Programming with Dependent Types in Coq: Inductive Families and Dependent Patter-Matching

Matthieu Sozeau, $\pi.r^2$, Inria Paris & IRIF

CASS 2020 Summer School
January 7th 2020
Chile

Dependently-Typed Programming in Coq

Indexed datatypes: vectors

Size-indexed version of lists.

$\texttt{Inductive}\ \mathsf{vec}\ (A : \texttt{Type}) : \mathsf{nat} \to \texttt{Type} :=$
$|\ \mathsf{nil} : \mathsf{vec}\ A\ 0$
$|\ \mathsf{cons}\ n : A \to \mathsf{vec}\ A\ n \to \mathsf{vec}\ A\ (\mathsf{S}\ n).$

The empty vector nil has size O while the cons operation increments the size by one.

- Indexed
- Recursive
- Terms *and* types carry more information

Notations

We declare notations similar to lists on vectors, as the size information will generally be left *implicit*.

```
Arguments nil {A}.
Arguments cons {A n}.

Notation "x :: v" := (cons x v) : vector_scope.
Notation "[ ]" := nil : vector_scope.
Notation "[ x ]" := (cons x nil) : vector_scope.
Notation "[ x ; y ; .. ; z ]" := (cons x (cons y .. (cons z nil) .. ))
: vector_scope.

Example v3 : vec bool 3 :=
 @cons bool 2 true (@cons _ 1 true (@cons _ 0 false nil)).

Example v3' : vec bool 3 :=
 cons true (cons true (cons false nil)).
```

Recall return clauses

```
Fixpoint vmap {A B} {n} (f : A → B) (v : vec A n) : vec B n
:= match v in vec _ k return vec B k with
    | nil ⇒ nil
    | cons n a v' ⇒ cons (f a) (vmap f v')
    end.
```

Recall return clauses

```
Fixpoint vmap {A B} {n} (f : A → B) (v : vec A n) : vec B n
:= match v in vec _ k return vec B k with
    | nil ⇒ nil
    | cons n a v' ⇒ cons (f a) (vmap f v')
   end.
```

```
Definition vhead {A} {n} (v : vec A (S n)) : A :=
  match v in vec _ k
    return match k with 0 ⇒ unit | S k ⇒ A end with
    | nil ⇒ tt
    | cons n a v' ⇒ a
  end.
```

We are encoding with the `match` in the return clause the discrimination of 0 and S n.

With equality instead:

```
Program Definition vhead_eq {A} {n} (v : vec A (S n)) : A :=
  match v in vec _ k return k = S n → A with
  | nil ⇒ fun (eq : 0 = S n) ⇒ False_rect _ _
  | cons n' a v' ⇒ fun (eq : S n' = S n) ⇒ a
  end (@eq_refl _ (S n)).
```

Same problem, we need to explicitely witness equality manipulations in the branches.

- Highly complicated!
- Obscures the computational content with these "commutative cuts" and coercions

Programming with dependent pattern-matching

- Mixes proofs/invariants with datastructures
- Advantage: less partiality and “garbage” representations, invariants are explicit
- Disadvantage: more involved definitions and proofs.
- Need for reasoning on equalities to justify inversions in general (fancy return clauses are not enough).
- Certified Programming with Dependent Types (Chlipala, 2011) goes into many tricks needed to program with these types in Coq.
- Equations: higher-level notation for writting these programs, close to Agda/Idris syntax.
  - Equations *embeds* the equational theory of inductive types in the pattern-matching algorithm.
  - Lets you focus on the program rather than making it type-check!

Introduction to EQUATIONS

In Coq!

`equations_intro.v`

www: http://mattam82.github.io/Coq-Equations/

Inductive predicates

Judgments and in general inductively defined derivation trees can be represented using indexed inductive types.
A typical de Bruijn encoding of STLC.

```
Inductive type :=
  | cst | arrow : type → type → type.
Inductive term :=
  | var : nat → term
  | lam : type → term → term
  | app : term → term → term.
Definition ctx := list type.
```

Judgments

The typing relation can be defined as the inductive family:

```
Inductive typing : ctx → term → type → Prop :=
| Var : ∀ (G : ctx) (x : nat) (A : type),
    List.nth_error G x = Some A →
    typing G (var x) A

| Abs : ∀ (G : ctx) (t : term) (A B : type),
    typing (A :: G) t B →
    typing G (lam A t) (arrow A B)

| App : ∀ (G : ctx) (t u : term) (A B : type),
    typing G t (arrow A B) →
    typing G u A →
    typing G (app t u) B.
```

Generalizing by equalities

Suppose you want to show:

Lemma invert_var Γ *x* *T* (*H* : typing Γ (var *x*) *T*) :
  List.nth_error Γ *x* = Some *T*.
Proof. elim: *H* ⇒ [*G x' A Hnth*|*G t A B HB*|*G t u A B HAB HA*].

  x : nat
  G : ctx
  x' : nat
  A : type
  H : nth_error *G x'* = Some *A*
  ============================
   nth_error *G x* = Some *A*
subgoal 2 (*ID* 384) is:
 nth_error *G x* = Some (arrow *A B*)
subgoal 3 (*ID* 394) is:
 nth_error *G x* = Some *B*

Generalizing by equalities

```
Lemma invert_var Γ x T (H : typing Γ (var x) T) :
  List.nth_error Γ x = Some T.
Proof.
  inversion H. subst. assumption.
Qed.
```

Generalizing by equalities

- Generalizing by equalities to keep information that is otherwise lost by the eliminator.
- Generalizes the return clause
  $\texttt{match}\ t\ \texttt{return}\ P\ \texttt{with}$
  into

  $\texttt{match}\ t\ \texttt{as}\ v\ \texttt{return}\ t = v \rightarrow P\ \texttt{with}$
  $\mid S\ y \Rightarrow \texttt{fun}\ H : t = S\ y \Rightarrow \ldots$
  $\mid \ldots$
  $\texttt{end}$
- For full generality, pack inductive value with its indices in a sigma-type:

  $\texttt{match}\ (t : I\ u)\ \texttt{in}\ I\ i\ \texttt{as}\ v\ \texttt{return}\ (u; t) =_{\_}\{i\ \&\ I\ i\}\ (i; v)$
  $\rightarrow P\ \texttt{with}$

  ...

Understanding inversion

```
Lemma invert_var' Γ x T (H : typing Γ (var x) T) :
  List.nth_error Γ x = Some T.
Proof.
  remember (var x) as t. move:Heqt.
```

Γ: ctx

x: nat

T: type

t: term

H: typing Γ t T

=================

$t = \text{var } x \to \text{nth\_error } \Gamma\ x = \text{Some } T$

Information is kept!

Goal: $t = \mathsf{var}\ x \rightarrow \mathrm{nth\_error}\ \Gamma\ x = \mathsf{Some}\ T$

`elim`: $H \Rightarrow [G\ x'\ A\ Hnth | G\ b\ A\ B\ HB | G\ f\ u\ A\ B\ HAB\ HA]$.

G: ctx
x': `nat`
A: `type`
$Hnth$: $\mathrm{nth\_error}\ G\ x' = \mathsf{Some}\ A$

$\mathsf{var}\ x' = \mathsf{var}\ x \rightarrow \mathrm{nth\_error}\ G\ x = \mathsf{Some}\ A$

$\ldots \rightarrow \mathsf{lam}\ A\ b = \mathsf{var}\ x \rightarrow \mathrm{nth\_error}\ G\ x = \mathsf{Some}\ (\mathsf{arrow}\ A\ B)$

$\ldots \rightarrow \mathsf{app}\ f\ u = \mathsf{var}\ x \rightarrow \mathrm{nth\_error}\ G\ x = \mathsf{Some}\ B$

Specialization by unification

In general we simplify the resulting equations according to:

- substitution (a.k.a. *eq_rect* rule):
  $(\forall\ (y : \mathsf{nat})\ (e : y = t) \to P\ y\ e) \simeq P\ t\ eq\_refl\ (y \notin FV(t))$
- injectivity: $(S\ u = S\ v \to P) \simeq (u = v \to P)$
- discrimination: $(0 = 1 \to P) \simeq P$
- acyclicity; $(y = c\ y \to P) \simeq P$
- *deletion* (a.k.a. axiom K):
  $(\forall\ (y : \mathsf{nat})\ (e : y = y) \to P\ y\ e) \simeq (\forall\ (y : \mathsf{nat}), P\ y\ eq\_refl)$

Pattern-matching and unification

Idea: reasoning up-to the theory of equality and constructors

Example: to eliminate t : typing Γ (var x) T, we unify with:

1. typing Γ' (var x') T' for Var Γ' x' T'
2. typing Γ' (lam A' t') (arrow A' B') for Abs
3. typing Γ' (app t' u') B' for App

Unification $t \equiv u \rightsquigarrow Q$ can result in:

- $Q = \texttt{Fail}$
- $Q = \texttt{Success}\ \sigma$ (with a substitution σ);
- $Q = \texttt{Stuck}\ t$ if t is outside the theory (e.g. a constant)

