378 supports the definition of nested inductive types, however there is little high-level support for  
 379 working with such definitions: either when writing programs or when reasoning on these inductive  
 380 types, the user is faced with the delicate representation of nested fixpoints, and the system does  
 381 not derive expressive enough eliminators automatically. EQUATIONS provides a higher-level view  
 382 on these types.

383 *2.2.1 Nested definitions.* A common use-case for these types is nesting the type of lists in the  
 384 definition of a new inductive type. Here we take the example of a well-scoped  $\lambda$ -term structure  
 385 with two constructors taking lists of terms as arguments: application and a `Meta` node. `Meta`  
 386 represents metavariables applied to a substitution in the language; this is how existential variables  
 387 are represented in Coq's open term syntax for example.

392



```

393 Inductive term (n : nat) : Set := Var (f : fin n) | Lam (t : term (S n))
394 | App (t : term n) (l : list (term n)) | Meta (id : nat) (l : list (term n)).

```

395  
396 Again, we can define lifting and substitution on this datatype. We skip the definition of lifting  
397 which can be found in the accompanying source code. For definitions of fixpoints on nested  
398 mutual inductive types, EQUATIONS allows to factorize the nested fixpoint definitions in toplevel  
399 **where** clauses, so that multiple calls to the nested function can refer to the same function. Below,  
400 **subst\_terms** is called multiple times in **subst\_term**, and of course it recursively calls itself and  
401 **subst\_term**.

```

402 Equations subst_term {k : nat} (u : fin (S k) → term k) (t : term (S k)) : term k := {
403   subst_term u (Var v) ⇒ u v; subst_term u (Lam t) ⇒ Lam (subst_term (extend_var u) t);
404   subst_term u (App t l) ⇒ App (subst_term u t) (subst_terms u l);
405   subst_term u (Meta t l) ⇒ Meta t (subst_terms u l) }
406 where subst_terms {k} (u : fin (S k) → term k) (t : list (term (S k))) : list (term k) := {
407   subst_terms u nil ⇒ nil; subst_terms u (cons t ts) ⇒ cons (subst_term u t) (subst_terms u ts) }.

```

408 The Coq kernel will check a single fixpoint definition for **subst\_term** where **subst\_terms** has  
409 been expanded at its call sites, as definitions on nested recursive types correspond to nested local  
410 fixpoints in CIC. The regular structural guardedness check is able to check that this definition is  
411 terminating. Note that one can optionally add a **struct x** annotation to **where** clauses to indicate  
412 which argument decreases explicitly.

413  
414 **2.2.2 Reasoning.** Remark that our definition of **subst\_terms** is equivalent to a call to **map** on  
415 lists. EQUATIONS currently needs the “expanded” version to properly recognize recursive calls, but  
416 one can readily add this equation to the **subst\_term** rewrite database to abstract away from this  
417 detail:

```

418 Lemma subst_terms_map k u t : @subst_terms k u t = List.map (@subst_term k u) t.
419 Proof. induction t; now simpl; rewrite ?IHt. Qed. HintRewrite subst_terms_map : subst_term.

```

420 The elimination principle generated from this definition is giving a conjunction of two predicates  
421 as a result, and has the proper induction hypotheses for nested recursive calls:

```

422 Check subst_term_elim : (* Predicates/Motives *)
423   ∀ (P : ∀ k : nat, (fin (S k) → term k) → term (S k) → term k → Prop)
424   (P0 : ∀ k : nat, (fin (S k) → term k) → list (term (S k)) → list (term k) → Prop),
425   (* Obligations/Methods *)
426   (∀ k u (f : fin (S k)), P k u (Var f) (u f)) →
427   (∀ k u (t : term (S (S k))), P (S k) (extend_var u) t (subst_term (extend_var u) t) →
428     P k u (Lam t) (Lam (subst_term (extend_var u) t))) →
429   (∀ k u (t0 : term (S k)) (l : list (term (S k))), P k u t0 (subst_term u t0) →
430     P0 k u l (subst_terms u l) → P k u (App t0 l) (App (subst_term u t0) (subst_terms u l))) →
431   (∀ k u (id0 : nat) (l0 : list (term (S k))), P0 k u l0 (subst_terms u l0) →
432     P k u (Meta id0 l0) (Meta id0 (subst_terms u l0))) → (∀ k u, P0 k u [] []) →
433   (∀ k u (t : term (S k)) (l : list (term (S k))), P k u t (subst_term u t) →
434     P0 k u l (subst_terms u l) → P0 k u (t :: l) (subst_term u t :: subst_terms u l)) →
435   (* Conclusion/Target*)
436   (∀ k u t, P k u t (subst_term u t)) ∧ (∀ k u t, P0 k u t (subst_terms u t)).

```

437  
438 One may want to specialize **P0** with **Forall2 P** to recover a **map**-like elimination principle. From  
439 **subst\_term\_elim**, one can indeed automatically derive another eliminator with a single predicate  
440 **P**, filling the last two methods for the recursive definition on lists of terms and assuming proofs of  
441

442 `forall2 P l` in induction hypotheses of `App` and `Meta` instead, see the accompanying `nestedrec.v`  
 443 source file for its construction.

444

### 445 2.3 Well-founded recursion

446 *Issues with the guardness check and a way out.* While the implementation of the guard condition  
 447 has been adapted over the years to try and allow as many safe cases as possible, it is still obviously an  
 448 approximation and may fail in some legitimate cases. This is aggravated by the fact that `EQUATIONS`  
 449 uses a lot of rewriting, making the job of the guard condition checker that much more difficult. In  
 450 many cases everything works as expected, but sometimes it (apparently) diverges because it must  
 451 unfold definitions or it fails to track the subterm relation correctly in the term due to rewritings  
 452 (applications of `J`) or lack of *commutative cuts*. Indeed a problematic case appears when the recursive  
 453 subterm is abstracted in branches of a match for example, a typical situation where  $\beta$  and  $\iota$  redexes  
 454 are mixed and do not commute. The syntactic check of `COQ` cannot check the typing constraints  
 455 on commutation that would allow some recursive definitions to pass. `EQUATIONS` provides an  
 456 automatic derivation of the well-foundedness of the `Subterm` relation on inductive families to  
 457 handle these cases where the guardedness check fails. It can also be the case that the recursion is  
 458 simply not structural and the user wants to use well-founded recursion.

459 Well-founded functions are defined as usual, except `EQUATIONS` will afterwards ask the user to  
 460 prove some obligations about the well-foundedness of the relation, and that the arguments decrease  
 461 according to the given order for each recursive call, like `PROGRAM` or `FUNCTION`. The worked out  
 462 example in section §6 gives many such examples.

463

464 *2.3.1 Nested well-founded recursion (rose.v).* To demonstrate nested well-founded recursive  
 465 definitions, we take a well-known example from the literature: rose trees. We will define a recursive  
 466 function gathering the elements in a `rose` tree in an efficient way, using nested well-founded  
 467 recursion instead of the guardness check of `COQ`. The `rose` trees are defined as trees whose nodes  
 468 contain lists of trees, i.e. forests.

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

470

471 This is a nested inductive type we can measure assuming a `list_size` function for measuring lists.  
 472 Here we use the usual guardness check of `COQ` that is able to unfold the definition of `list_size` to  
 473 check that this definition is terminating.

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

475

476 As explained at the beginning of this section, however, if we want to program more complex  
 477 recursions, or rearrange our terms slightly and freely perform dependent pattern-matching, the  
 478 limited syntactic guardness check will quickly get in our way.

479 Using a *nested where* clause and the support of `EQUATIONS` for well-founded recursion, we can  
 480 define the following function gathering the elements in a rose tree efficiently:

481 `Equations elements (r : rose) (acc : list A) : list A :=`  
 482 `elements r l by rec r (MR lt size) :=`  
 483 `elements (leaf a) acc := a :: acc;`  
 484 `elements (node l) acc := aux l _`  
 485 `where aux x (H : list_size size x < size (node l)) : list A :=`  
 486 `aux x H by rec x (MR lt (list_size size)) :=`  
 487 `aux nil _ := acc;`  
 488 `aux (cons x xs) H := elements x (aux xs (list_size_smaller x xs l H)).`

489 `Definition elems r := elements r nil.`

490

The function is nesting a well-founded recursion inside another one, based on the measure of `rose` trees and lists (MR  $R f$  is a combinator for  $\lambda x y, R (f x) (f y)$ ). The termination of this definition is ensured solely by logical means, it does not require any syntactic check. Note that the auxiliary definition's type mentions the variable  $l$  bound by the enclosing pattern-matching, to pass around information on the size of arguments. Local `where` clauses allow just that. This kind of nested pattern-matching and well-founded recursion was not supported by previous definition packages for COQ like `FUNCTION` or `PROGRAM`, and due to the required dependencies it is not supported by `ISABELLE`'s `Function` package either (see [4] for a survey of the treatment of recursion in type-theory based tools).

