Equations Reloaded

High-level dependently-typed functional programming and proving in Coq

MATTHIEU SOZEAU, Inria Paris & IRIF - Université Paris 7 Diderot, France
CYPRIEN MANGIN, 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 high-level proof principles for demonstrating properties about them, abstracting away from the implementation details of the function and its compiled form. 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 based on an original handling of the no-confusion property of constructors.

CCS Concepts: • **Software and its engineering → General programming languages**;

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

ACM Reference Format:
Matthieu Sozeau and Cyprien Mangin. 2018. Equations Reloaded: High-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 31 pages.

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 structural or well-founded recursion and compiles them down to the core type theory of Coq. In addition, it automatically derives useful reasoning principles in the form of propositional equations describing the functions, and elimination principles that ease reasoning on them, abstracting away from the compiled form. 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 [Norell 2007], where the justification of dependent-pattern matching definitions in terms of core rules is proven separately as in [Cockx 2017] 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. 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 :=
```

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

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

```
  nth (cons x _) fz := x;
  nth (cons _ v) (fs 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, in the second clause variables v and f have matching types.

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 (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[1]:

```
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 = a.
Check nth_equation_3 : ∀ A n a v f, nth (cons a v) (fs f) = nth v f.
```

The first generated "equation" is actually a proof that nth nil 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 a) →
      (∀ A n a v f, P A n v f (nth v f) → P A (S n) (cons a v) (fs 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.

Dependent pattern-matching allowed us to treat partiality by ascribing a precise type to nth avoiding the pathological case where the index is out-of-bounds. Dually, well-founded recursion allows us to use logic to justify totality in situations where types alone are not enough. Consider the following definition nubBy taken from Haskell's standard library, which removes duplicates from a list according to a comparison function, using a standard definition for filtering a list by a boolean predicate:

```
Equations? nubBy {A} (eq : A → A → bool) (l : list A) : list A by wf (length l) lt :=
  nubBy eq [] ⇒ [];
  nubBy eq (x :: xs) ⇒ x :: nubBy eq (filter (fun y ⇒ negb (eq x y)) xs).
  Proof. simpl. auto using filter_length with arith. Defined.
```

This function definition is not obviously structurally recursive, as it depends on the definition of filter, so Coq's guardedness checker and Agda's size-change termination checker would reject it. Indeed, for all we know, filter could add elements to xs and our definition would diverge. Using the

[1]We declared the type argument A of nil and cons implicit, as well as the n argument of cons, fz and fs for conciseness, and generally elide unnecessary type annotations.

by wf annotation, we can express that this function terminates because the size of the input list strictly decreases (the lt relation is less-than on natural numbers). We can prove independently that filter preserves or decreases the size of its input list to justify that indeed this function is terminating. The Equations? command enters the proof mode to solve obligations associated to the definition, here a single one for the recursive call. While it might not be apparent in the source code, the recursive prototype of the function is considered as part of the dependent pattern-matching problem we are solving, so in the second clause, nubBy's type is refined into:

$$\text{nubBy} : \forall (l' : \text{list } A), \text{length } l' < \text{length } (x :: xs) \to \text{list } A$$

This explains why the obligation to prove has conclusion:

$$\text{length } ((\text{filter } (\text{fun } y \Rightarrow \text{negb } (eq\ x\ y))\ xs) < \text{length } (x :: xs)$$

Programs defined by well-founded recursion use a specific fixpoint combinator that recurses on the well-foundedness proof and inspects the obligations proofs. Nonetheless, we can still derive equations and an elimination principle that entirely abstract away from this implementation detail: the equations for nubBy are precisely its two clauses in this case.

To the best of our knowledge, the only other option to prove this program is terminating would be to extend the type theory with a system of sized types [Abel 2006; Hughes et al. 1996], annotating lists with their sizes and the type of filter to reflect its size preservation property. We argue that well-founded recursion is more modular in the sense that it is kept separate from datatype and function definitions and multiple different measures or relations can be defined on the same type. Additionally, it does not require building an extension of the core type theory.

Finally, well-founded recursion works well with abstraction. The following excerpt is from Vazou et al. [2017]'s comparison of Liquid Haskell and Coq. The chunk function below chunks a value in an abstract type of chunkable monoidal types into a list of values of a given positive size i[2].

```
Context {T : Type} `{M : ChunkableMonoid T}.
Equations? chunk (i: { i : nat | i > 0 }) (x : T) : list T by wf (length x) lt :=
  chunk i x with dec (length x <=? i) :=
    { | left _ ⇒ [x] ;
      | right p ⇒ take i x :: chunk i (drop i x) }.
  Proof. apply leb_complete_conv in p. rewrite drop_spec. omega. auto with arith. Qed.
```

The termination of this function depends on the chunkable monoid interface of the abstract type T. The interface, defined as a type class, provides a measure length on T along with take and drop functions allowing to split a value in two parts, with proofs of their relations to the measure. For example drop_spec specifies that the length of drop i x is equal to length x - i. We use a with construct to test if length x <=? i, and wrap the test in dec which is turning a boolean into a proof that it is equal to true or false. In the case right p, p has hence type (length x <=? i) = false, which is essential to conclude that the recursive call is allowed.

Sized-types use a second class quantification on sizes, so the user cannot produce termination arguments like this one which rely on logical reasoning.

Issues of trust

While the difference of viewpoint between core calculus extensions and elaborations 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

[2] Context introduces the given variables in the local context and makes the following definitions parametric over them

so-called de Bruijn principle: proofs should be checkable using a relatively small proof checker. There is not only the possibility of bugs which 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 [The Univalent Foundations Program 2013]. These two axioms are contradictory (§1.2). The following sections explain our design choices to achieve axiomatic freedom, while providing the benefits 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.

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:

```
Set Warnings "-notation-overridden".
Inductive eq {A : Type} (x : A) : A → 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, called the J rule in type theory jargon:

```
eq_rect_dep : ∀ {A} x (P : ∀ y : A, x = y → 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.

An application eq_rect_dep $x\ P\ p\ y\ e$ computes 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. In other words, 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, and assumptions of equality type might be false (e.g. $0 = 1$). 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 [1992], axiomatically defining a notation for dependent pattern-matching programs, and later refined by McBride [1999], 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.

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

In [Goguen et al. 2006], dependent pattern-matching was explained in terms of simplification of heterogeneous equalities which were defined using the UIP principle (although, in his PhD, McBride [1999] 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) [Hofmann and Streicher 1994], while it is trivially derivable in Extensional Type Theory [Martin-Löf 1984, p32] and Observational Type Theory [Altenkirch et al. 2007]. It can be shown equivalent to Streicher's K [Streicher 1993] axiom which stipulates that proofs of *reflexive* equality can be eliminated to eq_refl. UIP and K are hence used interchangeably in the literature.

$$\mathsf{K} : \forall\,(A : \mathsf{Type})\,(x : A)\,(P : x = x \to \mathsf{Type}), P\ \mathsf{eq\_refl} \to \forall\,(e : x = x), P\ e \tag{2}$$

Enter Homotopy Type Theory (HoTT) and Univalence [Pelayo and Warren 2012], whose central principle directly contradicts the uniqueness of identity proofs principle. Univalence proclaims that equality of types is equal to equivalence of types (a higher-dimensional variant of isomorphism):

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

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 ω-groupoids [Lumsdaine 2010; van den Berg and Garner 2011]. 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 (hSet), 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 [Kraus et al. 2013] that usual data structures with decidable equality like natural numbers enjoy UIP, provably.

To remedy this apparent conflict between UIP and Univalence, and give a meaning to dependent pattern-matching compatible with both, 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 (dependent tuples). This was done for an "axiomatic" version implemented in Agda [Cockx et al. 2014] and for a "definitional" translation in Coq [Mangin and Sozeau 2015], which clearly delimited 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 and Devriese [2017, 2018] 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. This flag enforces 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 this solution involves constructing during unification a substitution that should come from a chain of computationally-relevant type equivalences which are not actually built by the unifier. They are proven to exist and enjoy a strong definitional property in the metatheoretical proofs only. While we were able to reproduce this result[3], 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 [Avigad et al. 2017] or using strict propositions extended with UIP [Gilbert et al.

[3] `material/theories/telescopes.v`.

2019]), 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 [Anand et al. 2017] that erases 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.

We provide a UIP typeclass to let the user either provide a provable instance of UIP on a given type or declare the axiom for all types, and parameterize pattern-matching compilation over it. Note that provable UIP on a given type is a central tool to provide well behaved structures in vanilla type theory already, and is extensively used in Mathematical Components [Mahboubi et al. 2018], for example to build finite sets whose extensional equality coincide with Leibniz equality. In the setting of HoTT, it can still be useful to use the UIP class to perform powerful dependent eliminations in proofs about hSets, without introducing any axiom.

