Equations reloaded

Cyprien Mangin, Matthieu Sozeau

Equations Reloaded

A definitional extension of Coq for dependent and recursive function definitions

CYPRIEN MANGIN, Inria Paris & IRIF - Université Paris 7 Diderot, France
MATTHIEU SOZEAU, Inria Paris & IRIF - Université Paris 7 Diderot, France

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. It additionally derives useful proof principles for demonstrating properties about them. We present a general design and implementation that provides a robust and expressive function definition package 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 versions of dependent elimination tactics. We introduce verified optimizations of the simplifier that allow generating smaller and simpler Equations definitions and proof terms for these tactics in general.

Additional Key Words and Phrases: dependent pattern-matching, proof assistants, recursion

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, well-founded recursion 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 ordinary Coq 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 separately as in [10] and 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

Authors' addresses: Cyprien Mangin, Inria Paris & IRIF - Université Paris 7 Diderot, France, cyprien.mangin@m4x.org; Matthieu Sozeau, Inria Paris & IRIF - Université Paris 7 Diderot, France, matthieu.sozeau@inria.fr.

2018. XXXX-XXXX/2018/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to the *n* argument of the fs constructor, it *must* be written as an inaccessible pattern or a wildcard as in the first clause[1]. 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 case 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[2]:

```
Check nth_equation_1 : ∀ (A : Type) (f : fin 0), ImpossibleCall (nth nil f).
Check nth_equation_2 : ∀ (A : Type) (n : nat) (a : A) (v : vector A n), nth (cons a v) (fz n) = a.
Check nth_equation_3 : ∀ A n a v f, nth (cons a v) (fs n 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. The two following equations reflect the computational behavior of nth and are definitional equalities: they can be proven using reduction only. Finally, the eliminator nth_elim provides an abstract view on nth:

```
Check nth_elim : ∀ P : ∀ (A : Type) (n : nat), vector A n → fin n → A → Prop,
    (∀ A n a v, P A (S n) (cons a v) (fz n) a) →
    (∀ 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)) →
    ∀ A n v f, P A n v f (nth v f).
```

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 gain trust in a given proof assistant's results. This is essentially the so-called de Bruijn principle: proofs should be checkable using a relatively small proof checker. 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 contradictory (§1.2).

Our thesis is that 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

[1] 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

[2] 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

of a high-level abstract view on function definitions by pattern-matching and recursion. The principle we follow is to maintain the abstraction given by the equational presentation of programs, avoiding the leakage of details of the translation. The following sections explain our design choices to achieve this.

1.1 The identity type

First of let us recall the identity type of type theory, also know as propositional equality. It is the central inductive family used in this work and the one whose structure is modified by axioms such as UIP or Univalence:

Inductive eq {A : Type} (x : A) : $A \longrightarrow$ Prop := eq_refl : $x = x$ where "x = y" := (eq x y) : *type_scope*.

Equality is an equivalence relation and its elimination principle eq_rect_dep is a dependent version of the Leibniz substitution principle, coined the J rule in type theory jargon:

eq_rect_dep : $\forall$ A x (P : $\forall$ y : A, $x = y \longrightarrow$ Type) (p : P x (eq_refl x)) (y : A) (e : $x = y$), P y e

Informally, this principle states that to prove a goal $P\ y\ e$ 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 eq_refl x : $x = x$. So, only the case where y and x are the same need be considered.

The computation rule of eq_rect_dep is reducing to its single arm p : P x (eq_refl x) when e is eq_refl x. 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 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. Both systems used the UIP principle from the start. Uniqueness of Identity Proofs states that all equality proofs at *any* type are equal.

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

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 iterated sigma types, potentially avoiding the use of UIP, would be possible as well). Agda implements by default this notion of dependent pattern-matching, assuming the UIP principle.

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 eq_refl. UIP and K are hence used interchangeably in the literature.

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

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 equal to equivalence of types (a higher-dimensional variant of isomorphism):

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

Informally, in Homotopy Type Theory, one is interested in the higher-dimensional structure of types and their equality types, which are shown to form weak ∞-groupoids [19, 33]. That is, in homotopy type theory, it is possible to define and manipulate types whose equality type is not just inhabited or uninhabited, but has actual structure and relevance. 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, equal only to themselves. Hence, UIP implies that the higher-dimensional structure of identity at any type is trivial. As an example, already at the level of types, one can build two distinct equivalences from booleans to booleans, the identity and the negation. The axiom allows deriving that the equality of types $\mathbb{B} = \mathbb{B}$ has two distinct elements, these two equivalences, contradicting UIP. One can however still show using a result of Hedberg [17] that usual data structures with decidable equality like natural numbers enjoy UIP, provably.

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

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

1.3 UIP versus Univalence

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

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

1.4 Computational Behavior

Using a definitional translation, compilation of dependent pattern-matching introduces many proof-manipulations to the implementations of definitions. It is actually the point of this elaboration to relieve the user from having to witness reasoning on the theory of equality, constructors and indexed inductive types to implement definitions by dependent pattern-matching. In section 5 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 UIP proof or an axiom. In case the compilation relies on a proof of UIP (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 UIP 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 on the inductive structure of terms. There are basically two ways to present recursion on inductive families in dependent 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 *schema* 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 reusing 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 guardedness check was too permissive. It considered pattern-matching (e.g. `match`) terms as subterms iff 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, based on syntax as well, 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 (§5.4 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 working more comfortably with nested inductive definitions than in vanilla Coq.
- A cleaner elaboration of Equations definitions, designed to avoid the use of a construction of Goguen et al [15] which was forcing unnecessary applications of UIP. The elaborator naturally handles mutual and nested fixpoint definitions. We also avoid the use of proof-irrelevance for proving unfolding lemmas of recursive definitions. The system is parameterized such that it can be used in a setting where the equality is in `Prop` and UIP holds or with a proof-relevant equality supporting univalence.
- 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[3]. It has been tested on a variety of examples, including a proof of strong normalization for predicative System F [20] and a reflexive

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

tactic for deciding equality of polynomials, available on the website. 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.

Structure of the paper. We will present the contributions in stages corresponding to the compiler's stages: in §2 we present the user-level features of Equations in a tutorial way, focusing on the novel treatment of structural and well-founded recursive definitions. In §3 we introduce the formal source language of Equations and present its architecture, as well as a dependent elimination tactic derived from it. In §4 we recall the theory of dependent pattern-matching compilation using equality of sigma types and present our ML compiler. In §5 we develop an optimization of simplification that can reduce term size and complexity of definitions. Finally we review related work in §6 and conclude.

2 FROM STRUCTURAL TO NESTED WELL-FOUNDED RECURSION

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

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

2.1 Mutual structural recursion

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

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

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

```
Equations lift_fin {n} (k : nat) (f : fin n) : fin (S n) :=
  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).
```

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

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

This representation lends itself naturally to a definition of parallel substitution. To define it at the case of abstraction, one must explain how to lift a substitution of k variables to a substitution of k+1 variables by preserving the $0th$ variable and lifting the result of the substitution otherwise. This pattern-matching involves the no-confusion principle on the natural number index (constructors are distinct and injective), but does not require UIP.

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Nested structural recursion

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

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

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

We can define lifting on this datatype like in the previous example. For definitions of fixpoints on nested mutual inductive types, Equations allows users to factorize the nested fixpoint definitions in toplevel where clauses, so that multiple calls to the nested function can refer to the same function. Below, subst_terms is called multiple times in subst_term, and of course it recursively calls itself and subst_term.

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

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

2.2.2 Reasoning. Remark that our definition of subst_terms is equivalent to a call to map on lists. Equations currently needs the "expanded" version to properly recognize recursive calls, but one can readily add this equation to the subst_term rewrite database gathering the defining equations of subst_term to abstract away from this detail:

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

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