We can show that `elems` is actually computing the same thing as the naïve algorithm concatenating elements of each tree in each forest.

```
Equations elements_spec (r : rose) : list A :=
  elements_spec (leaf a) := [a]; elements_spec (node l) := concat (List.map elements_spec l).
```

As `elements` takes an accumulator, we first have to prove a generalized lemma, typical of tail-recursive functions:

```
Lemma elements_correct (r : rose) acc : elements r acc = elements_spec r ++ acc.
```

*Proof.*

```
let elim := constr:(fun_elim (f:=elements)) in
  apply (elim (fun r acc f => f = elements_spec r ++ acc)
    (fun l acc x H r => r = concat (List.map elements_spec x) ++ acc));
  intros; simp elements_spec; simpl. now rewrite HI, H0, app_assoc. Qed.
```

We apply the eliminator providing the predicate for the nested recursive call and simplify using the `simp elements_spec` tactic which is rewriting with the defining equations of `elements_spec`. The induction hypotheses and associativity of concatenation are enough to solve the remaining goal which involves the two recursive calls to `elements` and `aux`. The above proof is very quick as the eliminator frees us from redoing all the nested recursive reasoning and the proofs that the induction hypotheses can be applied. It is now trivial to prove the correctness of our fast implementation:

```
Lemma elems_correct (r : rose) : elems r = elements_spec r.
```

**2.3.2 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 really is 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`:

$$\forall \Delta, f \bar{\Delta} = f\_unfold \bar{\Delta} \quad (4)$$

Using this lemma, we can also express cleanly the elimination principle of  $f$ , abstracting away from the proofs used to prove its termination.

*Accessibility is propositionally proof-irrelevant.* If we try to prove this lemma directly, we hit problems at partially applied recursive calls of  $f$ : our induction hypothesis would equate  $f$  and `f_unfold`, while the goals we would get would relate the unfolding of the `FixWf` combinator

underlying  $\mathbf{f}$  and  $\mathbf{f}$ . However the unfolding of  $\mathbf{FixWf}$  is *not* convertible to  $\mathbf{f\_unfold}$ , as it still relies on a subterm of the accessibility proof. Therefore, this proof method is not modular.

Using functional extensionality, it is possible to prove that constructive accessibility as defined in  $\mathbf{Coq}$  is proof-irrelevant, a folklore result. From this, it is then possible to prove a general unfolding lemma for  $\mathbf{FixWf}$ , that can be used to prove equation 4. Hence, our proofs of unfolding for well-founded recursive definitions rely on the functional extensionality axiom. This is the only axiom used by  $\mathbf{EQUATIONS}$ .

*Accessibility cannot be definitionally proof-irrelevant.* Note that even in a system with *definitionally* proof-irrelevant propositions, accessibility cannot be made irrelevant, as this breaks either decidability of type-checking or the transitivity property of conversion. This is because the precise computational behavior of the fixpoint combinator relies essentially on the shape of the accessibility proof (morally, it tells how many times the fixpoint can be unfolded). Trying to make it irrelevant means that one has to guess how many times the fixpoint should be unfolded during conversion without looking at the accessibility proof, which can always be a lie in type theory!

The proposal of Guarded Cubical Type Theory [9] is based on this insight as well: fixpoints are guarded by a computationally relevant path / propositional equality that cannot be degenerated into a definitional equality.

*Functional extensionality.* We argue that 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. A proper solution to the axiom issue is to work in a type theory which supports functional extensionality like OTT [3] or Cubical Type Theory[13].

This concludes our presentation of  $\mathbf{EQUATIONS}$ 's source language and associated tactics.

### 3 INTERPRETING EQUATIONS

We will now delve deeper in the compilation chain which starts from the source language to build splitting trees (§3), then compiles splitting trees to terms (§4.2 & §4.3) which we'll optimize in §5. From splitting trees, the defining equations and the graph of the function can be generically derived, for lack of space we do not detail this here. One can consult [30] for details of that construction, which straightforwardly extends to *where* clauses, mutual and nested (well-founded) recursion.

#### 3.1 Notations and terminology

We will use the notation  $\overline{\Delta}$  to denote the list of variables bound by a typing context  $\Delta$ , in the order of declarations, and also to denote lists in general. An *arity* is a type of the form  $\forall \Gamma, s$  where  $\Gamma$  is a (possibly empty) context and  $s$  is a sort (the  $\forall$  notation is overloaded to work on context rather than a single declaration). A sort (or kind) can be either **Prop** (categorizing propositions) or **Type** (categorizing computational types, like **bool**). The type of any type is always an arity. We will ignore universe levels throughout, but the system works with  $\mathbf{Coq}$  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  $\mathbf{l} : \forall \Delta, s$  and constructors  $\mathbf{l}_i : \forall \Gamma_i, \overline{\mathbf{t}}_i$  (where  $\Gamma_i \vdash \overline{\mathbf{t}}_i : \Delta$ ). 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. Likewise, the extension to mutual inductive types is straightforward but complicates notations,

589 hence we do not treat them formally here. The dependent sum / sigma type is written  $\Sigma x : \tau. \tau'$ ,  
 590 its introduction form is  $(-, -)$  and its projections are in postfix notation  $_.1 : \Sigma x : \tau. \tau' \rightarrow \tau$  and  
 591  $_.2 : \forall s : (\Sigma x : \tau. \tau'). \tau'[s.1]$ . The compilation process starts from a signature and a list of clauses  
 592 given by the user, constructed from the grammar given in figure 1.

593	term, type	$t, \tau$	$::=$	$x \mid \lambda x : \tau, t \mid \forall x : \tau, \tau' \mid \dots$
594	binding	$d$	$::=$	$(x : \tau) \mid (x := t : \tau)$
595	context	$\Gamma, \Delta$	$::=$	$\bar{d}$
596	program	$prog$	$::=$	$f \Gamma : \tau := \bar{c}$
597	user clause	$c$	$::=$	$f \overline{up} n$
598	user pattern	$up$	$::=$	$x \mid C \overline{up} \mid ?(t)$
599	user node	$n$	$::=$	$:=! x \mid \text{with } t := \{ \bar{c} \} := t \text{ where } \overline{prog} \mid \text{by } \text{rec } x R := prog$

601 Fig. 1. Definitions and user clauses  
 602

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

610 The syntax supports **with** clauses, for example, take the definition of **filter**:

```
611 Equations filter {A : Type} (P : A → bool) (l : list A) : list A :=
612 filter _ nil := nil;
613 filter P (cons hd tl) with P hd := { | true := cons hd (filter P tl); | false := filter P tl }.
```

614 In the second clause of the **filter** function, a pattern is added to the current problem which can  
 615 then be scrutinized like any other pattern, calling for a follow-up list of internal clauses, building a  
 616 subprogram. EQUATIONS can prove a relevant elimination principle for this function with 3 branches,  
 617 one for each leaf of the program, with hypotheses of the form  $P \text{ hd} = \text{true}$  or  $P \text{ hd} = \text{false}$  for the  
 618 respective branches of the subprogram.  
 619

### 620 3.2 Searching for a covering

621 The goal of the compiler is to produce a proof that the user clauses form an exhaustive covering of  
 622 the signature, compiling away nested pattern-matchings to simple case splits. As we have multiple  
 623 patterns to consider and allow overlapping clauses, there may be more than one way to order the  
 624 case splits to achieve the same results. We use inaccessible patterns (noted  $?(t)$ ) as in AGDA to help  
 625 recover a sense of what needs to be destructed and what is statically known to have a particular  
 626 value, but overlapping clauses force the compilation to be phrased as a search procedure. As usual,  
 627 we recover a deterministic semantics using a first-match rule when two clauses overlap. The search  
 628