1.4 A homogeneous No Confusion Principle

In Cockx and Devriese [2017], dependent elimination was solved by relying on a notion of higher-dimensional unification. We depart from this solution by relying instead on an homogeneous no-confusion principle, giving a simpler explanation of dependent pattern-matching compilation. No-confusion is the idea that constructors of inductive families are both discriminable and injective. Our solution is more limited in the sense that we do not handle higher-dimensional equations that can arise for example when pattern-matching on the equality type, e.g. for HoTT-style reasoning, but it does apply to indexed inductive families for which pattern-matching does not require UIP. We defer a detailed presentation of no-confusion and our solution to section 4.3, and first motivate why an elaborate solution is needed.

We take an example inspired by the work on Exceptional Type Theory [Pédrot and Tabareau 2018]. This extension of CIC introduces a notion of effect in the type theory, namely call-by-name exceptions. When working in the impure fragment of that theory, all inductive types have an additional raise_ℕ argument for a parametric type of exception names E. For example, natural numbers become:

```
Inductive ℕ (E : Type) : Type :=
| O : ℕ E | S : ℕ E → ℕ E
| raise_ℕ : E → ℕ E.
```

If the exception type is empty, then this type is equivalent to the natural numbers. In the raise_ℕ case we use the *empty pattern* ! to indicate a variable with empty type, here False.

```
Equations N_empty : ℕ False → nat :=
N_empty O := Datatypes.O;
N_empty (S n) := Datatypes.S (N_empty n);
N_empty (raise_ℕ !).
```

Exceptional type theory handles the whole of CIC, hence effectful values can also appear in indices of inductive families, which themselves contain exceptions. E.g., vectors become Vec:

```
Inductive Vec E (A : Type) : ℕ E → Type :=
| nil : Vec E A O | cons : ∀ {n} (x : A) (xs : Vec E A n), Vec E A (S n)
| raise_vec : ∀ (e : E), Vec E A (raise_ℕ e).
```

In op. cit., the authors develop an exceptional parametricity translation that can be used to show that translating any pure term in the effectful calculus produces an effect-free term. To do so, the parametricity translation produces inductive families expressing validity of a given potentialy effectful term. Validity is exception-freeness in this case. Below is the definition of the parametricity relation for effectful vectors: it explictely carves out exception free vectors and indices.

```
Inductive Vec_param {E A} : ∀ (n : ℕ E), Vec E A n → Type :=
| vnil_param : Vec_param O nil
| vcons_param : ∀ (n : ℕ E) (a : A) (v : Vec E A n), Vec_param n v → Vec_param (S n) (cons a v).
```

The exceptional parametricity translation produces validity proofs for any pure term of CIC, but as soon as one wants to reason on locally effectful terms to embed them in globally pure ones, one must reason with the validity proofs. This is where dependent pattern-matching can get in the way. Typically, one would like to show that if a non-empty vector is parametric then its tail will be as well:

```
Equations param_tl {E A} a n (v : Vec E A n) (X : Vec_param (S n) (cons a v)) : Vec_param n v :=
  param_tl a n v (vcons_param a n v X) := X.
```

Under the hood, dependent pattern-matching compilation needs to solve a constraint of the form cons a n v = cons a' n v' :> Vec E A (S n) into the simpler constraint $(a, v) = (a', v')$ using injectivity of the cons constructor. After that, the arguments a, a' and v, v' can be unified to prove that the vcons_param a n v X is the only way to introduce a proof of Vec_param (S n) (cons a v). It is this first seemingly trival simplification step between indexed vectors that, if done naïvely, would introduce UIP. Note that none of the indices here have decidable equality as E is abstract, so we cannot rely on a potential proof of UIP for Vec.

1.5 Pattern-Matching and Recursion

Dependently-typed functional programming involves not only pattern-matching on indexed families but also recursion on the inductive structure of terms. There are basically three ways to present recursion on inductive families in dependent type theories:

- 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. Lean uses this solution. This eliminator construction is usually justified from the construction of initial algebras in a categorical model. The eliminator for the accessibility inductive family provides a way to encode well-founded recursion.
- The second is to extend the type theory with size annotations [Abel 2006; Abel et al. 2017], providing a way to bake information about sizes of objects in types and reduce termination and productivity checking to type checking. This extension requires an elaborate extension of the type theory, its implementation and its models, along with pervasive size annotations in types and terms.
- The third way is to use a separate criterion to check termination of definitions, represented either as clauses (in Agda and Idris) or using a decomposition of eliminators into pattern-matching (e.g. ML's `match`) and recursion (ML's `let rec`) as done in Coq. Agda and Idris use an external size-change termination checker, while Coq uses a guardedness check.

The upside of the last two methods is that they provide 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 or nested structural recursion.

The downside is that it either requires to trust an external checker or extend the core type theory. In the case of Coq, there is a complex syntactic guard-checking criterion that must be used to verify that definitions are normalizing, as part of the type-checking algorithm implemented 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 a drastic way, disallowing perfectly fine definitions in Coq.

Again, to avoid these subtle trust issues, our solution is simple: elaborate complex recursive definitions using the tools of the logic itself instead of extending the core calculus. We will do so using the well-known, constructive accessibility characterization of well-founded recursion. Lean essentially uses the same methodology for defining functions. 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.

Beside the flexibility in termination orders we can use to define recursive definitions in the system, switching to well-founded recursion also permits not to worry about guard checking our (compiled) programs anymore. 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 terminating, using logical reasoning on the derived transitive closure of the strict subterm relation. Unless specified otherwise, all the structurally recursive definitions in this paper can equivalently be defined as well-founded on the subterm relation.

The set of relations that can be shown well-founded in a type theory is essentially a measure of its logical strength: Hancock [2000] provides a thorough exploration of this idea. The upshot is that well-founded recursion is the ultimate tool to write terminating programs in type theory. Following this idea and using a constructive version of Ramsey's theorem, Vytiniotis et al. [2012] have shown that using so-called "almost-full" relations (related to well-quasi-orders) allows to prove compositionally the well-foundedness of a large class of relations. They proposed that heuristic criterions like size-change termination should be internalized in type theory using almost-full relations. We have been able to finish their program and formalized a reflexive version of the size change termination principle that can indeed be used to construct well-founded relations in Coq, justifying size-change terminating programs, we hope to integrate it with Equations in the future.

1.6 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.

Nonetheless, one 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: Cockx [2017]'s proof apply directly in our case.

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 user decides to use UIP as an axiom, the propositional equalities can still be derived (UIP implies its own reduction rule) but we provide no guarantee about the computational behavior of the function inside Coq. We only conjecture that its *extraction*, which removes all proof decorations, will have the right computational behavior.

In the case of reduction of well-founded fixpoints, the situation is similar. If one uses our derived subterm relation to show termination, then the resulting function will compute in exactly the same way as if it was structurally recursive on the recursive argument: the accessibility proof of the subterm relation on a term directly mimicks the structure of that term. This is the same situation as for the Below predicate used to justify structurally recursive definitions since Goguen et al. [2006] (Lean uses Below by default). If one provides a closed proof of some other well-founded relation, then the definition will also compute as expected on closed terms, but we cannot provide guarantees on its definitional behavior on arbitrary arguments: the equations generated by a well-founded definition *do not* form a terminating rewrite system in general. This actually shows the power of well-founded recursion: it goes beyond structurally-recursive definitions by incorporating arbitrary logical reasoning in the termination argument.

1.7 Contributions

In its first version [Sozeau 2010], the Equations tool relied on heterogeneous equality (a.k.a. "John-Major" equality) to implement the so-called "specialization by unification" [Goguen et al. 2006] necessary to witness dependent pattern-matching compilation. It was only a prototype, using large amounts of fragile $\mathcal{L}_{\text{tac}}$ definitions and tricks to implement simplification.

In this paper, we present a complete rewrite of Equations based on a new implementation of simplification which removes these limitations. Our main contributions are:

- An extended source language for Equations including global and local `where` clauses for defining mutual or nested structurally recursive functions and nested well-founded programs respectively (§ 2.2). Through dependent pattern-matching, mutual well-founded programs can also be easily represented. The language supports `with` and pattern-matching lambdas, and integrates well with Coq's implicit arguments and notations and its proof mode.
- A new dependent pattern-matching simplification algorithm, implemented in ML, and compatible with both the UIP principle and univalence. This algorithm relies on an original homogeneous variant of the no-confusion property to treat injectivity of constructors in indexed families and is the main technical contribution of this paper.
- This algorithm produces axiom-free proof terms to be checked by the Coq kernel, and can be used independently to build a new dependent elimination tactic.
- Based on the intermediate hierarchical representation of programs, we can derive their 1-unfolding and associated equations and elimination principles. The eliminators derived for recursive programs are the most general ones and allow working more comfortably with mutual, nested and well-founded recursive definitions than in vanilla Coq.