```
Check subst_term_elim : (*   Predicates/Motives   *)
   ∀ (P : ∀ k : nat, (fin (S k) → term k) → term (S k) → term k → Prop)
     (P0 : ∀ k : nat, (fin (S k) → term k) → list (term (S k)) → list (term k) → Prop),
   (*   Obligations/Methods   *)
   (∀ k u (f : fin (S k)), P k u (Var f) (u f)) →
   (∀ k u (t : term (S (S k))), P (S k) (extend_var u) t (subst_term (extend_var u) t) →
       P k u (Lam t) (Lam (subst_term (extend_var u) t))) →
   (∀ k u (t0 : term (S k)) (l : list (term (S k))), P k u t0 (subst_term u t0) →
       P0 k u l (subst_terms u l) → P k u (App t0 l) (App (subst_term u t0) (subst_terms u l))) →
   (∀ k u (id0 : nat) (l0 : list (term (S k))), P0 k u l0 (subst_terms u l0) →
       P k u (Meta id0 l0) (Meta id0 (subst_terms u l0))) → (∀ k u, P0 k u [] []) →
   (∀ k u (t : term (S k)) (l : list (term (S k))), P k u t (subst_term u t) →
       P0 k u l (subst_terms u l) → P0 k u (t :: l) (subst_term u t :: subst_terms u l)) →
```