Pattern-matching and unification

Idea: reasoning up-to the theory of equality and constructors

Example: to eliminate t : typing Γ (var x) T, we unify with:

1. typing Γ' (var x') T' for Var Γ' x' T'
   $\leadsto$ `Success` $[\Gamma' := \Gamma, x' := x, T' := T]$
2. typing Γ' (lam A' t') (arrow A' B') for Abs
3. typing Γ' (app t' u') B' for App

Unification $t \equiv u \leadsto Q$ can result in:

- $Q =$ `Fail`
- $Q =$ `Success` σ (with a substitution σ);
- $Q =$ `Stuck` t if t is outside the theory (e.g. a constant)

Pattern-matching and unification

Idea: reasoning up-to the theory of equality and constructors

Example: to eliminate t : typing Γ (var x) T, we unify with:

1. typing Γ' (var x') T' for Var Γ' x' T'
   $\rightsquigarrow$ `Success` $[\Gamma' := \Gamma, x' := x, T' := T]$
2. typing Γ' (lam A' t') (arrow A' B') for Abs $\rightsquigarrow$ `Fail`
3. typing Γ' (app t' u') B' for App $\rightsquigarrow$ `Fail`

Unification $t \equiv u \rightsquigarrow Q$ can result in:

- $Q =$ `Fail`
- $Q =$ `Success` σ (with a substitution σ);
- $Q =$ `Stuck` t if t is outside the theory (e.g. a constant)

Unification rules

$$\frac{x \notin \mathcal{FV}(t)}{x \equiv t \rightsquigarrow \texttt{Success}\ \sigma[x := t]}\ \textsc{Solution}$$

$$\frac{C \text{ constructor context}}{x \equiv C[x] \rightsquigarrow \texttt{Fail}}\ \textsc{Cycle} \qquad \frac{}{\mathsf{C}\ \_ \equiv \mathsf{D}\ \_ \rightsquigarrow \texttt{Fail}}\ \textsc{Discrimination}$$

$$\frac{t_1 \ldots t_n \equiv u_1 \ldots u_n \rightsquigarrow Q}{\mathsf{C}\ t_1 \ldots t_n \equiv \mathsf{C}\ u_1 \ldots u_n \rightsquigarrow Q}\ \textsc{Injectivity}$$

$$\frac{p_1 \equiv q_1 \rightsquigarrow \texttt{Success}\ \sigma \qquad (p_2 \ldots p_n)\sigma \equiv (q_2 \ldots q_n)\sigma \rightsquigarrow Q}{p_1 \ldots p_n \equiv q_1 \ldots q_n \rightsquigarrow Q \cup \sigma}\ \textsc{Patterns}$$

$$\frac{}{t \equiv t \rightsquigarrow \texttt{Success}\ []}\ \textsc{Deletion} \qquad \frac{\text{Otherwise}}{t \equiv u \rightsquigarrow \texttt{Stuck}\ u}\ \textsc{Stuck}$$

Pattern-Matching Compilation

Pattern-matching compilation uses unification to:

- Decide which program clause to choose
- Decide which constructors can apply when we eliminate a variable in an indexed family.

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

$\mathsf{cover}(m\ n : \mathsf{nat} \vdash m\ n)$

Pattern-Matching Compilation

Pattern-matching compilation uses unification to:

- Decide which program clause to choose
- Decide which constructors can apply when we eliminate a variable in an indexed family.

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

$\mathsf{cover}(m\ n : \mathsf{nat} \vdash m\ n) \to \mathsf{O}\ \mathsf{O} \equiv m\ n \rightsquigarrow \mathtt{Stuck}\ m$

Pattern-Matching Compilation

Pattern-matching compilation uses unification to:

- Decide which program clause to choose
- Decide which constructors can apply when we eliminate a variable in an indexed family.

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

Split($m\ n$: nat $\vdash m\ n$, m, [ ])

Pattern-Matching Compilation

Pattern-matching compilation uses unification to:

- Decide which program clause to choose
- Decide which constructors can apply when we eliminate a variable in an indexed family.

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

Split($m\ n$: nat $\vdash n\ m$, m, [
  cover(n : nat $\vdash$ O n)
  cover($m'\ n$: nat $\vdash$ (S m') n)])

Pattern-Matching Compilation

Pattern-matching compilation uses unification to:

- Decide which program clause to choose
- Decide which constructors can apply when we eliminate a variable in an indexed family.