629	program	$prog$	$::=$	$(\ell_p, \Gamma, \text{rec}?, spl)$
630	recursion	$rec$	$::=$	$wf(t, R) \mid \text{struct } x$
631	pattern substitution	$c$	$::=$	$\Delta \vdash \bar{p} : \Gamma$
632	pattern	$p$	$::=$	$x \mid C \bar{p} \mid ?(t)$
633	splitting	$spl$	$::=$	$\text{Split}(c, x, ((spl)?)^n) \mid \text{Compute}(c, t, prog^*)$

634 Fig. 2. Grammar of programs, splitting trees and pattern substitutions  
 635  
 636  
 637

for a covering works by gradually refining a *pattern substitution*  $\Delta \vdash \bar{p} : \Gamma$  and building a splitting tree. A pattern substitution (fig. 2), is a substitution from  $\Delta$  to  $\Gamma$ , associating to each variable in  $\Gamma$  a pattern  $p$  typable in  $\Delta$ . We start the search with the problem  $\Gamma \vdash \bar{\Gamma} : \Gamma$ , i.e. the identity substitution on  $\Gamma$ , 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 initially declared type  $\tau$ .

The search for a covering and building of the splitting tree is entirely standard and mostly unmodified from the previous version (one can refer to [30] for details). This follows the intuitive semantics of dependent pattern-matching (e.g., the same as in AGDA or LEAN): covering succeeds if we can exhaustively unify the types of the patterns in each clause with the types of the matched objects, for unification in the theory of constructors and equality, up-to definitional equality.

So, we consider that we are directly given a splitting tree corresponding to our definition. A splitting can either be:

- A `Split`( $\Delta \vdash \bar{p} : \Gamma, x, (s?)^n$ ) node denoting that the variable  $x$  is an object of an inductive type with  $n$  constructors and that splitting it in context  $\Delta$  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 pattern substitution used in each of the subcoverings.
- A `Compute`( $\Delta \vdash \bar{p} : \Gamma, t, \bar{w}$ ) node, denoting a right-hand side whose definition is  $t$  (of type  $\tau[\bar{p}]$ ) under some set of auxiliary local definitions  $\bar{w}$ . Both `with` and `where` clauses are compiled this way. A `with` clause is essentially interpreted as a `where` clause with a single argument for the abstracted object and correspondingly generalized return type. The `with` clauses differ from arbitrary `where` clauses essentially because when generating the elimination principle of the function one can automatically infer the (refined) predicate applying to the `where` subprogram from the enclosing program's predicate. General `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 `wf`( $t, R$ ) annotation denotes an application of the well-founded fixpoint combinator to define the rest of the function  $\ell_p$  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  $\Gamma$ .
- A `struct`  $x$  annotation denotes a usual structurally recursive fixpoint of `Coq`, where  $x$  is a single variable declared in the context  $\Gamma$ .

### 3.3 Elaboration of recursion

The compilation of mutual and nested recursive definitions presented in §2 is mainly a delicate engineering issue. It is a matter of threading the recursive prototypes in the splitting tree (using the `hide(x)` patterns), in the end generating functionals that can be used to build primitive fixpoints blocks or be passed to a well-founded fixpoint combinator. To generate the unfolding and elimination principles, a simple substitution operation on splitting trees is used to produce the splitting tree of the unfolded version. We will now focus on the compilation of pattern-matching which is at the center of the compiler, starting with a use of the splitting tree structure before going further and looking at the generation of terms from splitting trees in §4.

### 3.4 A dependent elimination tactic (material/theories/fle\_trans.v)

The covering mechanism provides an easy way to implement 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  $l \bar{t}$  from context  $\Gamma$ . The dependent elimination tactic takes as input from the user a list of patterns corresponding to the different cases of this elimination. It is also possible to give no patterns, in that case the tactic will generate some by starting to build a splitting tree with a `Split` node on variable  $x$ . In any case, for each pattern  $p$ , the tactic produces an equation where:

- the left-hand side is the list of all variables in context  $\Gamma$ , except for the variable  $x$  which is replaced by the pattern  $p$ ;
- the right-hand side is a `Program` node  $:= ?h$  with a hole  $?h$  as a term.

We have a type and a list of equations, 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. 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  $\{1 \dots n\}$ .

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

```
Inductive fle :  $\forall n, \text{fin } n \rightarrow \text{fin } n \rightarrow \text{Set} :=
| flez :  $\forall n (j : \text{fin } (S n)), \text{fle } fz j | fles :  $\forall n (i j : \text{fin } (S n)), \text{fle } i j \rightarrow \text{fle } (fs i) (fs j).$$$ 
```

We will need a `NoConfusion` principle for `fin`, which we derive automatically, see section 4.1.2.

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 our dependent elimination tactic:

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

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] ; [ 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, using: `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.

## 4 CRAFTING TERMS FOR COQ

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 [15] and [30] with minor modifications. We recall the main construction here, and then present the simplification engine used by `EQUATIONS` to perform specialization by unification.

### 4.1 Prerequisites

We will need a few tools to implement the compilation of splitting trees, or more specifically the dependent elimination of a variable.

*4.1.1 Packing inductives.* First of all, we will simplify our development by considering only homogeneous relations between inductive families. Indeed we can define for any inductive type  $\forall \Delta, l \Delta$  (any arity in general) a corresponding closed type by wrapping the indices  $\Delta$  in a dependent sum and both the indices and the inductive type in another dependent sum.

*Definition 4.1 (Telescope transformation).* For any context  $\Delta$ , we define packing of a context  $\Sigma(\Delta)$  or an instance  $\sigma(\Delta)(i)$  and unpacking  $\bar{\Sigma}(\Delta, s)$  by recursion on the context.

$$\begin{array}{lll}
\Sigma(\epsilon) = \mathbf{unit} & \Sigma(x : \tau, \Delta) = \Sigma x : \tau, \Sigma(\Delta) & \Sigma(x := t : \tau, \Delta) = \Sigma(\Delta[t/x]) \\
\sigma(\epsilon)(\epsilon) = \mathbf{tt} & \sigma(x : \tau, \Delta)(t, \bar{\delta}) = (t, \sigma(\Delta)(\bar{\delta})) & \sigma(x := t : \tau, \Delta)(\bar{\delta}) = \sigma(\Delta[t/x])(\bar{\delta}) \\
\bar{\Sigma}(\epsilon, s) = \epsilon & \bar{\Sigma}(x : \tau, \Delta, s) = s.1, \bar{\Sigma}(\Delta, s.2) & \bar{\Sigma}(x := t : \tau, \Delta, s) = \bar{\Sigma}(\Delta[t/x], s)
\end{array}$$

For an inductive  $\mathbf{l} : \forall \bar{\Delta}, s$ , its packing is defined as  $\Sigma i : \Sigma(\Delta), I \bar{\Sigma}(\Delta, i)$ . We follow [11] and denote this type as  $\bar{\mathbf{l}}$ . It 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 l** command. This provides in particular a trivial function to inject a value in the signature:

$$\mathbf{signature\_pack} : \forall \Delta(x : \mathbf{l} \bar{\Delta}), \Sigma i : \Sigma(\Delta), I \bar{\Sigma}(\Delta, i) := \lambda(\bar{\Delta} : \Delta)(x : \mathbf{l} \bar{\Delta}), (\sigma(\Delta)(\bar{\Delta}), x)$$

*4.1.2 Injectivity and discrimination of constructors.* During the simplification part of dependent elimination – which we will cover below – the simplifier will need to deal with equalities between constructors. We need a tactic that can simplify any equality of telescopes, that is an equality of the shape:

$$(i_0, \mathbf{C}_0 \bar{a}_0) =_{\bar{\mathbf{l}}} (i_1, \mathbf{C}_1 \bar{a}_1) \quad \text{where } \forall j \in \{0, 1\}, \mathbf{C}_j \bar{a}_j : I \bar{\Sigma}(\Delta, i_j) \quad (5)$$

As an aside, this is the first time we see an equality between telescopes. In [24] as well as the previous version of EQUATIONS, an equality between telescopes was interpreted as a sequence of heterogeneous equalities based on so-called John Major equality.