```
(* Conclusion/Target*)
(∀ k u t, P k u t (subst_term u t)) ∧ (∀ k u t, P0 k u t (subst_terms u t)).
```

One may want to specialize P0 with Forall2 P to recover a map-like elimination principle. From subst_term_elim, one can indeed automatically derive another eliminator with a single predicate P, filling the last two methods for the recursive definition on lists of terms and assuming proofs of Forall2 P l (subst_terms u l) in the induction hypotheses of App and Meta instead.

2.3 Well-founded recursion

Issues with the guardedness check and a way out. While the implementation of the guard condition has been adapted over the years to try and allow as many safe cases as possible, it is still obviously an approximation and may fail in some legitimate cases. This is aggravated by the fact that the Equations compiler introduces rewritings with propositional equalities, making the job of the guard condition checker that much more difficult. In many cases everything works as expected, but sometimes it (apparently) diverges because it must unfold definitions or it fails to track the subterm relation correctly in the term due to rewritings (applications of J) or lack of *commutative cuts*. Indeed a problematic case appears when a recursive subterm is abstracted in branches of a pattern-matching for example, a typical situation where β and ι redexes are mixed and do not commute. The syntactic check of Coq cannot check the typing constraints on commutation that would allow the recursive definitions to pass, and it is notoriously difficult to provide a corresponding explanation to the user, as it requires understanding the guard condition and its failure on the compiled definition. To relieve the user from such complications and avoid the syntactic guardedness check entirely, Equations provides an automatic derivation of the well-foundedness of the Subterm relation on inductive families. It can be used to explicitly show why a structurally recursive definition is correct, using logical reasoning on the derived transitive closure of the strict subterm relation.

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

2.3.1 Nested well-founded recursion. To demonstrate nested well-founded recursive definitions, we take a well-known example from the literature: rose trees. We will define a recursive function gathering the elements in a rose tree in an efficient way, using nested well-founded recursion instead of the guardedness check of Coq. The rose trees are defined as trees whose nodes contain lists of trees, i.e. forests.

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

This is a nested inductive type we can measure assuming a list_size function for measuring lists. Here we use the usual guardedness check of Coq that is able to unfold the definition of list_size to check that this definition is terminating.

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

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

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

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

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

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.
  apply (elements_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 H2, H1, 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 definition, 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:

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

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

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 f and f. However the unfolding of FixWf is *not* convertible to 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 Coq is proof-irrelevant, a folklore result. From this, it is then possible to prove a general unfolding lemma for 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 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 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, a refinement of case trees (§3), then compiles splitting trees to terms (§4.2 & §4.3) which we will optimize in §5. From splitting trees, the defining equations and the graph of the function(s) can be generically derived, for lack of space we do not detail this here. One can consult [30] for details of that construction, which 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 Δ, in the order of declarations, and also to denote lists in general. An *arity* is a type of the form $\forall\, \Gamma, s$ where Γ 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 Coq versions featuring typical ambiguity and universe polymorphism, which we use to formalize our constructions. We consider inductive families to be defined in a (elided) global context by an arity $\mathsf{I} : \forall\, \Delta, s$ and constructors $\overline{\mathsf{I}_i : \forall\, \Gamma_i, \mathsf{I}\ \overline{t_i}}$ (where $\Gamma_i \vdash \overline{t_i} : \Delta$). 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, hence we do not treat them formally here. The dependent sum / sigma type is written $\Sigma x : \tau.\tau'$, its introduction form is $(\_, \_)$ and its projections are in post-fix notation $\_.1 : \Sigma x : \tau.\tau' \longrightarrow \tau$ and $\_.2 : \forall s : (\Sigma x : \tau.\tau').\tau'[s.1]$. The compilation process starts from a signature and a list of clauses given by the user, constructed from the grammar given in figure 1.

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

Fig. 1. Definitions and user clauses

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

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