This new system is released, stable and freely available[4]. It has been tested on a variety of examples, including a proof of strong normalization for predicative System F [Mangin and Sozeau 2015], a reflexive tactic for deciding equality of polynomials, parts of Chlipala [2011]'s book on

[4] material/equations.tgz. Sources of the article examples can be found in material/article.

certified programming with dependent types and an interpreter for an intrinsically-typed imperative language from Poulsen et al. [2018], all available on the website.

Structure of the paper. The Equations package is structured modularly and the rest of the article follows this structure:

(1) It first parses Equations definitions using an extension of Coq's parser (§2) into an abstract syntax tree of mutual and nested recursive programs, which we present in detail through examples.
(2) It then interprets these programs as splitting trees, performing coverage checking of pattern-matching using unification and elaborating recursive programs (§3). This pass may fail if the pattern-matching problem falls out of the theory of constructors and equality or if type-checking of right hand sides fails.
(3) Splitting trees can then be elaborated to Coq terms by witnessing dependent-pattern matchings with explicit equality manipulations and recursive definitions with the primitive `fix` construct of Coq or a generic well-founded fixpoint combinator (§4). This pass relies on automatically derivable definitions for inductive families (NoConfusion and Subterm) and may fail if they are not available. It also depends on the configurable assumption of UIP or the availability of a homogeneous no-confusion principle (§4.3).
(4) Using a substitution operation on splitting trees, we can derive the 1-unfolding of recursive definitions. From this, we can straightforwardly derive propositional equations corresponding to the defining equations of the programs, their graph and elimination principle (§5).

We compare our solution to other systems and review related work in §6.

2 THE SOURCE LANGUAGE OF EQUATIONS

2.1 Source syntax

The compilation process starts from a list of programs consisting of a signature and a list of clauses given by the user, constructed from the grammar given in figure 1. In the grammar, $\overrightarrow{t}$ denotes a possibly empty list of t, $\overrightarrow{t}^{+}$ a non-empty list. Concrete syntax is in `typewriter` font.

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

Fig. 1. Definitions and user clauses

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

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

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

2.2 From Structural to Nested Well-founded Recursion

Equations allows the user to define nested or mutually 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. In the `wf` annotation, the first term should be typable in the program's context Γ (and possibly the enclosing context for nested programs) with a *closed* type τ (e.g. nat), while the relation must be a locally closed term of type $\tau \rightarrow \tau \rightarrow$ Prop. If the relation is not given, Equations launches a type-class search for an instance of WellFounded on the carrier type (derived Subterm relations are such instances). The optional `struct` annotation indicates optionally which argument is structurally recursive, or let it be inferred.

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 on one of the arguments, as in the the nth example in the introduction. Using the top-level `with` construct, one can straightforwardly define mutually recursive definitions as with the `Fixpoint with` construction of Coq. We next describe the novel treatment of local, nested and well-founded recursion that Equations provides.

2.3 Local recursion

A common idiom of functional programming is the worker/wrapper pattern. It usually involves a recursive function that computes the result, wrapped in a toplevel function calling it with specific parameters. The paradigmatic example is probably list reversal, whose tail-recursive version can be written using a recursive local where clause:

```
Equations rev_acc {A} (l : list A) : list A :=
  rev_acc l := go l []
   where go : list A → list A → list A :=
         go [] acc := acc;
         go (hd :: tl) acc := go tl (hd :: acc).
```

A typical issue with such accumulating functions is that one has to write lemmas in two versions to prove properties about them, once about the internal go function and then on its wrapper. Using the functional elimination principle associated to rev_acc, we can show both properties simultaneously.

```
Lemma rev_acc_eq : ∀ {A} (l : list A), rev_acc l = rev l.
Proof.
```

We apply functional elimination on the rev_acc *l* call. The eliminator expects two predicates: one specifying the wrapper and another for the worker. For the wrapper, we give the expected final goal but for the worker we have to invent a kind of loop invariant: here that the result of the whole go *acc l* call is equal to rev *l* ++ *acc*.

```
apply (rev_acc_elim (fun A l revaccl ⇒ revaccl = rev l)
                    (fun A _ l acc go_res ⇒ go_res = rev l ++ acc)).
```

Functional elimination provides us with the worker property for the initial go [] *l* call, i.e. that it is equal to rev *l* ++ [], which trivially gives us the result. For the worker proof itself, the result follows from associativity of *app* and the induction hypothesis. Qed.

The local function could equivalently be defined as well-founded on the size of the l argument, the same equations and eliminator would be derived.

2.4 Nested structural recursion

Mutual recursion can be seen as a special case 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.

2.4.1 Structural recursion on nested types. 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 an application constructor taking lists of terms as arguments (see, e.g. [Poulsen et al. 2018] for an application of well-scoped terms).

```
Inductive term : nat → Set :=
| Var {n} (f : fin n) : term n
| Lam {n} (t : term (S n)) : term n
| App {n} (t : term n) (l : list (term n)) : term 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 assume lift : ∀ {n} (k : nat) (σ : term n) : term (S n) and concentrate on substitution which is defined similarly. Using lift we can define a substitution extension function extend_var which lifts a substitution of k variables into a substitution of k + 1 variables keeping the first variable untouched.

```
Equations extend_var {k l : nat} (σ : fin k → term l) (f : fin (S k)) : term (S l) :=
```

```
  extend_var σ fz ⇒ Var fz ;
  extend_var σ (fs f) ⇒ lift 0 (σ f).
```

For definitions of fixpoints on nested mutual inductive types, Equations allows users to factorize the nested fixpoint definitions in toplevel `where` clauses, so that one does not need to write an internal fixpoint construction inside the program. Multiple calls to the nested function can also refer to the same function, e.g if we extend our term structure with other constructors using lists of terms. We want to use the notation t [σ] for subtitution σ applied to t. We first have to declare it to the parser, and then bind it to its expansion using a where clause.

```
Reserved Notation "t [ σ ]" (at level 10).
Equations tsubst {k l : nat} (σ : fin k → term l) (t : term k) : term l := {
  (Var v) [σ] ⇒ σ v;
  (Lam t) [σ] ⇒ Lam (t [extend_var σ]);
  (App t l) [σ] ⇒ App (t [σ]) (tsubsts σ l) }
where tsubsts {k l} (σ : fin k → term l) (t : list (term k)) : list (term l) := {
  tsubsts σ nil ⇒ nil;
  tsubsts σ (cons t ts) ⇒ cons (t [σ]) (tsubsts σ ts) }
where "t [ σ ]" := (tsubst σ t) : term.
```

The Coq kernel will check a single fixpoint definition for tsubst where tsubsts has been expanded at its call sites, as definitions on nested recursive types correspond to nested local fixpoints in CIC.

2.4.2 *Reasoning.*

Remark that our definition of tsubsts 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 tsubst rewrite database gathering the definining equations of tsubst to abstract away from this detail:

```
Lemma tsubsts_map k l σ t : @tsubsts k l σ t = List.map (tsubst σ) t.
Hint Rewrite tsubsts_map : tsubst.
```

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. Given that the tsubsts function is essentially mapping the substitution, we can derive a specialized induction principle giving us Forall2 $P\ l$ (map (tsubst σ) l) hypotheses for the recursive call to tsubsts. Forall2 $P\ l\ l'$ is equivalent to the pointwise conjunction of P for the elements of l and l'. We can derive:

```
Lemma tsubst_elim_all (P : ∀ k l : nat, (fin k → term l) → term k → term l → Prop) :
  (∀ k l σ (f : fin k), P k l σ (Var f) (σ f)) →
  (∀ k l σ (t : term (S k)), P (S k) (S l) (extend_var σ) t (t [extend_var σ]) →
    P k l σ (Lam t) (Lam (t [extend_var σ]))) →
  (∀ k l σ (t : term k) (ts : list (term k)), P k l σ t (t [σ]) →
    Forall2 (P k l σ) ts (map (tsubst σ) ts) →
    P k l σ (App t ts) (App (t [σ]) (map (tsubst σ) ts))) →
  ∀ k l σ t, P k l σ t (t [σ]).
```

This is good, however the program still relies on the syntactic guardedness check. It only takes a bit of type information to get this useful reasoning principle directly, using well-founded recursion.

2.5 Well-founded recursion on nested types

Well-founded recursion requires us to give an explicit relation explaining why going through the list in the application case is ok. We will do so by relying on the fact that map $f\ l$ can only apply f to members of l. Note that this is not a consequence of the parametricity of map, but can easly be seen from its definition. To reflect this in the logic, we must define a variant of map that carries proofs of membership in l of each element passed to f. Membership is a standard notion of the theory of lists, which can be defined inductively as follows[5]:

```
Inductive In {A} (x : A) : list A → Prop :=
| here {xs} : x ∈ (x :: xs)
| there {y xs} : x ∈ xs → x ∈ (y :: xs) where "x ∈ s" := (In x s).
```

```
Equations mapIn {A B : Type} (l : list A) (f : ∀ a, a ∈ l → B) : list B :=
  mapIn nil _ := nil;
  mapIn (cons x xs) f := cons (f x here) (mapIn xs (fun x H ⇒ f x (there H))).
```

mapIn is a dependently-typed variant of map which passes proofs of membership to f. Note that f's type is refined to $\forall\ a, a \in (x :: xs) \rightarrow B$ in the second clause. In case the function does not use its argument, it behaves like a regular map.