**Inductive JMeq**  $\{A : \mathbf{Type}\} (x : A) : \forall \{B : \mathbf{Type}\}, B \rightarrow \mathbf{Prop} := \mathbf{JMeq\_refl} : \mathbf{JMeq} \ x \ x$ .

With this equality type, we do not need to care about the dependencies between equalities and can just consider them independently. The main drawback is that using an equality  $\mathbf{JMeq} \ x \ y$  between two terms in the same type requires to invoke an axiom equivalent to UIP, which we want to avoid. Therefore, as was explained in [11] we instead use telescopes and homogeneous equalities. However, contrary to [11] we mainly make use of equalities of telescopes, instead of telescopes of equalities. Both are however equivalent, that is:

**Theorem teq\_eqtel**  $\{A : \mathbf{Type}\} \{B : A \rightarrow \mathbf{Type}\} (x1 \ x2 : A) (y1 : B \ x1) (y2 : B \ x2) :$   
 $\{e : x1 = x2 \ \& \ \mathbf{eq\_rect} \ y1 \ e = y2\} \leftrightarrow (x1, y1) = (x2, y2)$ .

On the equality (5), the tactic should either give us equalities between the arguments  $\bar{a}_0$  and  $\bar{a}_1$  (injectivity) that can be further simplified or deriving a contradiction if  $\mathbf{C}_0$  is different from  $\mathbf{C}_1$  (conflict). McBride *et al.*[24] describe a generic method to derive such an eliminator that can be adapted to work on telescopic equalities instead of heterogeneous equalities – [11] describes it in detail. We implement this construction as another **Derive** scheme in COQ. For any (computational) inductive type  $\mathbf{l} : \forall \Gamma, \mathbf{Type}$ , we can use **Derive NoConfusion for l** to derive an instance of the type class **NoConfusionPackage**  $\bar{\mathbf{l}}$  that provides a proof of isomorphism of the two types:

$$\forall x \ y : \bar{\mathbf{l}}, \mathbf{NoConfusion} \ \bar{\mathbf{l}} \ x \ y \simeq x =_{\bar{\mathbf{l}}} y \quad (6)$$

When  $x$  and  $y$  are of the shape in equation (5), **NoConfusion**  $x \ y$  directly reduces to either **True**, **False** or an equality between the arguments  $\bar{a}_0$  and  $\bar{a}_1$ . 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.



785 4.1.3 *The logic interface.* As we intend for EQUATIONS to be usable in different settings, using the  
 786 usual proof-irrelevant equality or a univalent one, we aim to abstract the logic interface required by  
 787 EQUATIONS to be able to perform elimination using any equality satisfying a minimal interface. As  
 788 such, the user will be able choose between the usual equality in the impredicative Prop universe,  
 789 a proof-relevant and universe polymorphic identity type like HoTT’s path type, or even another  
 790 custom equality type. It is still a work-in-progress in the implementation to be parametric enough,  
 791 but the groundwork for it is done<sup>5</sup>.

792 The simplifier used in EQUATIONS is mainly dealing with uses of the eliminator for equality, and  
 793 equalities between constructors. We encapsulate the basic blocks that it needs in a few lemmas and  
 794 classes that have to be provided for any equality type to be used with EQUATIONS. Due to space  
 795 constraints, we externalize the listing of the definitions required in this interface that can be found  
 796 in `intf.mli`. To summarize, we need an equality type with the usual constructor and eliminators  
 797 and several lemmas about K, NoConfusion and equalities of telescopes. We also need most of  
 798 the lemmas about equalities to preserve `eq_refl` (i.e. reduce to `eq_refl` when applied to `eq_refl`) to  
 799 ensure the preservation of the computational behavior of definitions.

## 801 4.2 Compilation of a splitting tree

802 4.2.1 *Overview.* After building a splitting tree, the overall process of compiling it to a Coq term  
 803 is a straightforward recursive algorithm. In the case of a `Compute( $\Delta \vdash \bar{p} : \Gamma, t, \bar{w}$ )` node, we simply  
 804 need to check that the user-given term  $t$  has the expected type  $\tau[\bar{p}]$  under context  $\Delta$ , and compile  
 805 each auxiliary local definition in  $\bar{w}$ .

806 For a `Split( $\Delta \vdash \bar{p} : \Gamma, x, (s?)^n$ )` node, we can recursively compile each subtree to obtain one  
 807 term for each branch after the elimination of the variable  $x$ . The interesting part is the dependent  
 808 elimination of  $x$ , for which we need to produce a Coq term to witness the dependent elimination.  
 809

810 4.2.2 *Generalization, elimination, specialization.* The dependent pattern-matching notation acts  
 811 as a high-level interface to a unification procedure on the theory of constructors and uninterpreted  
 812 functions. Our main building block in the compilation process is hence a mechanism to produce  
 813 witnesses for the resolution of constraints in this theory, that is used to compile `Split` nodes. The  
 814 proof terms will be formed by applications of simplification combinators dealing with substitution  
 815 and proofs of injectivity and discrimination of constructors, their two main properties.

816 The design of this simplifier is based on the “specialization by unification” method developed  
 817 in [23, 24]. The problem we face is to eliminate an object  $x$  of type  $\mathbb{1} \bar{t}$  in a goal  $\Gamma \vdash \tau$  potentially  
 818 depending on  $x$ . We want the elimination to produce subgoals for the allowed constructors of this  
 819 family instance. To do that, we generalize the goal by a fresh variables:  $(i : \Sigma(\Delta)) (x' : \mathbb{1} \bar{\Sigma}(\Delta, i))$   
 820 and an equation between telescopes asserting that  $x'$  is equal to  $x$ , giving us a new, equivalent goal:  
 821

$$822 \Gamma, i : \Sigma(\Delta), x' : \mathbb{1} \bar{\Sigma}(\Delta, i) \vdash (i, x') = (\sigma(\Delta)(\bar{t}), x) \rightarrow \tau \quad (7)$$

823  
 824 After unpacking the index  $i$  to its constituent variables and reductions, this gives us an equivalent  
 825 goal where  $x'$  is a general instance of  $\mathbb{1}$ , i.e., it is applied to *variables* only, so no information is  
 826 lost by applying the general eliminator. Applying this we get subgoals corresponding to all its  
 827 constructors, all starting with an equation relating the indices  $t$  of the original instance to the  
 828 indices of the constructor. We will use the algorithm presented in sections 4.3 and 5 to simplify  
 829 these equations.

830  
 831  
 832 <sup>5</sup>See Anonymized: `HoTT_light.v` for an example from the HoTT library [8]

### 4.3 A simplification engine in OCAML

While the previous version of EQUATIONS relied on  $\mathcal{L}_{\text{tac}}$ , the tactic language shipped with COQ, to compile a splitting tree to a term, this approach caused quite a few problems, mainly due to the lack of elegant way to communicate information between what has been computed in OCAML and what needs to be done in  $\mathcal{L}_{\text{tac}}$ .

Therefore, in the current version, we moved most of the compilation procedure, and more specifically the simplification engine used to perform specialization by unification, to an OCAML module. We gain a more robust engine for the simplification that we present here, as well as the possibility of fine-tuning the way we eliminate a variable, as we will see in section 5.

This engine 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  $c$ . Apart from small bureaucratic details, the term  $c$  will simply be an application of the appropriate lemma from the logic interface and we will omit it in the description of the steps.

*A note on K.* There are two simplification steps which make use of  $K$  on a given type: dependent **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. There is no fundamental reason for not using  $K$  directly, it just happens that in practical cases, having decidable equality is natural. If there is ever a use for it, we will be able to switch to using  $K$  only.

**4.3.1 Simplification steps.** In this section we describe each simplification step in order. For each one, we show 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 [10] 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  $\Gamma$  which we do not write most of the time because it will not change. The arrow  $\Rightarrow$  in the presentation of the steps denotes the progression of the goal, not an implication – in other words, the right hand-side would imply the left hand-side, not the other way around. 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.

$$\text{Remove sigma } \boxed{\forall(e : (x, p) = (y, q)), P e} \Rightarrow \boxed{\forall(e' : x = y) (e : \text{rew } e' p = q), P (\text{sigma\_eq } e' e)}$$

This step ensures that the other simplification steps do not need to deal with equality of telescopes but rather a curried telescope of equalities, making use of the equivalence between the two shown in 4.1.2. The function `sigma_eq` combines the two equalities into one well-typed equality between  $(x, p)$  and  $(y, q)$ , while `rew` (a.k.a. `ap` or `subst`) rewrites in the type of  $p$  with the equality  $e'$ ; it is just an application of  $J$ .