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

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

3.2 Searching for a covering

The goal of the compiler is to produce a proof that the user clauses form an exhaustive covering of the signature, compiling away nested pattern-matchings to simple case splits. As we have multiple patterns to consider and allow overlapping clauses, there may be more than one way to order the case splits to achieve the same results. We use inaccessible patterns (noted $?(t)$) as in Agda to help recover a sense of what needs to be destructed and what is statically known to have a particular value, but overlapping clauses force the compilation to be phrased as a search procedure. As usual, we recover a deterministic semantics using a first-match rule when two clauses overlap. The search for a covering works by gradually refining a *pattern substitution* $\Delta \vdash \overline{p} : \Gamma$ and building a splitting tree. A pattern substitution (fig. 2), is a substitution from Δ to Γ, associating to each variable in Γ a pattern p typable in Δ. We start the search with the problem $\Gamma \vdash \overline{\Gamma} : \Gamma$, i.e. the identity substitution

`program`	$prog$	$::=$	$(\ell_p, \Gamma, \tau, rec?, spl)$
`recursion`	rec	$::=$	$\texttt{wf}(t, R) \mid \texttt{struct}\ x$
`pattern substitution`	c	$::=$	$\Delta \vdash \overline{p} : \Gamma$
`pattern`	p	$::=$	$x \mid \mathsf{C}\ \overline{p} \mid ?(t)$
`splitting`	spl	$::=$	$\mathsf{Split}(c, x, ((spl)?)^n) \ \mid\ \mathsf{Compute}(c, t, prog^*)$

Fig. 2. Grammar of programs, splitting trees and pattern substitutions

on Γ, and the list of user clauses. At each point during covering, we can compute the expected target type of the current subprogram by applying this substitution to its initially declared type τ.

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 $\mathsf{Split}(\Delta \vdash \overline{p} : \Gamma, x, (s?)^n)$ node denoting that the variable x is an object of an inductive type with n constructors and that splitting it in context Δ will generate n subgoals which are covered by the optional subcoverings s. When the type of x does not unify with a particular constructor's type the corresponding splitting is empty. Otherwise the substitution built by unification determines the pattern substitution used in each of the subcoverings.
- A $\mathsf{Compute}(\Delta \vdash \overline{p} : \Gamma, t, \overline{w})$ node, denoting a right-hand side whose definition is t (of type $\tau[\overline{p}]$) under some set of auxiliary local definitions $\overline{w}$. Both `with` and `where` clauses are compiled this way. 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 $(\ell_p, \Gamma, \tau, rec?, s)$, the optional rec annotation describes its recursive structure.

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

As an example, the subst0 definition from section 2.1 performs case analysis on a term of type fin 1. In its splitting representation, the first branch computes a result while the other is impossible due to another inversion. Assuming $\Gamma = (t : \mathsf{term}\ 0)(f : \mathsf{fin}\ 1)$, the context of initial arguments, the program's representation is: $(\mathsf{subst0}, \Gamma, \mathsf{term}\ 0, \mathsf{None}, s)$ where s is:

$$\mathsf{Split}(\Gamma \vdash t\ f : \Gamma)(f) \begin{bmatrix} \mathsf{Some}(\mathsf{Compute}(((t : \mathsf{term}\ 0) \vdash t\ (\mathsf{fz}\ 0) : \Gamma), t, \epsilon)); \\ \mathsf{Some}(\mathsf{Split}(((t : \mathsf{term}\ 0)(f : \mathsf{fin}\ 0) \vdash t\ (\mathsf{fs}\ 0\ f) : \Gamma), f, \begin{bmatrix} \mathsf{None}; \\ \mathsf{None} \end{bmatrix}) \end{bmatrix}$$

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

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 $\mathsf{I}\ \bar{t}$ from context Γ. 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 Γ, 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 $\{1 \ldots n\}$.

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

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

We will need a 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 $\forall\ \Delta,\ \mathsf{I}\ \Delta$ (any arity in general) a corresponding closed type by wrapping the indices Δ in a dependent sum and both the indices and the inductive type in another dependent sum.

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

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

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

$$\mathsf{signature\_pack} : \forall\Delta(x : \mathsf{I}\ \overline{\Delta}), \Sigma i : \Sigma(\Delta), I\ \overline{\Sigma}(\Delta, i) := \lambda(\overline{\Delta} : \Delta)(x : \mathsf{I}\ \overline{\Delta}), (\sigma(\Delta)(\overline{\Delta}), x)$$

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:

$$(i_0, \mathsf{C}_0\ \overline{a_0}) =_{\bar{\mathsf{I}}} (i_1, \mathsf{C}_1\ \overline{a_1}) \quad \text{where } \forall j \in \{0, 1\}, \mathsf{C}_\mathsf{j}\ \overline{a_j} : I\ \overline{\Sigma}(\Delta, i_j) \tag{5}$$

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 : Type} (x : A) : ∀ {B : Type}, B → Prop := JMeq_refl : 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 `JMeq x y` between two terms in the same type requires to invoke an axiom equivalent to UIP, which we want to avoid. Therefore, we instead use telescopes and homogeneous equalities. However, contrary to the variant used by Cockx [11], we mainly make use of equalities of telescopes, instead of telescopes of equalities. Both are however equivalent, that is:

```
Theorem teleq_eqtel {A : Type} {B : A → Type} (x1 x2 : A) (y1 : B x1) (y2 : B x2) :
  {e : x1 = x2 & eq_rect y1 e = y2} ↔ (x1, y1) = (x2, y2).
```

On the equality (5), the tactic should either give us equalities between the arguments $\overline{a_0}$ and $\overline{a_1}$ (injectivity) that can be further simplified or deriving a contradiction if C_0 is different from

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 – Cockx [11] describes it in detail. We implement this construction as another `Derive` scheme in Coq. For any (computational) inductive type $\mathsf{I} : \forall\, \Gamma, \mathsf{Type}$, we can use `Derive NoConfusion for I` to derive an instance of the type class NoConfusionPackage $\bar{\mathsf{I}}$ that provides a proof of isomorphism of the two types:

$$\forall\, x\ y : \bar{\mathsf{I}}, \mathrm{NoConfusion}\ \bar{\mathsf{I}}\ x\ y \simeq x =_{\bar{\mathsf{I}}} y \tag{6}$$

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 $\overline{a_0}$ and $\overline{a_1}$. 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.

4.1.3 The logic interface. As we intend for Equations to be usable in different settings, using the usual proof-irrelevant equality or a univalent one, we aim to abstract the logic interface required by Equations to be able to perform elimination using any equality satisfying a minimal interface. As such, the user will be able choose between the usual equality in the impredicative `Prop` universe, a proof-relevant and universe polymorphic identity type like HoTT's path type, or even another custom equality type. It is still a work-in-progress in the implementation to be parametric enough, but the groundwork for it is done[4].

The simplifier used in Equations is mainly dealing with uses of the eliminator for equality, and equalities between constructors. We encapsulate the basic blocks that it needs in a few lemmas and classes that have to be provided for any equality type to be used with Equations. The exact interface can be found in the supplementary material (intf.mli). To summarize, we need an equality type with the usual constructor and eliminators and several lemmas about K, NoConfusion and equalities of telescopes. We also need most of the lemmas about equalities to preserve eq_refl (i.e. reduce to eq_refl when applied to eq_refl) to ensure the preservation of the computational behavior of definitions.

4.2 Compilation of a splitting tree

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

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

On the example of subst0 from section 2.1, the following term is compiled from the splitting tree:

```
λ (t : term 0) (f : fin 1),
  match f in (fin n) return (n = 1 → term 0) with
  | @fz n ⇒ apply_noConfusion (S n) 1 (λ H : n = 0, solution_left 0 t n H)
  | @fs n f0 ⇒ apply_noConfusion (S n) 1 (λ H : n = 0, solution_left 0
            (λ f1 : fin 0, subst0_obligation_1 t f1) n H f0)
  end eq_refl
```

Compilation inserts no-confusion and solution lemmas that perform rewritings so as to be able to typecheck the right-hand side (in the first branch) or derive a contradiction (in the second branch).

[4] See http://mattam82.github.io/Coq-Equations/examples/HoTT_light.html for an example from the HoTT library [8]

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

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

$$\Gamma, i : \Sigma(\Delta), x' : \mathsf{I}\ \overline{\Sigma}(\Delta, i) \vdash (i, x') = (\sigma(\Delta)(\bar{t}), x) \rightarrow \tau \tag{7}$$

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

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 (see [10] for such a presentation); instead, we choose here to show how it acts on a given goal, since we are directly manipulating terms. Each of these simplification steps apply under a certain context Γ which we do not write most of the time because it will not change. The arrow ⇒ 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.

Remove sigma $\boxed{\forall(e\ :\ (\mathtt{x},\ \mathtt{p})\ =\ (\mathtt{y},\ \mathtt{q})),\ \mathtt{P}\ e} \Rightarrow \boxed{\begin{array}{l}\forall(e'\ :\ \mathtt{x}\ =\ \mathtt{y})\ (e\ :\ \mathsf{rew}\ e'\ \mathtt{p}\ =\ \mathtt{q}),\\ \mathtt{P}\ (\mathsf{sigma\_eq}\ e'\ e)\end{array}}$

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 $\boxed{\forall(e\ :\ \mathtt{t}\ =\ \mathtt{t}),\ \mathtt{P}\ e} \Rightarrow \boxed{\mathtt{P}\ \mathsf{eq\_refl}}$

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

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

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

$$\Gamma \vdash \mathsf{cons}\ x\ v = \mathsf{cons}\ y\ w \longrightarrow \mathsf{vector}\ A\ n$$

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 $\boxed{\forall(e\ :\ \mathsf{C}\ \bar{t}\ =\ \mathsf{D}\ \bar{u}),\ \mathtt{P}\ e} \Rightarrow \boxed{\forall(e\ :\ (\overline{idx_t},\ \mathsf{C}\ \bar{t})\ =\ (\overline{idx_u},\ \mathsf{D}\ \bar{u})),\ \mathsf{ind\_pack\_inv}\ \mathtt{P}\ e}$

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 Cockx [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:

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

NoConfusion $\boxed{\forall (e\ :\ (\overline{idx_t},\ \mathsf{C}\ \overline{t}) = (\overline{idx_u},\ \mathsf{D}\ \overline{u})),\ \mathsf{P}\ e}$

$\Rightarrow$

- $\forall\ (e\ :\ \overline{t} = \overline{u}),\ \mathsf{P}\ (\mathsf{noConf\_inv}\ e)$ if C and D are the same constructor
- solved if C and D are distinct constructors

To implement this step, we use the NoConfusionPackage class that we are able to derive automatically (see section 4.1.2). The equality between $\overline{t}$ and $\overline{u}$ is, in general, an equality between telescopes, which will then be further simplified; when it is fully simplified, e and noConf_inv e will reduce to eq_refl.

Following the example, we get this goal:

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

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

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

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

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

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

Unpack $\boxed{\mathsf{ind\_pack\_inv}\ \ \mathsf{P}\ \ \mathsf{eq\_refl}} \Rightarrow \boxed{\mathsf{P}\ \ \mathsf{eq\_refl}}$

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

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

$$(A : \mathsf{Type})(n : \mathsf{nat})(y : A)(v : \mathsf{vector}\ A\ n) \vdash \mathsf{vector}\ A\ n$$

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

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

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

5 SMART CASE

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

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

$$\texttt{match}\ y\ \texttt{in}\ \mathsf{I}\ \overline{u}\ \texttt{return}\ \Delta \to \overline{t'} = \overline{u'} \to \mathsf{T}\ \texttt{with}\ \ldots\ \texttt{end}\ \Delta\ \mathsf{eq\_refl}$$

In this term, $(y : \mathsf{I}\ \overline{u})$ is a fresh variable that is introduced to generalize x and T is some type corresponding to the goal we want to prove. In the return type, $\overline{t'} = \overline{u'}$ is a telescopic equality between subsets of the telescopes $(\overline{t}, x)$ and $(\overline{u}, y)$. This telescopic equality will then need to be simplified in each branch of the `match` by using the simplifications steps from the previous section. Finally, Δ is a list of variables whose type we need to rewrite. In the rest of this section we will call 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 $\Gamma_0\ (x : \mathsf{I}\ \overline{t})\ \Gamma_1$, where $\overline{t} : \overline{\tau}$ 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.

$$\Gamma_0\ (x : \mathsf{I}\ \overline{t})\ \Gamma_1\ (\overline{u} : \overline{\tau})\ (y : \mathsf{I}\ \overline{u})\ (e : (\overline{t}, x) = (\overline{u}, y))$$

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

$$\Gamma_0\ (x : \mathsf{I}\ \overline{t})\ \Gamma_1\ (\overline{a} : \overline{T_i})\ (e : (\overline{t}, x) = (\overline{u_i}, \mathsf{C_i}\ \overline{a}))$$

(3) Simplify the equality e, effectively unifying x with $\mathsf{C_i}\ \overline{a}$ if it succeeds positively, or directly solving the goal if it succeeds negatively.

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

$$\texttt{match}\ y\ \texttt{in}\ \mathsf{I}\ \overline{u}\ \texttt{return}\ (\overline{t},\ x) = (\overline{u},\ y) \to \mathsf{T}\ \texttt{with}\ \ldots\ \texttt{end}\ \mathsf{eq\_refl}$$

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

To do so, we will start from the context after generalization:

$$\Gamma_0\ (x : \mathsf{I}\ \overline{t})\ \Gamma_1\ (\overline{u} : \overline{\tau})\ (y : \mathsf{I}\ \overline{u})\ (e : (\overline{t}, x) = (\overline{u}, y))$$

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

$$\Gamma_0\ (x : \mathsf{I}\ \overline{t})\ \Gamma_1\ (\overline{u} : \overline{\tau})\ (y : \mathsf{I}\ \overline{u})\ (e_1 : t_1 = u_1)\ e_2 \ldots e_n$$

We will then perform a ***Solution*** step on some of the equalities e_j that will be chosen according to two criteria explained later, while maintaining this general shape for the telescope:

$$\Gamma\ (\overline{u} : \overline{\tau})\ (y : \mathsf{I}\ \overline{u})\ \Delta\ (e : \overline{t'} = \overline{u'})$$

where e is a telescopic equality (possibly empty) which is some subset of the initial equality, and Δ 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:

$$\texttt{match } y \texttt{ in I } \overline{u} \texttt{ return } \Delta \rightarrow \overline{t'} = \overline{u'} \rightarrow \top \texttt{ with ... end } \Delta \texttt{ eq\_refl}$$

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 An example

Before explaining in details these two optimizations, we will show an example in which they apply and happen to reduce the dependent elimination to a simple general induction where no rewriting is needed. Consider the following function, introduced in 2.1.