```
Lemma mapIn_irrel {A B} (f : A → B) l : mapIn l (fun (x : A) (_ : In x l) ⇒ f x) = List.map f l.
```

More interestingly, mapIn transforms a predicate valid on all members of a list into a property of the list and its mapping, which is easily proven by functional elimination.

```
Lemma mapIn_spec {A B} (l : list A) (g : ∀ x : A, In x l → B) (P : A → B → Prop) :
  (∀ a (ina : In a l), P a (g a ina)) ↔ Forall2 P l (mapIn l g).
```

We can also define a well-founded relation on term that shows that any member of the list in the App constructor is a subterm. The Subterm derivatiom algorithm of Equations does not yet recognize these nested types: it would produce a relation without term_sub_3 here, so we define it ourselves. Note that this is an *heterogeneous* relation between terms with potentially different numbers of free variables. The Var constructor has no recursive subterm, while Lam has a direct subterm with one more free variable and App has one constructor for the substerm and one for the argument list. We add these constructors to a hint database that is used to prove recursive call obligations.

```
Inductive term_sub : ∀ {m n}, term m → term n → Prop :=
| term_sub_1 : ∀ n (t : term (S n)), term_sub t (Lam t)
| term_sub_2 : ∀ n (t : term n) l, term_sub t (App t l)
| term_sub_3 : ∀ n (t : term n) l x, In x l → term_sub x (App t l).
```

We define the actual relation that is well-founded as the transitive closure of term_sub. Note that the relation must be *homogeneous* and on a closed type, so we pack the terms with their number of free variables in a dependent pair. It is shown well-founded by a *nested* structural recursion on the term and list term structure and dependent elimination of the term_sub inductive family: in the end we must always come back to Coq's primitive fixpoint constructions. However this must be done only once per datatype: this relation can be used to justify nested recursive definitions without ever coming back to the syntactic check, while still enjoying the same definitional equations on closed terms. For non-nested cases using our derived Subterm relation, we even get the *same* definitional equations, as for the standard Below encoding.

```
Definition term_subterm := clos_trans (λ x y : (Σ n, term n), term_sub x.2 y.2).
Instance wf_term_subterm : WellFounded term_subterm.
```

We come back to our definition of substitution. This time the function is defined as well-founded on the subterm relation for terms, and we do not need to inline the definition of mapIn. The proofs

[5]We skip implicit arguments, notations, hints and type-class instance declarations in the following, unless they are crucial.

of termination are solved automatically using the previously declared hints. They are trivial: for the nested recursive call, we must show term_sub t (App t l) under the assumption $Inl : t \in l$.

```
Equations tsubst2 {k l} (σ : fin k → term l) (t : term k) : term l by wf (k, t) term_subterm := {
  (Var v) [σ] ⇒ σ v;
  (Lam t) [σ] ⇒ Lam (t [extend_var σ]);
  (App t l) [σ] ⇒ App (t [σ]) (mapIn l (fun t Inl ⇒ t [σ])) }
where "t [ σ ]" := (tsubst2 σ t) : term.
```

An eliminator equivalent to tsubst_elim_all is automatically derived. Note that mapIn is entirely generic for lists. The principle that mapping a function over a container should give us a proof that the elements passed to the client do belong to the container is essential here. In particular this idea would also work with an abstract container type (e.g. sets) if it provided a similarly strongly specified map function. This is proof dependent programming, but of course the extraction of mapIn does *not* carry membership proofs, as In is a *proposition* here (a Prop).

In a system with sized types, a direct call to regular map would be allowed here, but requires the term datatype and substitution function to be indexed by sizes[6]. With size annotations the l argument of App has type list (term j k) while the initial term t has type term i k, with size relation $j < i$. The nested call becomes map (tsubst j σ) l: i.e. we pass a smaller size to the substitution. In contrast, we do the proof passing explicitly through mapIn. We can also provide a size-based variant of map and recover a similar elimination principle for a substitution based on it (see §7).

Mutual well-founded recursion. Finally, combining dependent-pattern matching and well-founded recursion on indexed families allows to express mutual recursion. We can provide a GADT-like encoding of signatures sign A P of mutual functions of signature $\forall (x : A), P\ x$ for varying A and P, define measures or well-founded orders on them and from that, reduce mutual functions to single well-founded functions. This is a folklore encoding trick reminiscent of how mutually inductive families are reduced to inductive families in Type Theory[Paulin-Mohring 1996]. The interested reader can find a worked-out example for tsubst in the appendix (§7) as well.

This concludes our presentation of the core features of Equations's source language and its derived notions.

3 ELABORATING EQUATIONS TO SPLITTING TREES

We will now present in more formal terms the process of elaboration from an Equations definition to its intermediate splitting tree representation.

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 a 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 $\overrightarrow{\mathsf{I}_i : \forall\ \Gamma_i, \mathsf{I}\ \overrightarrow{t_i}}$ (where $\Gamma_i \vdash \overrightarrow{t_i} : \Delta$). Although CIC distinguishes between parameters and indices and our implementation does too, we will not distinguish them in the presentation for the sake of

[6] material/agda/nested_sized.agda formalizes this

simplicity. The dependent sum / sigma type is written $\Sigma x : \tau.\tau'$, its introduction form is $(\_, \_)$ and its projections are in post-fix notation $\_.1 : \Sigma x : \tau.\tau' \to \tau$ and $\_.2 : \forall s : (\Sigma x : \tau.\tau').\tau'[s.1]$.

3.2 Searching for a covering

The first phase of the compiler produces 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 $?(t)$), equivalent to Agda's "dot" patterns to recover a sense of which case splittings are performed: inaccessibles are never matched on but are determined by other patterns. The compilation is as a search procedure, as usual, we recover a deterministic semantics using a first-match rule when two clauses overlap.

`program`	$prog, where$	::=	$(\ell_p, \Gamma, \tau, rec?, spl)$
`recursion`	rec	::=	$\mathtt{wf}(t, R) \mid \mathtt{struct}\ x \mid \mathtt{nested}\ x$
`pattern substitution`	σ	::=	$\Delta \vdash \overrightarrow{p} : \Gamma$
`pattern`	p	::=	$x \mid \mathsf{C}\ \overrightarrow{p} \mid ?(t) \mid \mathtt{hide}(x)$
`splitting`	spl	::=	$\mathsf{Split}(\sigma, x, ((spl)?)^n) \mid \mathsf{Compute}(\sigma, \overrightarrow{where}, t)$

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

The search for a covering works by gradually refining a *pattern substitution* $\Delta \vdash \overrightarrow{p} : \Gamma$ 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. Patterns are built from variables, constructors, arbitrary inaccessible terms `?( t )` or hidden variables `hide(` x `)`. Hidden variables are used to handle implicit bindings of variables from the enclosing context that do not need to be matched by a user pattern but might still get refined through dependent pattern-matching: recursive function prototypes and the enclosing contexts of local `where` nodes will be interpreted as such.

To check covering of a subprogram with bindings Γ and enclosing context Δ, we start the search with the problem $\Delta, \Gamma \vdash \overline{\mathtt{hide}(\Delta)}\ \overline{\Gamma} : \Delta, \Gamma$, i.e. the identity substitution on Δ, Γ, hiding the enclosing variables. Covering also takes the list of user clauses. For example, coverage checking of tsubst from section §2.4 starts with a pattern substitution for a context with *tsubst*, *tsubsts*, k, l, σ and t, where the two recursive prototypes are hidden. The left-hand sides of clauses will already have been parsed to full applications of tsubst (each left-hand side is a well-typed instantiation of the function), from which we will get patterns for k, l, *sigma* and t.

At each point during covering, we can compute the expected target type of the current subprogram by applying the substitution to its initially declared type τ. The search for a covering and building of the splitting tree is entirely standard. This follows the intuitive semantics of dependent pattern-matching (e.g., the same as in Agda, Idris and 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. Cockx and Abel [2018] have given a pen-and-paper proof of elaboration from clauses to case trees for dependent (co)pattern-matching that handles a superset of Equations definitions, we will not attempt to do better here.

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

- A $\mathsf{Split}(\Delta \vdash \overrightarrow{p} : \Gamma, x, (spl?)^n)$ 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 *spl*. When the type of x does not unify with a particular constructor type the corresponding splitting is empty. Otherwise the substitution built by unification determines the pattern substitution used in each of the subcoverings.

- A $\mathsf{Compute}(\Delta \vdash \overrightarrow{p} : \Gamma, \overrightarrow{w}, t)$ node, denoting a right-hand side whose definition is t (of type $\tau[\overrightarrow{p}]$) under some set of auxiliary local definitions $\overrightarrow{w}$. Both `with` and `where` clauses are compiled this way. A `with` clause is essentially interpreted as a `where` clause with a new argument for the abstracted object and correspondingly generalized return type. There are subtleties related to the elaboration of `with`, due to the strengthening and abstraction it performs that were already worked out in detail by Sozeau [2010], we do not focus on this here. 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. Local `where` clauses otherwise directly elaborate to a context of auxiliary local definitions in this representation, enriching the local context Δ in which the body t is type-checked.

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

- A $\mathtt{wf}(t,\ R)$ annotation denotes an application of the well-founded fixpoint combinator to relation R (typed in the empty context) and measure t (typed in Δ, Γ). The subsplitting s corresponding to the subprogram has a new variable for the recursive prototype:

$$\Delta, \Gamma, \ell_p : \forall \overline{\gamma' : \Gamma}, R\ t[\gamma']\ t[\gamma] \to \tau$$

- A `struct` x or `nested` x annotation denotes a structurally recursive or nested recursive fixpoint of Coq, where x is a single variable declared in the context Γ. In that case the recursive prototype added to the context is a closed type. At the toplevel, we allow a mutual set of structurally or nested recursive programs, which are then typed in a context with all the recursive prototypes available.

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 [Goguen et al. 2006] and [Sozeau 2010] with minor modifications. We recall the main construction first, then focus on our solution to the injectivity of constructors and finally present the simplification engine used by Equations to perform specialization by unification.

4.1 Compilation of a splitting tree

4.1.1 Overview. We must now give a witness for each node in the splitting tree. In the case of a $\mathsf{Compute}(\Delta \vdash \overrightarrow{p} : \Gamma, \overrightarrow{w}, t)$ node, we first compile each auxiliary local definition in $\overrightarrow{w}$, producing a context extension of Δ in which t has type $\tau[\overrightarrow{p}]$.

For a $\mathsf{Split}(\Delta \vdash \overrightarrow{p} : \Gamma, x, (s?)^n)$ 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 witnessing the elimination. To do so, we will generate a proof term using an eliminator of the type of x, no-confusion and rewriting lemmas, whose leaves correspond to each non-empty splitting in s.

4.1.2 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.

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

For an inductive $\mathsf{I} : \forall\overline{\Delta}, s$, its packing is defined as $\Sigma\ i : \Sigma(\Delta), \mathsf{I}\ \overline{\Sigma}(\Delta, i)$. We follow Cockx and Devriese [2017] 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 function to inject a value in the signature, which can be used when programming with packed types:

$$\mathsf{pack}_{\mathsf{I}} : \forall\ \Delta\ (x\ :\ \mathsf{I}\ \overline{\Delta}), \bar{\mathsf{I}} := \lambda\ (\overline{\Delta} : \Delta)(x : \mathsf{I}\ \overline{\Delta}), (\sigma(\Delta)(\overline{\Delta}), x)$$

4.1.3 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 by McBride et al. [2004]. 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 $x'\ :\ \bar{\mathsf{I}}$ and an equation between telescopes asserting that x' is equal to the packing of x, giving us a new, equivalent goal:

$$\Gamma, x' : \bar{\mathsf{I}} \vdash x' = \mathsf{pack}_{\mathsf{I}}\ \bar{t}\ x \to \tau \tag{4}$$

After unpacking the variable x' into its index and inductive components, and furthermore unpacking the index into its constituent variables and performing 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 case analysis to it. 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 section 4.4 to simplify these equations. In the following, we consider equality to be in `Prop` but that is irrelevant to our results. Equations is parametric in the sort of the equality, so the paths equality type of HoTT works equally well.

4.1.4 Injectivity and discrimination of constructors. During the simplification part of dependent elimination we 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:

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

As an aside, this is the first time we see an equality between telescopes. Contrary to the variant used by Cockx and Devriese [2017], 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 $\overrightarrow{a_0}$ and $\overrightarrow{a_1}$ (injectivity) that can be further simplified or derive a contradiction if C_0 is different from C_1

(conflict). McBride et al. [2004] describe a generic method to derive such an eliminator that can be adapted to work on telescopic equalities instead of heterogeneous equalities – we will present an extensive example below. 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:

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

4.2 NoConfusion and injectivity of inductive families

On vectors, the heterogeneous no-confusion relation is empty outside the diagonal, and otherwise returns an equality of the arguments of the constructors (or True when there are none). This is an example of the encode-decode method from Homotopy Type Theory: we are characterizing the equality in the vector family in an alternative way.