**Deletion**  $\boxed{\forall (e : t = t), P e} \Rightarrow \boxed{P \text{ eq\_refl}}$

This step requires K on the type of  $t$ , as described above, unless  $P$  does not actually depend on  $e$ . In that case, we can just remove  $e$  and do not need K, as shown in the example `nondep_K.v`. Such cases arise more frequently in proofs than in definitions.

**Solution**  $\boxed{\forall \Gamma, \forall (e : x = t), P x e} \Rightarrow \boxed{\forall \Gamma', P t \text{ eq\_refl}}$

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 an OCAML function which, from a context, a variable  $x$  and a term  $t$ , computes a pattern substitution 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 pattern substitution.

**True and False**  $\boxed{\forall (e : \text{True}), P e} \Rightarrow \boxed{P \text{ I}}$      $\boxed{\forall (e : \text{False}), P e} \Rightarrow \boxed{\text{solved}}$

These steps are trivial and solve some goals produced by the **NoConfusion** step.

**4.3.2 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  $\Gamma = (A : \text{Type})(n : \text{nat})(x y : A)(v w : \text{vector } A n)$  and the goal:

$$\Gamma \vdash \text{cons } x v = \text{cons } y w \rightarrow \text{vector } A n$$

where `vector` is the same inductive type of length-indexed lists as in the introduction. When we are done simplifying the equality at the head of the goal, we expect the variables  $x$  and  $y$  to be unified, as well as the variables  $v$  and  $w$ . We do so with the following steps.

**Pack**  $\boxed{\forall (e : C \bar{t} = D \bar{u}), P e} \Rightarrow \boxed{\forall (e : (\overline{id x_t}, C \bar{t}) = (\overline{id x_u}, D \bar{u})), \text{ind\_pack\_inv } P e}$

As we define `NoConfusionPackage` on a homogeneous type, we need to pack the values with their indices. This step requires K on the type of the indices of the inductive type. There is room for improvement by adapting the idea of [10], 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:

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

**NoConfusion**  $\boxed{\forall (e : (\overline{id x_t}, C \bar{t}) = (\overline{id x_u}, D \bar{u})), P e}$

$\Rightarrow$   $\boxed{\begin{array}{l} - \forall (e : \bar{t} = \bar{u}), P (\text{noConf\_inv } e) \text{ if } C \text{ and } D \text{ are the same constructor} \\ - \text{solved if } C \text{ and } D \text{ are distinct constructors} \end{array}}$

To implement this step, we use the `NoConfusionPackage` class that we are able to derive automatically (see section 4.1.2). 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`.

932 Following the example, we get this goal:

$$933 \quad \Gamma \vdash \forall(e : \text{NoConfusion } (\mathbf{S} \ n, \text{cons } x \ v) (\mathbf{S} \ n, \text{cons } y \ w)),$$

$$934 \quad \text{ind\_pack\_inv } (\text{vector } A \ n) (\text{noConfusion\_inv } e)$$

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

$$937 \quad \Gamma \vdash \forall(e : (x, n, v) = (y, n, w)), \text{ind\_pack\_inv } (\text{vector } A \ n) (\text{noConfusion\_inv } e)$$

939 We end up with a telescopic equality at the head, which we can recursively simplify to unify  
940 its left- and right-hand sides, by using the same simplification steps described previously. If  
941 the constructors were distinct, for instance **cons** and **nil**, then the application of **NoConfusion**  
942 would instead reduce to **False**, leading to a trivial absurdity.

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

$$945 \quad A \ n \ (y : A)(v : \text{vector } A \ n) \vdash \text{ind\_pack\_inv } (\text{vector } A \ n) (\text{noConfusion\_inv } \text{eq\_refl})$$

$$946 \quad \text{Unpack } \boxed{\text{ind\_pack\_inv } P \ \text{eq\_refl}} \Rightarrow \boxed{P \ \text{eq\_refl}}$$

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

949 In practice, we now rely on the fact that **noConfusion\_inv** applied to **eq\_refl** reduces to  
950 **eq\_refl** to conclude and get the final goal after unification:

$$951 \quad (A : \text{Type})(n : \text{nat})(y : A)(v : \text{vector } A \ n) \vdash \text{vector } A \ n$$

953 **4.3.3 A simplification tactic.** The simplification engine we just described has been implemented  
954 to be as independent as possible from **EQUATIONS**. We used it to provide a simplification tactic that  
955 can apply to any goal with an equality between such telescopes. The user can either let the tactic  
956 infer steps to apply, or specify a sequence of steps.

957 For instance, with a goal such as  $\mathbf{S} \ n = \mathbf{S} \ m \rightarrow n = m$ , one could use this simplification tactic and  
958 write `simplify  $\{\rightarrow\}$`  to end up with  $m = m$  as goal through a **NoConfusion** step and a **Solution**  
959 step. The current syntax, which is subject to change, uses  $\rightarrow$  and  $\leftarrow$  for a **Solution** step to the right  
960 or to the left,  $-$  for **Deletion** and  $\{\dots\}$  for **NoConfusion**.

961 Since the **NoConfusion** step requires to simplify the equations it generates, it is natural to have  
962 a nested syntax for this step. It is also possible to let the module decide which step to apply by  
963 using one of the inference rules:  $?$  to let it infer one step, and  $*$  to let it fully simplify a telescopic  
964 equality.  
965

## 966 5 SMART CASE

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

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

$$972 \quad \text{match } y \text{ in } \bar{I} \ \bar{u} \text{ return } \Delta \rightarrow \bar{t}' = \bar{u}' \rightarrow \top \text{ with } \dots \text{ end } \Delta \ \text{eq\_refl}$$

974 In this term,  $(y : \bar{I} \ \bar{u})$  is a fresh variable that is introduced to generalize  $x$  and  $\top$  is some type  
975 corresponding to the goal we want to prove. In the return type,  $\bar{t}' = \bar{u}'$  is a telescopic equality  
976 between subsets of the telescopes  $(\bar{I}, x)$  and  $(\bar{u}, y)$ . This telescopic equality will then need to be  
977 simplified in each branch of the **match** by using the simplifications steps from the previous section.  
978 Finally,  $\Delta$  is a list of variables whose type we need to rewrite. In the rest of this section we will call  
979 these variables *cuts* in reference to the same problematic commutative cuts from section 2.3.

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.

### 5.1 From telescopes to elimination

Let us consider some context  $\Gamma_0 (x : l \bar{t}) \Gamma_1$ , 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 (7), is the following:

- (1) Introduce fresh binders for new indices, a new inductive value and an equality.

$$\Gamma_0 (x : l \bar{t}) \Gamma_1 (\bar{u} : \bar{\tau}) (y : l \bar{u}) (e : (\bar{t}, x) = (\bar{u}, y))$$

- (2) Eliminate the fresh variable  $y$  using its dependent eliminator. This produces one branch for each constructor of the inductive type  $l$ . In the branch for constructor  $C_i : \forall \bar{T}_i, l \bar{u}_i$ , the indices  $\bar{u}$  and the variable  $y$  are instantiated according to the type of  $C_i$ , and fresh constructors arguments  $\bar{a}$  are introduced.

$$\Gamma_0 (x : l \bar{t}) \Gamma_1 (\bar{a} : \bar{T}_i) (e : (\bar{t}, x) = (\bar{u}_i, C_i \bar{a}))$$

- (3) Simplify the equality  $e$ , effectively unifying  $x$  with  $C_i \bar{a}$  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 in l  $\bar{u}$  return  $(\bar{t}, x) = (\bar{u}, y) \rightarrow T$  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:

$$\Gamma_0 (x : l \bar{t}) \Gamma_1 (\bar{u} : \bar{\tau}) (y : l \bar{u}) (e : (\bar{t}, x) = (\bar{u}, y))$$

Recall that an equality of telescopes is equivalent to a telescope of equalities where each one can depend on the previous ones.

$$\Gamma_0 (x : l \bar{t}) \Gamma_1 (\bar{u} : \bar{\tau}) (y : l \bar{u}) (e_1 : t_1 = u_1) e_2 \dots e_n$$

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:

$$\Gamma (\bar{u} : \bar{\tau}) (y : l \bar{u}) \Delta (e : \bar{t}' = \bar{u}')$$

where  $e$  is a telescopic equality (possibly empty) which is some subset of the initial equality, and  $\Delta$  is a list of variables cut from the initial context. 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 in l  $\bar{u}$  return  $\Delta \rightarrow \bar{t}' = \bar{u}' \rightarrow T$  with ... end  $\Delta$  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 – that is, before building the `match`: homogeneous equalities and equalities on variables on which nothing depends.