```
Equations lift_fin {n} (k : nat) (f : fin n) : fin (S n) :=
  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).
```

Following the splitting tree, we will need to first eliminate k – whose type is not dependent, nothing special to do here, and then in the context $(n : \mathsf{nat})\ (k : \mathsf{nat})\ (f : \mathsf{fin}\ n)$ we have to eliminate f. We start by generalizing, introducing fresh variables and an equality.

$$(n : \mathsf{nat})\ (k : \mathsf{nat})\ (f : \mathsf{fin}\ n)\ (m : \mathsf{nat})\ (g : \mathsf{fin}\ m)\ (e : (n, f) = (m, g))$$

At this point the straightforward way would simply apply the eliminator for g. Instead, we will apply our optimizations.

- n is a variable, which does not appear anywhere else in the type of f, and whose type is non-dependent. Therefore we immediately use a ***Solution*** step to remove this equality.

$$(k : \mathsf{nat})\ (m : \mathsf{nat})\ (f : \mathsf{fin}\ m)\ (g : \mathsf{fin}\ m)\ (e : f = g)$$

- f is a variable, which does not appear in the type of f, and whose type only depends on a variable that we already unified with a fresh variable. Again, we use a ***Solution*** step to remove it.

$$(k : \mathsf{nat})\ (m : \mathsf{nat})\ (g : \mathsf{fin}\ m)$$

Now we apply the eliminator for g and end up with simply applying the normal eliminator for the original variable, without needing to simplify any further equality.

Of course if we simply check that the original variable to eliminate is fully generalized, as it is the case in this example, we could immediately eliminate it without introducing any equality, but these optimizations also help to reduce the number of equalities in intermediary cases where we can remove some indices but not all.

5.3 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:

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

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

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

It is possible that z was already in Δ, for instance if it depended on a previously solved equality. In this case we just don't need to move it.

Then we move the equality $(e : z = u_j)$ right after the declaration of z. We can do so because the dependency criterion ensures that this equality is homogeneous, that is it does not depend on any equality in e_0.

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

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

$$\Gamma'\ (\overline{u} : \overline{\tau})\ (y : \mathsf{I}\ \overline{u})\ \Delta_z[z := u_j]\ \Delta\ e_0\ e_1[z := u_j, e := \mathsf{eq\_refl}]$$

We end up again with a telescope which has the shape that we want to maintain, and can keep on solving the other equalities that we selected. 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.

5.4 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:

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

where $\mathrm{rew}\ e_0\ z$ performs rewriting in the type of z 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 z is a variable and nothing depends on z and e in e_1 or the (elided) goal.

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

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

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

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

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

This effectively clears z 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.

5.5 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 Δ, which were empty originally. These are variables whose type is "rewritten" definitionally, through the use of the `return` clause of the `match`. This is not an actual problem as long as the guard condition of Coq is able to track well enough these variables. On this subject, this work helped uncover a case where the guard condition was not liberal enough in tracking the subterm relationship. For more details, see the relevant pull-request[5]. No such problems arise when using well-founded recursion instead.

6 RELATED AND FUTURE WORK

In [15] and [30], a specific $\mathsf{f}_{\mathsf{comp}}$ 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.

[5] Not anonymized: https://github.com/coq/coq/pull/920

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

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

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

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

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

In future work, we plan to implement a translation to lift CIC terms into splitting trees, so that the lemma generation phase of Equations can be reused to generate lemmas for existing Coq definitions, and to improve support for nested recursion by recognizing recursive calls through constants like List.map. We also hope to extend the recursion support of Equations to co-patterns and the reduction of productivity to well-founded recursion pioneered by Abel & Pientka [2]. Finally, given the proximity of Equations and Haskell definitions, Equations could provide a better front-end to the `hs-to-coq` tool [31] for the verification of Haskell programs in Coq.

CONCLUSION

We presented a full-featured definitional extension of Coq that makes the definition and reasoning on programs using dependent pattern-matching and complex recursion schemes efficient and effective, without sacrificing assurance. The source language and proof generation facilities of Equations support both `with` and `where` clauses, encompassing mutual, nested and well-founded recursive definitions, which provides a comfortable environment for reasoning on recursive functions. Our central technical contribution is an independent, optimized dependent pattern-matching compiler, based on simplification of equalities of telescopes. It can be reused to implement a robust dependent elimination tactic.

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

[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.

[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

[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

[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 LFMTP'15.

[21] Per Martin-Löf. 1984. *Intuitionistic type theory.* Studies in Proof Theory, Vol. 1. Bibliopolis. iv+91 pages.

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

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

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

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

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

[27] Christine Paulin-Mohring. 1993. Inductive Definitions in the System Coq - Rules and Properties. In *Typed Lambda 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

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

[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

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

[31] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 14–27. https://doi.org/10.1145/3167092

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

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