```
Equations NoConf_vector {A} (x y : Σ n, vector A n) : Prop :=
NoConf_vector (_, nil) (_, nil) := True ;
NoConf_vector (_, @cons a n v) (_, @cons a' n' v') :=
  (a, n, v) = (a', n', v') :> Σ (a : A) (n : nat), vector A n ;
NoConf_vector _ _ := False.
```

One can show that the type noConf_vector $x\ y$ is equivalent to $x = y$: in the first direction, we pattern-match on all arguments, except the indices.

```
Equations noConf_vector {A} (x y : Σ n, vector A n) : NoConf_vector x y → x = y :=
noConf_vector (_, nil) (_, nil) I := eq_refl;
noConf_vector (_, @cons a n v) (_, @cons a' n' v') eq_refl := eq_refl.
```

In the other direction, we split on the equality proof $x = y$, which determines that both arguments are the same, hence the inaccessible patterns. Pattern-matching on the first vector, we discover that its indices are also determined uniquely (the ?(0) and ?(S n) inaccessible patterns witness that):

```
Equations noConf_vector_inv A (x y : Σ n, vector A n) : x = y → NoConf_vector x y :=
noConf_vector_inv A (?(0), nil) ?((0, nil)) eq_refl := I;
noConf_vector_inv A (?(S n), @cons a n v) ?((S n, cons a v)) eq_refl := eq_refl.
```

The equivalence is also *strong* in the sense that even for open terms headed by a constructor, passing a reflexivity proof eq_refl to the inverse noConf_vector_inv and composing with noConf_vector should produce a reflexivity proof. Cockx [2017] showed that this is necessary for the computational behavior of definitions to be correct. Let's recall why this is so. The basic problem we want to solve is injectivity of constructors of indexed families. Concretely, the question is: under which conditions can we solve the following goal depending on an equality between vectors of the same length:

```
Lemma inject_vcons {A} n (a a' : A) (v v' : vector A n) (P : ∀ a' v', cons a v = cons a' v' → Type)
      (prf : P a v eq_refl) : (∀ (e : cons a v = cons a' v'), P a' v' e).
```

We want to simplify the equality by applying injectivity of cons, however me must do so in a proof-relevant way as the goal $P\ a'\ v'\ e$ depends on the shape of the proof e. The solution of op. cit. is to first transform this homogeneous equality into an heterogeneous one, essentially by using the inverse of the J rule. This gives the following goal, where $e..i$ is a notation for projecting the ith component of an equality of telescopes, and e # t is transport along e (a variant of J):

$$\forall\ (e : (\mathsf{S}\ n, \mathsf{cons}\ a\ v) = (\mathsf{S}\ n, \mathsf{cons}\ a'\ v'))\ (e' : e\,..1 = \mathsf{eq\_refl}), P\ a'\ v'\ (e'\ \#\ e\,..2)$$

We now have an heterogeneous equality between two vectors of size S n on which noConf_vector can be applied, as it is an equivalence. This gives an equivalent goal:

$\forall\ (H : \mathsf{NoConf\_vector}\ (\mathsf{S}\ n, \mathsf{cons}\ a\ v)\ (\mathsf{S}\ n, \mathsf{cons}\ a'\ v'))$
$(e' : (\mathsf{noConf\_vector}\ H)\ ..1 = \mathsf{eq\_refl}), P\ a'\ v'\ (e' \#\ (\mathsf{noConf\_vector}\ H)\ ..2)$

Now NoConf_vector can compute as we have constructors at the head: moreover they are both cons, so it gives us a telescopic equality of the arguments. Note that the rest of the goal depends on the shape of H, which should ultimately become a reflexivity proof for this to progress.

$\forall\ (H : (a, n, v) = (a', n, v'))\ (e' : (\mathsf{noConf\_vector}\ H)\ ..1 = \mathsf{eq\_refl} :> \mathsf{S}\ n = \mathsf{S}\ n),$
$P\ a'\ v'\ (e' \#\ (\mathsf{noConf\_vector}\ (\mathsf{S}\ n, \mathsf{cons}\ a\ v)\ (\mathsf{S}\ n, \mathsf{cons}\ a'\ v')\ H)\ ..2)$

Applying simplifications of telescopic equality and the J rule, we can unify a and a' here to make a modicum of progress. But the first equality in the goal will still contain a proof of $n = n$ that cannot be eliminated directly without UIP. The insight of Cockx and Devriese [2017] is to use higher-dimensional unification at this point, concentrating on lower dimensional variants of H and e'. This amounts to doing a nested unification, resulting in a final instantiation of H and e' by eq_refl, which makes noConf_vector reduce to eq_refl as well (as this is a strong equivalence), allowing to close the proof using prf. In other words, it really suffices to show that P holds for the same two vectors and a proof of reflexivity.

4.3 A homogeneous no-confusion principle

Actually, relying on higher-dimensional unification for this problem is overkill. We can rather devise a more precise *homogeneous* no-confusion principle working on any two terms in the *same instance* of the inductive family to directly solve injectivity goals between constructors. Our analysis of the problem is that the heterogeneous no-confusion principle is too general. By comparing constructors in different instances of the same family, it does not rely on the index structure of each constructor, while the problems of injectivity we face are always between objects in the same instance of the familiy, i.e. they share the exact same indices. This means that we can reduce the content of proofs in the no-confusion notion while still maintaining an equivalence with propositional equality. On the cons diagonal case of vectors for example, we can just ask for equalities between the head and the tail of the vector, while removing the problematic equality between the natural number arguments.

Forced arguments. Our homogeneous no-confusion principle essentially relies on the analysis of forced arguments of constructors, a notion pioneered by Brady et al. [2003] and recently used in [Gilbert et al. 2019] to provide a characterization of strict inductive propositions. In essence, an argument x of a constructor c is forced if it appears in c's conclusion in a pattern position. For example, the n argument of cons appears under a successor in cons's conclusion vector A (S n), so it is forced. Initially this information was used to analyse which arguments could be erased at compile-time, but here can make use of this in type theory and internalize this notion through a type equivalence. For vectors, the homogeneous no-confusion principle is defined as:

```
Equations NoConfHom_vector {A n} (x y : vector A n) : Prop :=
NoConfHom_vector nil nil := True ;
NoConfHom_vector (@cons a n v) (@cons a' n v') := (a, v) = (a', v') :> Σ (_ : A), vector A n.
```

This relies on dependent pattern-matching for the index n of vectors: by discriminating on it, Equations can see that there are only two cases to consider: if both vectors are empty or both have the same size n (we don't need to write inaccessible patterns for the n variables, as proven by Cockx and Abel [2018]). In case they are both cons we just need to record the equality of the heads and tails, which actually degenerates into an equality between non-dependent pairs. One can show again that NoConfHom_vector x y is equivalent to $x = y$ by dependent pattern-matching. We just show the forward direction, where we can see that n is inaccessible.

```
Equations noconf_vector {A} n (x y : vector A n) : NoConfHom_vector x y → x = y :=
noconf_vector ?(0) nil nil I := eq_refl;
noconf_vector ?(S n) (@cons a n v) (@cons a' n v') eq_refl := eq_refl.
```

As for NoConfusion, we have a generic `Derive` NoConfusionHom command for generating the homogeneous no-confusion principle of a given datatype, reusing the elaborator of Equations. This is registered as an instance of the NoConfusionPackage type class, which contains the proof of isomorphism with equality.

Note that this derivation will fail if equality of constructors in the inductive family cannot be reduced to equalities of their non-forced arguments. Typically, this is the case of equality: implementing homogeneous no-confusion on equality would be equivalent to UIP as it would be showing that equality of equalities is equivalent to True!

Other examples that fall out of this criterion include any non-linear use of an index (equality is the canonical example) and constructor conclusions that do not fall into the pattern subset of terms, typically function applications. This is not a restriction: unification would get stuck on these indices unless one is performing a general elimination, i.e. eliminating a term in $\mathsf{I}\ \overline{x}$ where all x's are variables, in which case no-confusion is not necessary. The fact that NoConfusionHom is derivable on an inductive family actually ensures that one will never need UIP in pattern-matching problems involving it. One can think of this property as saying they are well-behaved index types. The fin, vector and term types of § 2.4, along with Vec_param and its indices from the introduction enjoy homogeneous no-confusion, as well as most intrinsically-typed syntaxes we are aware of that do not use functions in index positions.

To summarize, we have now two notions of no-confusion: one that applies to heterogeneous goals and another for homogeneous ones. We will favor the homogeneous one during simplification, but the heterogeneous version is still useful if we want to use UIP.

4.4 A simplification engine in OCaml

The initial version of Equations relied on $\mathcal{L}_{\text{tac}}$, the tactic language shipped with Coq, to compile a splitting tree to a term, which was very fragile and slow. Therefore, in the current version, we moved all the the compilation procedure to OCaml. 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.

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 and a context map $\sigma : \Gamma' \vdash \overrightarrow{p} : \Gamma$ explaining the steps performed. Apart from small bureaucratic details, the term c will simply be an application of the appropriate lemma from Equations' Coq library. We can check that the returned context map at the end of simplification is compatible with the one of the splitting to plug the terms in a type-safe way.

4.4.1 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 and what the goal looks likes after it is applied. Note that we could also describe each step as an equivalence of telescopes; instead, we choose here to show how it acts on a given goal, since we are directly manipulating terms and do not need a whole equivalence structure in general. Each of these simplification steps apply under a certain context Γ which stays fixed except for the solution rule. 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 by the first simplification step.

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

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 4.1.4. The function sigma_eq combines the two equalities into one well-typed equality between `(x, p)` and `(y, q)`.

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

This step requires UIP on the type of `t` (it is precisely the K principle), unless P does not actually depend on e. In that case, we can just clear e.

NoCycleLeft $\boxed{\forall\Gamma,\ \forall\ (e : x = \mathtt{t} :> A),\ \mathtt{P}\ x\ e} \Rightarrow \boxed{\mathsf{NoCycle}\ x\ t}$ where $x \in \mathcal{FV}(t)$

Here t must be a constructor application and x appear as a subterm of t. This implements the occur-check of unification, relying on a NoCycle A instance proving that values in A are acyclic. The resulting subgoal must be discharged automatically by typeclass resolution, or simplification fails. NoCycle proofs can be produced from any WellFounded relation, in particular derived Subterm relations. E.g. for natural numbers, NoCycle x (S x) is equivalent to nat_subterm x (S x), which can easily be inhabited.

NoCycleRight handles the t = x case similarly, producing a NoCycle x t subgoal as well.

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

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.

SolutionRight is the symmetric case when the variable is on the right of the equality.

NoConfusion $\boxed{\forall(e : \mathtt{C}\ \bar{t} = \mathtt{D}\ \bar{u}),\ \mathtt{P}\ e}$

$\Rightarrow$
- $\forall\ (e : \mathsf{True}),\ \mathtt{P}\ (\mathsf{noConf\_inv}\ e)$ if C and D are the same constructor and all their arguments are forced.
- $\forall\ (e : \mathsf{False}),\ \mathtt{P}\ (\mathsf{noConf\_inv}\ e)$ if C and D are distinct constructors.
- $\forall\ (e : \bar{t}_{|n} = \bar{u}_{|n}),\ \mathtt{P}\ (\mathsf{noConf\_inv}\ e)$ otherwise, where $_{|n}$ restricts the vector of arguments to non-forced ones.

To implement this step, we use the NoConfusionPackage class that we are able to derive automatically (§4.1.4). As we favor the ***Remove sigma*** step, we will first try this rule to discriminate constructors, and look for an instance of homogeneous no-confusion. If no instance for this type is found, we will try the following rule.

The equality between $\bar{t}$ and $\bar{u}$ is, in general, an equality between telescopes, which will then be further simplified; when it is fully simplified, e and noConf_inv e will reduce to eq_refl.

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

In case the previous rule failed, we need to turn the equality into an equality of packed inductives. This step requires UIP on the type of the indices of the inductive type, or it will fail. The function ind_pack_inv is an opaque function (not simplified by the other steps) which goes back to the original equality between the values in the inductive family, making use of UIP. It also serves as a marker that a ***NoConfusion*** step involving UIP is in progress. Note that we do not generate a higher-dimensional equality between $e..1$ and a reflexivity proof as in §4.2: the uniqueness of identity proofs on the index type trivializes it. The goal produced is always amenable to an application of heterogeneous NoConfusion, which can always be derived, so we apply it eagerly and continue simplification.

Unpack $\boxed{\texttt{ind\_pack\_inv P eq\_refl}} \Rightarrow \boxed{\texttt{P eq\_refl}}$
To close a simplification started with ***Pack***, we use again the UIP proof of ind_pack_inv to simplify the goal: this is a non-definitional equivalence. In essence, it applies the reduction rule of UIP, saying that if A has unicity of proofs, then the type $x = y :> A$ does too.

True and ***False*** $\boxed{\texttt{∀(}e\texttt{ : True), P }e} \Rightarrow \boxed{\texttt{P I}}$ $\boxed{\texttt{∀(}e\texttt{ : False), P }e} \Rightarrow \boxed{\text{solved}}$
These steps are trivial and solve some goals produced by the ***NoConfusion*** step. D

We loop, applying these rules until we have solved the equality constraints entirely and return the resulting proof term and context map, or report an unsolvable constraint to the user.

There are two simplification steps which can make use of UIP on a given type: dependent ***Deletion***, as expected, and ***Pack*** which requires it on the indices of the inductive type. To enable these two rules, one must set a flag `Equations With UIP`. We can derive instances of UIP automatically using a `Derive EqDec for I` command for inductive types with decidable equality, as EqDec A implies UIP A, or the user can introduce his own instances. If we cannot find such a proof, we fail informing the user what instances are necessary. In all the examples presented in this paper we never relied on these two rules, but they can be useful, especially in proofs, see the example below.

Tactics. The simplification engine is independent from Equations, so we can use it to provide a tactic `simplify` that can simplify any goal with an equality between telescopes. The user can either let the tactic infer steps to apply, or specify a sequence of steps. This provides a combination of `discriminate`, `injection` and `subst`, plus acyclicity that can work with inductive families and optionally, UIP instances. Additionally, we developed a higher level dependent elimination tactic reusing the covering algorithm of Equations, i.e. taking a list of patterns as arguments. It provides a robust replacement to the `inversion` tactic. To see it in action, consider this proof on the inductive predicate representing ≤ on naturals, from Coq's standard library:

```
Inductive le (n : nat) : nat → Prop :=
| le_n : le n n
| le_S m : le n m → le n (S m).
Lemma le_UIP : ∀ (n m : nat) (p q : le n m), p = q.
Proof. intros n m p; induction p; intros q.
  dependent elimination q as [le_n|le_S m q]. reflexivity. admit.
```

The first case requires to invert an le n n proof dependently, which requires a proof of UIP nat.

5 PROOF PRINCIPLES

To generate the equations, unfolding and elimination principles of recursive definitions, we first instantiate their splitting tree by substituting any reference to a recursive definition by its implementation. As recursive functions cannot be split during pattern matching, only their types can change through refinement: they are morally just passed around everywhere. The type of structural prototypes is closed, while the type of well-founded prototypes is not: the proof argument gets refined by pattern matching. However, the defined function corresponding to a well-founded definition is itself closed and no longer quantifies on a proof that some relation holds between arguments, we can hence just ignore that argument during substitution. This gives a splitting tree with no recursion anymore. The equations of the programs correspond to the leaves of that splitting tree. We can also map that splitting tree to a Coq term corresponding to the 1-unfolded program in the case of well-founded recursion.

5.1 Unfolding lemmas

For a well-founded recursive function f, Equations defines an *unfolded* version of the function called f_unfold. Equations then proves automatically, by following the structure of the splitting tree, that f and f_unfold coincide at any point. The content of f_unfold is easier to manipulate than f because the "recursive" calls do not need to include the proofs that the recursive arguments decrease and it does not include an application of the well-founded recursion combinator: i.e. it really is non-recursive. The unfolding lemma for a function f has type $\forall \Delta, \mathsf{f}\ \overline{\Delta} = \mathsf{f\_unfold}\ \overline{\Delta}$, where f is directly an application of the well-founded recursion combinator FixWf. Using this lemma, we can express cleanly the elimination principle of f, abstracting away from the proofs used to justify its termination.

If we try to prove this lemma directly by induction, 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 an unfolding of the FixWf combinator and f. However the unfolding of FixWf is *not* convertible to f_unfold in general, 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 for unfolding equations. Hence, our proofs of unfolding for well-founded recursive definitions rely on the functional extensionality axiom. This is the only axiom used by Equations. One could also leave the extensionality of the functional as a proof obligation to the user, or attempt to prove it automatically, knowing that this will require intricate proofs in case of higher-order calls. We leave this for future work.

5.2 Elimination principle

For lack of space, we only sketch the construction of the elimination principle from the splitting tree, and refer to [Sozeau 2010] for details. Every (nested or mutual) program gives rise to a predicate. Every leaf of the program node gives rise to a method of the eliminator, where recursive calls produce induction hypotheses and calls to local `where` clauses produce hypotheses for their respective predicate. The `with` clauses essentially transfer a predicate from the enclosing program to their subprogram, adding an equality hypothesis.

6 RELATED AND FUTURE WORK

Cockx and Devriese [2018] present an improvement on the simplification of unification constraints for indexed datatypes avoiding more uses of UIP, and the resolution of higher-dimensional equations, implemented in Agda. 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 families with parameterized inductive types, 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. In general, Agda (and Idris) can handle all pattern-matching definitions we can write in Equations, but do not have specific support for well-founded recursion.

The technique of small inversions [Monin and Shi 2013] 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 [Avigad et al. 2017] essentially follows the same architecture as Equations, except it is restricted to toplevel clauses without `with` or `where` clauses, and does not generate elimination principles. As mentioned in the introduction, pattern-matching compilation is simplified by using definitional proof-irrelevance. It uses the Below construction to justify structurally recursive definitions, falling back to inference of a well-founded order in case this check fails (op. cit. §8.4). Lean handles nested and mutual inductive types by rewriting inductive definitions using an isomorphism with regular inductive definitions, resulting in back and forth translations. The translation appears to be partial: it cannot handle the definition of term from section 2.4 and requires to write a mutual type of term lists with its own map function[7]. We have not been able to handle the case of the abstraction constructor either in that case, Lean fails to check termination of the lifting and substitution functions. We believe this is mainly an implementation quirk, where the automation does not find the right termination measure.

The Function package [Barthe et al. 2006] of Coq also derives eliminators from well-founded definition and automatically proves the completeness of the graph, which we currently lack. It is also clever about handling overlapping in pattern-matchings, providing a graph that corresponds more closely to the shape of the definition entered by the user. One can use dependent pattern-matching on views to factorize cases similarly. The main advantage of Equations is that it allows definitions by dependent pattern-matching that Function cannot handle.

The Program package [Sozeau 2007] 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.

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. We also plan to integrate the size-change termination principle to handle a larger class of well-founded recursive definitions automatically. The dependent elimination tactic could be improved to give a dependent induction tactic, which requires applying simplification in induction hypotheses. 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 [Abel and Pientka 2016]. Finally, given the proximity of Equations and Haskell definitions, Equations could provide a better back-end to the `hs-to-coq` tool [Spector-Zabusky et al. 2018] for the verification of Haskell programs in Coq.

CONCLUSION

We presented a full-featured definitional extension of Coq, which makes developing 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 a stand-alone dependent pattern-matching compiler,

[7] material/lean/nested.lean tested with Lean 3.4.2

based on simplification of equalities of telescopes and homogeneous no-confusion. It can be reused to implement a robust dependent elimination tactic.

REFERENCES

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

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

Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. 2017. Normalization by evaluation for sized dependent types. *PACMPL* 1, ICFP (2017), 33:1–33:30. https://doi.org/10.1145/3110277

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational Equality, Now!. In *PLPV'07*. Freiburg, Germany.

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

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.

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

Edwin Brady, Conor McBride, and James McKinna. 2003. Inductive Families Need Not Store Their Indices.. In *TYPES (Lecture Notes in Computer Science)*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.), Vol. 3085. Springer, 115–129.

Adam Chlipala. 2011. *Certified Programming with Dependent Types*. Vol. 20. MIT Press.

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

Jesper Cockx and Andreas Abel. 2018. Elaborating dependent (co)pattern matching. *PACMPL* 2, ICFP (2018), 75:1–75:30. https://doi.org/10.1145/3236770

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

Jesper Cockx and Dominique Devriese. 2018. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *J. Funct. Program.* 28 (2018), e12. https://doi.org/10.1017/S095679681800014X

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

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

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages* (Jan. 2019), 1–28. https://doi.org/10.1145/329031610.1145/3290316

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

Peter Hancock. 2000. *Ordinals and Interactive Programs*. Ph.D. Dissertation. LFCS. http://www.lfcs.inf.ed.ac.uk/reports/00/ECS-LFCS-00-421/index.html

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

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the correctness of reactive systems using sized types. In *POPL*, Vol. 96. 410–423.

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

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

Assia Mahboubi, Enrico Tassi, Yves Bertot, and Georges Gonthier. 2018. *Mathematical Components*.

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.

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

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

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

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

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

C. Paulin-Mohring. 1996. *Définitions Inductives en Théorie des Types d'Ordre Supérieur.* Habilitation à diriger les recherches. Université Claude Bernard Lyon I. http://www.lri.fr/~paulin/PUBLIS/habilitation.ps.gz

Pierre-Marie Pédrot and Nicolas Tabareau. 2018. Failure is Not an Option An Exceptional Type Theory. In *ESOP 2018 - 27th European Symposium on Programming (LNCS)*, Vol. 10801. Springer, Thessaloniki, Greece, 245–271. https://doi.org/10.1007/978-3-319-89884-1_9

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

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *PACMPL* 2, POPL (2018), 16:1–16:34. https://doi.org/10.1145/3158104

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

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

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

Thomas Streicher. 1993. *Semantical Investigations into Intensional Type Theory.* Habilitationsschrift. LMU München.

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

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

Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. 2017. A tale of two provers: verifying monoidal string matching in liquid Haskell and Coq. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, Iavor S. Diatchki (Ed.). ACM, 63–74. https://doi.org/10.1145/3122955.3122963

Dimitrios Vytiniotis, Thierry Coquand, and David Wahlstedt. 2012. Stop When You Are Almost-Full - Adventures in Constructive Termination. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings (Lecture Notes in Computer Science)*, Lennart Beringer and Amy P. Felty (Eds.), Vol. 7406. Springer, 250–265. https://doi.org/10.1007/978-3-642-32347-8_17

7 APPENDIX

7.1 Sized-based measure library

We use a type class Sized for sizes on arbitrary types.