## 5.2 Homogeneous solutions

Firstly, we consider in order each element of  $(\bar{t}, x)$  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:

$$\Gamma (\bar{u} : \bar{\tau}) (y : l \bar{u}) \Delta e_0 (e : x = u_j) e_1$$

If we have chosen to solve the equality  $(e : x = u_j)$ , then we move  $x$  and the variables  $\Delta_x$  that depend on it *after*  $y$  and its indices. We can perform this permutation because the linearity criterion ensures that  $x$  is a variable not in  $(\bar{u}, y)$ .

$$\Gamma' (\bar{u} : \bar{\tau}) (y : l \bar{u}) (x : \tau_j) \Delta_x \Delta e_0 (e : x = u_j) e_1$$

It is possible that  $x$  was already in  $\Delta$ , 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  $e_0$ .

$$\Gamma' (\bar{u} : \bar{\tau}) (y : l \bar{u}) (x : \tau_j) (e : x = u_j) \Delta_x \Delta e_0 e_1$$

Finally, we can apply a **Solution** step on  $e$ .

$$\Gamma' (\bar{u} : \bar{\tau}) (y : l \bar{u}) \Delta_x [x := u_j] \Delta e_0 e_1 [x := u_j, e := \text{eq\_refl}]$$

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. Using this optimization allows eliminations on already general instances (i.e., instances made only of variables occurring linearly) to reduce to a simple match with no equality manipulations anymore, resulting in simpler (and arguably more natural) compiled terms.

## 5.3 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. To make clear that this optimization can apply at all, it is important to remind that the variable we want to originally eliminate does appear in the telescope under consideration. Therefore, in a case where the goal was not dependent in this variable, we can remove it from the telescope, and then maybe some previous indices.

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

$$\Gamma (\bar{u} : \bar{\tau}) (y : l \bar{u}) \Delta e_0 (e : \text{rew } e_0 \ x = u_j) e_1$$

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  $e$ .

$$\Gamma' (\bar{u} : \bar{\tau}) (y : l \bar{u}) \Delta e_0 (x : \tau_j) (e : \text{rew } e_0 \ x = u_j) e_1$$

If  $x$  was in  $\Delta$  instead of  $\Gamma$ , it does not change anything.

Finally, we can perform a **Solution** step on  $e$ :

$$\Gamma' (\bar{u} : \bar{\tau}) (y : l \bar{u}) \Delta e_0 e_1$$

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.

## 5.4 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 pattern substitution 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 pattern substitution from the splitting tree. We can deduce from these a permutation of contexts 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  $\Delta$ , 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<sup>6</sup>. No such problems arise when using well-founded recursion instead.

## 6 EXAMPLE: A DEPENDENTLY-TYPED REFLEXIVE TACTIC

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

<sup>6</sup>Not anonymized: <https://github.com/coq/coq/pull/920>

1128 original solution used an earlier version of EQUATIONS already but not all its features which we  
1129 present here.

1130 The goal of this exercise was to develop a reflexive tactic for proving boolean tautologies, by  
1131 reflecting them into the set of multivariate polynomials over  $\mathbb{Z}$ . The complete annotated proof  
1132 script is available in `polynomials.v`. We encourage readers to follow the rest of the section by  
1133 running it: this section is directly extracted from it and does not explain the proofs in much detail.

1134  
1135 *6.0.1 Multivariate polynomials.* Using an indexed inductive type, we ensure that polynomials of  
1136  $\mathbb{Z}[(X_i)_{i \in \mathbb{N}}]$  have a unique representation. The first index indicates that the polynomial is null. The  
1137 second index gives the number of variables  $i$ .

```
1138 Inductive poly : bool → nat → Type :=
1139 | poly_z : poly true 0 | poly_c (z : Z) : IsNZ z → poly false 0
1140 | poly_l {n b} (q : poly b n) : poly b (S n)
1141 | poly_s {n b} (p : poly b n) (q : poly false (S n)) : poly false (S n).
```

- 1142 • `poly_z` represents the null polynomial.
- 1143 • `poly_c c` represents the constant polynomial  $c$  where  $c$  is non-zero (i.e. has a proof of `IsNZ c`).
- 1144 • `poly_l n q` represents the injection of  $q$ , a polynomial on  $n$  variables, as a polynomial on  $n+1$   
1145 variables.
- 1146 • Finally, `poly_s p q : poly _ (S n)` represents  $p + X_n * q$  where  $p$  is a polynomial on  $n$  variables  
1147 while  $q$  is a polynomial on  $n + 1$  variables, and the multiplication is not trivial as  $q$  is non-null.

1148 These indices enforce a canonical representation by ordering the multiplications of the variables.  
1149 A similar encoding is actually used in the `ring` tactic of `Coq`.

1151 **Derive Signature NoConfusion Subterm for poly.**

1152 In addition to the usual eliminators of the inductive type generated by `Coq`, we automatically  
1153 derive a few constructions on this `poly` datatype, and the `mono` datatype that follows, that will be  
1154 used by the `Equations` command:

- 1155 • Its **Signature**: as described earlier (§4.2), this is the packing of a polynomial with its two  
1156 indices, a boolean and a natural number in this case.
- 1157 • Its **NoConfusion** property used to simplify equalities between constructors of the `poly` type  
1158 (equation 6).
- 1159 • Finally, its **Subterm** relation, to be used when performing well-founded recursion on `poly`.

1161 *6.0.2 Monomials.* Monomials represent parts of polynomials, and one can compute the coeffi-  
1162 cient constant by which each monomial is multiplied in a given polynom. One can see this datatype  
1163 as a type of paths in the tree structure of polynomials. Again, the index of a `mono` gives the number  
1164 of its variables.

```
1165 Inductive mono : nat → Type := mono_c : mono 0 | mono_l : ∀ {n}, mono n → mono (S n)
1166 | mono_s : ∀ {n}, mono (S n) → mono (S n).
```

1168 Our first interesting definition computes the coefficient in  $\mathbb{Z}$  by which a monomial  $m$  is multiplied  
1169 in a polynomial  $p$ .

```
1170 Equations get_coef {n} (m : mono n) {b} (p : poly b n) : Z
1171 := get_coef m p by rec (pack m) mono_subterm :=
1172 get_coef mono_c poly_z := 0%Z; get_coef mono_c (poly_c z _) := z;
1173 get_coef (mono_l m) (poly_l p) := get_coef m p;
1174 get_coef (mono_l m) (poly_s p _) := get_coef m p;
1175 get_coef (mono_s m) (poly_l _) := 0%Z; get_coef (mono_s m) (poly_s p q) := get_coef m q.
```

1176



1177 The definition can be done using either the usual structural recursion of `Coq` or well-founded  
 1178 recursion. If we use structural recursion however, the guardness check will not be able to verify the  
 1179 automatically generated proof that the function respects its graph, as it involves too much rewriting  
 1180 due to dependent pattern-matching. We could prove it using a dependent induction instead of using  
 1181 the raw fixpoint combinator as the recursion is on direct subterms of the monomial, but in general  
 1182 it could be arbitrarily complicated, so we present a version allowing deep pattern-matching and  
 1183 recursion. Note that this means we lose the definitional behavior of `get_coef` during proofs on open  
 1184 terms, but this can advantageously be replaced using explicit `rewrite` calls, providing much more  
 1185 control over simplification than the reduction tactics, especially in presence of recursive functions.  
 1186 The `get_coef` function still uses no axioms, so it can be used to compute as part of a reflexive tactic  
 1187 for example.

1188 We want to do recursion on the (dependent) `m : mono n` argument, using the derived `mono_subterm`  
 1189 relation, which expects an element in the signature of `mono`, `{ n : nat & mono n }`, so we use `pack`  
 1190 `m` to lift `m` into its signature type (`pack` is just an abbreviation for the `signature_pack` overloaded  
 1191 constant defined in §4.2).

1192 The rest of the definition is standard: to fetch a monomial coefficient, we simultaneously pattern-  
 1193 match on the monomial and polynomial. Note that many cases are impossible due to the invariants  
 1194 enforced in `poly` and `mono`. For example `mono_c` can only match polynomials built from `poly_z` or  
 1195 `poly_c`, etc.

1196 **6.0.3 Two detailed proofs.** The monomial decomposition is actually a complete characterization  
 1197 of a polynomial: two polynomials with the same coefficients for every monomial are the same.

1199 **Theorem** `get_coef_eq {n} b1 b2 (p1 : poly b1 n) (p2 : poly b2 n) :`  
 1200 `(∀ (m : mono n), get_coef m p1 = get_coef m p2) → (b1 , p1) =_{ null & poly null n } (b2 , p2).`  
 1201 **Proof with** `(simp get_coef in *; auto).`

1202 Throughout the proof, we use the `simp` tactic defined by `EQUATIONS` which is a wrapper around  
 1203 `autorewrite` using the hint database associated to the constant `get_coef`: the database contains  
 1204 the defining equations of `get_coef` as rewrite rules that can be used to simplify calls to `get_coef` in  
 1205 the goal.

1206 `intros Hcoef. induction p1 as [ | z Hz | n b p1 | n b p1 IHp q1 IHq ] in b2, p2, Hcoef ⊢ ×.`  
 1207 `1-2: dependent elimination p2 as [poly_z | poly_c z i].`  
 1208 `5-6: dependent elimination p2 as [poly_l n b' p2 | poly_s n b' p2 q2].`

1210 We first do an induction on `p1` and then eliminate (dependently) `p2`. The patterns for the  
 1211 dependent elimination tactic differ from usual Coq patterns as they mention the constructors  
 1212 that should be matched in each branch. The first two branches need to consider variable-closed  
 1213 `p2`s while the next two branches (subgoals 5 and 6 after the first elimination) have `p2 : poly _ (S n)`,  
 1214 hence the `poly_l` and `poly_s` patterns. The elided rest of the tactic solves simple subgoals.

1215 We now focus on the case for `poly_l` on both sides. After some simplifications of the induction  
 1216 hypothesis using the `Hcoef` hypothesis, we get to the following goal:

1217 `(b, b' : bool) (n : nat) (p1 : poly b n) (p2 : poly b' n) IHp1 : (b; p1) = (b'; p2)`  
 1218 `=====`  
 1219 `(b; poly_l p1) = (b'; poly_l p2)`

1221 The `IHp1` hypothesis, as a general equality between dependent pairs can again be eliminated  
 1222 dependently to substitute `b'` by `b` and `p2` by `p1` simultaneously, using dependent `elimination IHp1`  
 1223 `as [eq_refl]`, leaving us with a trivial subgoal. The next step is to give an evaluation semantics to  
 1224 polynomials. We program `eval p v` where `v` is a valuation in `Z` for all the variables in `p : poly _ n`.

1225

```

1226 Equations eval {n} {b} (p : poly b n) (v : vector Z n) : Z
1227   := eval p v by rec (pack p) poly_subterm :=
1228   eval poly_z nil := 0%Z; eval (poly_c z _) nil := z;
1229   eval (poly_l p) (cons _ _ xs) := eval p xs;
1230   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. The interest of this representation is its canonicity: two polynomials having the same value for any valuation are syntactically equal. To show this, we first need to prove that evaluating a null polynomial always computes to 0, whichever valuation is used. Conversely we also show that every non-null polynomial yields a non-zero value for some evaluation. The first lemma 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.

```

1231
1232
1233
1234
1235
1236
1237
1238
1239 Lemma poly_z_eval {n} (p : poly true n) v : eval p v = 0%Z.
1240 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  $p = \text{poly\_z}$  and  $\text{eval } q \ v = 0$  for the `poly_l` recursive constructor, in which case the conclusion directly follows from the induction hypothesis corresponding to the recursive call. The second subgoal is hence discharged with an assumption call. The second lemma `poly_nz_eval` proves:  $\forall n (p : \text{poly false } n), \exists v, \text{IsNZ } (\text{eval } p \ v)$ . To show that polynomials having the same value are syntactically equal, we first establish a lemma showing that if the addition of a polynomial and the negation of another is null then they are syntactically equal:

```

1241
1242
1243
1244
1245
1246
1247
1248
1249 Lemma poly_diff_z_eq {n} {b1} (p1 : poly b1 n) {b2} (p2 : poly b2 n) :
1250   (plus p1 (poly_neg p2)).1 = true → (- , p1) =_{null & poly null n} (- , p2).
1251

```

From this we can deduce that two polynomials with the same values are syntactically equal:

```

1252
1253
1254 Theorem poly_eval_eq {n} {b1} (p1 : poly b1 n) {b2} (p2 : poly b2 n) :
1255   (∀ v, eval p1 v = eval p2 v) → (b1, p1) =_{null & poly null n} (b2, p2).

```

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.

## 7 RELATED AND FUTURE WORK

In [15] and [30], a specific `fcomp` constant associated to a definition `f` was used to keep track of the type of the “programming problem”, that is the current refinement of patterns, to express elaboration steps and communicate with tactics. This introduces unnecessary dependencies as all arguments of the function are considered dependent in its return type, which results in many uses of `K` that can actually be avoided. In particular it prevents the optimizations of section 5.

Cockx and Devriese [11] present an improvement on the simplification of unification constraints for indexed datatypes avoiding more uses of `K`. We reproduced its proof in `Coq` and are looking at ways to integrate it during simplification. In private communication with Cockx, it turns out that a presentation of inductive types with so-called “protestant” inductive types, that is inductives with parameters only, using an equivalent encoding of indices with equalities in the constructor (which is how GADTs are compiled in functional languages usually), would allow our current compilation scheme to enjoy the same benefits. We leave a careful study of this issue to future work.

1274

1275 The technique of small inversions [25] is an alternative way to implement dependent eliminations,  
1276 that is restricted to linear cases and discriminable indexes. We could benefit from integrating it in  
1277 the compilation scheme to produce simpler proof terms in these cases.

1278 The equation compiler of LEAN [6] is similar to our system. As mentioned in the introduction,  
1279 pattern-matching compilation is simplified by using definitional proof-irrelevance. It also supports  
1280 well-founded recursion using a fixpoint combinator and inference of the well-founded relation (op.  
1281 cit. §8.4), but does not derive the corresponding elimination principle, only the unfolding equation.

1282 The FUNCTION package [7] of COQ also provides support for deriving an eliminator from a  
1283 well-founded definition and also automatically proves the completeness of the graph that we  
1284 currently lack. It is also clever about handling overlapping or default branches in pattern-matchings,  
1285 providing a graph that corresponds more closely to the shape of the definition entered by the user.  
1286 We leave to future work a refinement of the splitting tree structure to handle a similar optimization  
1287 when typing constraints allow it. The main advantage of EQUATIONS is that it allows definitions by  
1288 dependent pattern-matching and recursion schemes that FUNCTION cannot handle.

1289 The PROGRAM package [29] of COQ also allows definition by pattern-matching on dependent  
1290 types and well-founded recursion. It implements pattern-matching compilation using the usual  
1291 generalization-by-equalities pattern, generalizing the branches of a match by an heterogeneous  
1292 equality between the pattern and the discrimineed. It is limited to heterogeneous equality which  
1293 implicitly requires uniqueness of identity proofs on the type universe (compared to UIP on specific  
1294 types like `nat`), hence the definitions never compute and are not compatible with a univalent  
1295 universe. It handles “shallow” pattern-matching on a single object at a time and does not provide  
1296 any simplification engine, making it rather limited in the scope of definitions it can handle. The  
1297 well-founded recursion support is also limited: only the definition of a well-founded fixpoint is  
1298 supported, no equations, unfolding lemmas or elimination principles are generated.