```
Class Sized (A : Type) := size : A → nat.
```

For lists, we must be careful to define the sizing function so that it takes *SizeA* as a parameter so that later size functions can use it in a nested manner and satisfy the guardness check. To do so we define it in a section.

```
Section list_size. Context {A : Type} {SizeA : Sized A}.
  Equations list_size : Sized (list A) by struct :=
  list_size nil := 0;
  list_size (cons x xs) := S (size x + list_size xs).
End list_size.
```

The following section derives a generic map_size function for Sized types.

```
Section map_size. Context {A : Type} {SizeA : Sized A}.
  Equations? map_size {B} (l : list A) (g : ∀ (x : A), size x < size l → B) : list B :=
  map_size nil _ := nil;
  map_size (cons x xs) g := cons (g x _) (map_size xs (fun x H ⇒ g x _)).
  Proof. all:cbn;lia. Qed.
```

It has the expected spec when the function does not use the size information.

```
  Lemma map_size_spec {B} (g : A → B) (l : list A) : map_size l (fun x _ ⇒ g x) = List.map g l.
```

This proves a stronger specification for map witnessing that it passes smaller arguments to its function argument g.

```
  Lemma map_size_transform {B} (g : A → B) (l : list A) (P : A → B → Prop) :
    (∀ a (Ha : size a < size l), P a (g a)) → Forall2 P l (map g l).
End map_size.
```

7.2 Size-based nested well-founded recursion

The term_size function uses the oveloaded *list_size* above.

```
Equations term_size {n} : Sized (term n) by struct :=
  term_size (Var v) := 1;
  term_size (Lam t) := S (size t);
  term_size (App t l) := S (size t + size l).
```

Finally subst_term can be defined using recursion on sizes.

```
Equations? subst {k l} (σ : fin k → term l) (t : term k) : term l by wf (term_size t) lt := {
  (Var v) [σ] ⇒ σ v;
  (Lam t) [σ] ⇒ Lam (t [extend_var σ]);
  (App t l) [σ] ⇒ App (t [σ]) (map_size l (fun t Inl ⇒ t [σ])) }
where "t [ σ ]" := (subst σ t) : term.
Proof. all:unfold size in *; simpl; lia. Defined.
Hint Rewrite @map_size_spec : subst.
```

An eliminator talking only about map can also be derived if desired.

```
Lemma subst_term'_elim_all :
  ∀ (P : ∀ k l : nat, (fin k → term l) → term k → term l → Prop),
  (∀ k l σ (f : fin k), P k l σ (Var f) (σ f)) →
  (∀ k l σ (t : term (S k)), P (S k) (S l) (extend_var σ) t (t [extend_var σ]) →
    P k l σ (Lam t) (Lam (t [extend_var σ]))) →
  (∀ k l σ (t : term k) (ts : list (term k)), P k l σ t (t [σ]) →
    Forall2 (P k l σ) ts (map (subst σ) ts) →
    P k l σ (App t ts) (App (t [σ]) (map (subst σ) ts))) →
  ∀ k l σ t, P k l σ t (t [σ]).
Proof. intros P ????. apply (subst_elim _); intros *; auto.
         intros Ht Hl; generalize (map_size_transform _ _ _ Hl); simp subst.
Qed.
```

7.3 Mutual well-founded recursion through GADTs

We present an instance of this pattern for the term substitution functions which is actually nested, but it can apply to arbitrary nested or mutual definitions. The subst_ty family encodes the types of our functions.

```
Inductive subst_ty : ∀ (A : Type) (P : A → Type), Type :=
| tysubst : subst_ty (Σ {k l} (σ : fin k → term l) , term k) (fun a ⇒ term a.2.1)
| tysubsts : subst_ty (Σ {k l} (σ : fin k → term l) , list (term k)) (fun a ⇒ list (term a.2.1)).
```

We will use a simple measure, assuming an overloaded size function and definition of size for terms and lists of sized objects.

```
Equations measure {A P} (t : subst_ty A P) (x : A) : nat :=
  measure tysubst (_, _, _, t) ⇒ size t;
  measure tysubsts (_, _, _, l) ⇒ size l.
```

We define the function by recursion on the abstract packed argument according to this measure. Using dependent pattern matching, the clauses for tysubst refine the argument and return type to match the type of *subst_term* and similarly for tysubsts, we can hence do pattern-matching as usual on the actual arguments. Termination is easily proven by reasoning on sizes.

```
Equations? subst {A P} (t : subst_ty A P) (x : A) : P x by wf (measure t x) lt := {
  (Var v) [σ] ⇒ σ v;
  (Lam t) [σ] ⇒ Lam (t [extend_var σ]);
  (App t l) [σ] ⇒ App (t [σ]) (subst tysubsts (_, _, σ, l));
  subst t1 (k, l, σ, ts) ⇒ map_size ts (fun t Ints ⇒ t [σ] : term l) }
where "t [ σ ]" := (subst tysubst (_, _, σ, t)) : term.
Proof. all:repeat (unfold size; simp term_size); lia. Defined.
```

7.4 Well-founded nested recursion as mutual recursion with sizes

The following example uses just dependent elimination on a finite type (booleans) and shows that this also applies to nested recursive definitions. We use a simpler type of rose trees here.

We first define rose trees and their Sized instance.

```
Section RoseMut. Context {A : Set}.
  Inductive t : Set :=
  | leaf (a : A) : t
  | node (l : list t) : t.

  Equations t_size : Sized t by struct :=
  t_size (leaf _) := 0;
  t_size (node l) := S (size l).
```

The measure just takes the size. Here we encode the mutuality using branching on a boolean.

```
  Equations mutmeasure (b : bool) (arg : if b then t else list t) : nat :=
  mutmeasure true t := size t;
  mutmeasure false lt := size lt.
```

The argument and return type depend on the function label (true or false here) and any well-founded recursive call is allowed.

```
  Equations? elements (b : bool) (x : if b then t else list t) : if b then list A else list A
    by wf (mutmeasure b x) lt :=
  elements true (leaf a) := [a];
  elements true (node l) := elements false l;
  elements false nil := nil;
  elements false (cons t ts) := elements true t ++ elements false ts.
  Proof. all:cbn; lia. Qed.
```

Dependent return types also possible of course. elements_dep is a trivial copy of elements that additionally shows it is computing a sublist of elements. It requires a dependent return type elements_dep_type.

```
Equations elements_dep_type (b : bool) (x : if b then t else list t) : Set :=
  elements_dep_type true t := { l' : list A | ∀ x, In x l' → In x (elements true t) };
  elements_dep_type false l := { l' : list A | ∀ x, In x l' → In x (elements false l) }.
```

We put ourselves in *Program* mode to have coercions of subset types. We reset the default obligation tactic which is otherwise applied to initial goals when using Equations?.

```
Obligation Tactic := idtac.
Set Program Mode.

Equations? elements_dep (b : bool) (x : if b then t else list t) : elements_dep_type b x
  by wf (mutmeasure b x) lt :=
elements_dep true (leaf a) := [a];
elements_dep true (node l) := elements_dep false l;
elements_dep false nil := nil;
elements_dep false (cons t ts) := elements_dep true t ++ elements_dep false ts.
Proof.
  all:(simp mutmeasure elements).
  2-3:(cbn; lia). all:destruct elements_dep; simpl. apply i.
  destruct elements_dep;simpl. intros. rewrite app_in in *. intuition.
Qed.

End RoseMut.
```