1299 The treatment of nested recursive definitions is close to the one implemented by the `Nfix` package  
1300 of Lescuyer [18] which provides a similar notation, except no elimination principle was generated  
1301 in this case. LEAN handles nested inductive types by rewriting inductive definitions using an  
1302 isomorphism with a mutual inductive definitions, resulting in back and forth translations. As far  
1303 as we know, EQUATIONS is the first tool to provide support for defining and reasoning on nested  
1304 well-founded fixpoint definitions on inductive families.

1305 In future work, we plan to implement a translation to lift CIC terms into splitting trees, so that  
1306 the lemma generation phase of EQUATIONS can be reused to generate lemmas for existing COQ  
1307 definitions, and to improve support for nested recursion by reconigizing recursive calls through  
1308 constants like `List.map`. We also hope to extend the recursion support of EQUATIONS to co-patterns  
1309 and the reduction of productivity to well-founded recursion pionnered by Abel & Pientka [2].  
1310 Finally, given the proximity of EQUATIONS and HASKELL definitions, EQUATIONS could provide a  
1311 better frontend to the `hs-to-coq` tool [31] for the verification of HASKELL programs in COQ.

## 1312 CONCLUSION

1314 We presented a full-featured definitional extension of COQ that makes the definition and reasoning  
1315 on programs using dependent pattern-matching and complex recursion schemes efficient and  
1316 effective, without sacrificing assurance. The source language and proof generation facilities of  
1317 EQUATIONS support both `with` and `where` clauses, encompassing mutual, nested and well-founded  
1318 recursive definitions, which provides a comfortable environment for reasoning on recursive func-  
1319 tions. Our central technical contribution is an independent, optimized dependent pattern-matching  
1320 compiler, based on simplification of equalities of telescopes, that can be reused to implement a ro-  
1321 bust dependent elimination tactic. We hope to have demonstrated its usefulness on a medium-sized  
1322 example.

1323

## REFERENCES

- [1] Andreas Abel. 2006. Semi-continuous Sized Types and Termination. In *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25–29, 2006, Proceedings (Lecture Notes in Computer Science)*, Zoltán Ésik (Ed.), Vol. 4207. Springer, 72–88. [https://doi.org/10.1007/11874683\\_5](https://doi.org/10.1007/11874683_5)
- [2] Andreas Abel and Brigitte Pientka. 2016. Well-founded recursion with copatterns and sized types. *J. Funct. Program.* 26 (2016), e2. <https://doi.org/10.1017/S0956796816000022>
- [3] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational Equality, Now!. In *PLPV'07*. Freiburg, Germany.
- [4] Ana Bove and Alexander Krauss and Matthieu Sozeau. 2015. Partiality and recursion in interactive theorem provers – an overview. *Mathematical Structures in Computer Science FirstView* (2 2015), 1–51. <https://doi.org/10.1017/S0960129514000115>
- [5] 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*. Paris, France. <http://conf.researchr.org/event/CoqPL-2017/main-certicoq-a-verified-compiler-for-coq>
- [6] 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](https://leanprover.github.io/reference/lean_reference.pdf).
- [7] 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](https://doi.org/10.1007/11737414_9)
- [8] Bauer, A. and Gross, J. and LeFanu Lumsdaine, P. and Shulman, M. and Sozeau, M. and Spitters, B. 2016. The HoTT Library: A formalization of homotopy type theory in Coq. *ArXiv e-prints* (Oct. 2016). arXiv:cs.LO/1610.04591 <https://arxiv.org/abs/1610.04591> Accepted at CPP'17.
- [9] Lars Birkedal, Ales Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2016. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France (LIPIcs)*, Jean-Marc Talbot and Laurent Regnier (Eds.), Vol. 62. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 23:1–23:17. <https://doi.org/10.4230/LIPIcs.CSL.2016.23>
- [10] Jesper Cockx. 2017. *Dependent Pattern Matching and Proof-Relevant Unification*. Ph.D. Dissertation. Katholieke Universiteit Leuven, Belgium. <https://lirias.kuleuven.be/handle/123456789/583556>
- [11] 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>
- [12] 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>
- [13] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2016. Cubical Type Theory: a constructive interpretation of the univalence axiom. *CoRR* abs/1611.02108 (2016). arXiv:1611.02108 <http://arxiv.org/abs/1611.02108>
- [14] Thierry Coquand. 1992. Pattern Matching with Dependent Types. (1992). <http://www.cs.chalmers.se/~coquand/pattern.ps> Proceedings of the Workshop on Logical Frameworks.
- [15] 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>
- [16] 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>
- [17] Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. 2013. Generalizations of Hedberg’s Theorem. In *Typed Lambda Calculi and Applications*, Masahito Hasegawa (Ed.). Lecture Notes in Computer Science, Vol. 7941. Springer Berlin Heidelberg, 173–188. [https://doi.org/10.1007/978-3-642-38946-7\\_14](https://doi.org/10.1007/978-3-642-38946-7_14)
- [18] Stéphane Lescuyer. 2011. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq*. Ph.D. Dissertation. University of Paris-Sud, Orsay, France. <https://tel.archives-ouvertes.fr/tel-00713668>
- [19] Peter LeFanu Lumsdaine. 2010. Weak omega-categories from intensional type theory. *Logical Methods in Computer Science* 6, 3 (2010).
- [20] 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.LFMTPT'15>
- [21] Per Martin-Löf. 1984. *Intuitionistic type theory*. Studies in Proof Theory, Vol. 1. Bibliopolis. iv+91 pages.
- [22] Conor McBride. 1999. *Independently Typed Functional Programs and Their Proofs*. Ph.D. Dissertation. University of Edinburgh. <http://citeseer.ist.psu.edu/mcbride99independently.html>

- 1373 [23] Conor McBride. 2000. Elimination with a Motive. In *TYPES (Lecture Notes in Computer Science)*, Paul Callaghan,  
1374 Zhaohui Luo, James McKinna, and Robert Pollack (Eds.), Vol. 2277. Springer, 197–216.
- 1375 [24] Conor McBride, Healfdene Goguen, and James McKinna. 2004. A Few Constructions on Constructors. *Types for Proofs*  
1376 *and Programs* (2004), 186–200. [https://doi.org/10.1007/11617990\\_12](https://doi.org/10.1007/11617990_12)
- 1377 [25] Jean-François Monin and Xiaomu Shi. 2013. *Handcrafted Inversions Made Operational on Operational Semantics*. Springer  
1378 Berlin Heidelberg, Berlin, Heidelberg, 338–353. [https://doi.org/10.1007/978-3-642-39634-2\\_25](https://doi.org/10.1007/978-3-642-39634-2_25)
- 1379 [26] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation.  
1380 Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.  
1381 <http://www.cs.chalmers.se/~ulfn/papers/thesis.html>
- 1382 [27] Christine Paulin-Mohring. 1993. Inductive Definitions in the System Coq - Rules and Properties. In *Typed Lambda*  
1383 *Calculi and Applications (Lecture Notes in Computer Science)*, Vol. 664. Springer-Verlag, 328–345. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.32.5387&rep=rep1&type=pdf>
- 1384 [28] Álvaro Pelayo and Michael A. Warren. 2012. Homotopy type theory and Voevodsky’s univalent foundations. (10 2012).  
1385 arXiv:1210.5658 <http://arxiv.org/abs/1210.5658>
- 1386 [29] Matthieu Sozeau. 2007. Program-ing Finger Trees in Coq. In *ICFP’07*. ACM Press, Freiburg, Germany, 13–24. <https://doi.org/10.1145/1291151.1291156>
- 1387 [30] Matthieu Sozeau. 2010. Equations: A Dependent Pattern-Matching Compiler. In *First International Conference on*  
1388 *Interactive Theorem Proving*. Springer.
- 1389 [31] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable  
1390 Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los*  
1391 *Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 14–27. <https://doi.org/10.1145/3167092>
- 1392 [32] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations for Mathematics*. Institute  
1393 for Advanced Study. <http://homotopytypetheory.org/book>
- 1394 [33] Benno van den Berg and Richard Garner. 2011. Types are weak  $\omega$ -groupoids. *Proceedings of the London Mathematical*  
1395 *Society* 102, 2 (2011), 370–394. <https://doi.org/10.1112/plms/pdq026>

1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421