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

Split($m\ n$: nat $\vdash m\ n$, m, [
  Split(n : nat $\vdash$ O n, n, [
    Compute($\vdash$ O O $\Rightarrow$ true),
    Compute(n' : nat $\vdash$ O (S n') $\Rightarrow$ false)]),
  cover($m'\ n$: nat $\vdash$ (S m') n)])

Pattern-Matching Compilation

Pattern-matching compilation uses unification to:

- Decide which program clause to choose
- Decide which constructors can apply when we eliminate a variable in an indexed family.

Overlapping clauses and first-match semantics:

```
Equations equal (m n : nat) : bool :=
  equal O O := true;
  equal (S m') (S n') := equal m' n';
  equal m n := false.
```

Split($m\ n$: nat $\vdash m\ n$, m, [
  Split(n : nat $\vdash$ O n, n, [
    Compute($\vdash$ O O $\Rightarrow$ true),
    Compute(n' : nat $\vdash$ O (S n') $\Rightarrow$ false)]),
  Split($m'\ n$: nat $\vdash$ (S m') n, n, [
    Compute(m' : nat $\vdash$ (S m') O $\Rightarrow$ false),
    Compute($m'\ n'$: nat $\vdash$ (S m') (S n') $\Rightarrow$ equal $m'\ n'$)])])

Dependent pattern-matching

```
Inductive vector (A : Type) : nat → Type :=
| nil : vector A 0
| cons {n : nat} : A → vector A n → vector A (S n).
Equations tail A n (v : vector A (S n)) : vector A n :=
  tail A n (@cons ?(n) _ v) := v.
```

Each variable must appear only once, except in inaccessible patterns.

cover($A\ n\ v$: vector A (S n)) $\vdash A\ n\ v$)

Dependent pattern-matching

```
Inductive vector (A : Type) : nat → Type :=
| nil : vector A 0
| cons {n : nat} : A → vector A n → vector A (S n).
Equations tail A n (v : vector A (S n)) : vector A n :=
  tail A n (@cons ?(n) _ v) := v.
```

Each variable must appear only once, except in inaccessible patterns.

Split($A\ n\ (v : \mathsf{vector}\ A\ (\mathsf{S}\ n)) \vdash A\ n\ v$, v, [
  Fail; // $\mathsf{O} \neq \mathsf{S}\ n$
  cover($A\ n'\ a\ (v' : \mathsf{vector}\ A\ n') \vdash A\ n'\ (@\mathsf{cons}\ ?(n')\ a\ v')$)])

Dependent pattern-matching

```
Inductive vector (A : Type) : nat → Type :=
| nil : vector A 0
| cons {n : nat} : A → vector A n → vector A (S n).
Equations tail A n (v : vector A (S n)) : vector A n :=
  tail A n (@cons ?(n) _ v) := v.
```

Each variable must appear only once, except in inaccessible patterns.

Split($A\ n\ (v :$ vector A (S n)) $\vdash A\ n\ v$, v, [
 Fail; // S $n \neq$ O
 Compute($A\ n'\ a\ (v' :$ vector $A\ n') \vdash A\ n'$ (@cons ?(n') $a\ v'$)
 $\Rightarrow v'$)])

It's all the same

- Inductive families vs subset types.
- Structure vs property.

$\texttt{Definition}\ \mathrm{ilist}\ A\ n := \{l\ \texttt{:}\ \mathsf{list}\ A \mid \mathrm{length}\ l\ \texttt{=}\ n\}.$

It's all the same

- Inductive families vs subset types.
- Structure vs property.

```
Definition ilist A n := {l : list A | length l = n}.
```

Let's show this type is *isomorphic* to vectors.

```
Record Iso (A B : Type) :=
  { iso_lr : A → B; iso_rl : B → A;
    iso_lr_rl : ∀ x, iso_lr (iso_rl x) = x;
    iso_rl_lr : ∀ x, iso_rl (iso_lr x) = x }.
```

Exercise

```
Program Fixpoint vect_ilist {A n} (v : vec A n) : ilist A n :=
  match v in vec _ n return ilist A n with
  | nil ⇒ Datatypes.nil
  | cons n x xs ⇒ Datatypes.cons x (vect_ilist xs)
  end.

Fixpoint ilist_vect {A} (l : list A) : vec A (length l) :=
  exercise.

Program Definition vect_ilist_iso {A} (n : nat) :
  Iso (vec A n) (@ilist A n) :=
  { iso_lr := fun x ⇒ vect_ilist x ;
     iso_rl := fun x ⇒ ilist_vect x }.
Solve Obligations with Exercise.
```

The relationship can be made explicit, categorically or using a universe of datatypes: Ornaments (Dagand and McBride, 2013; Dagand, 2017).

More examples

- Matrices, any bounded datastructure

```
Definition square_matrix {A} n := vec (vec A n) n.
```

- Balancing/shape invariants: e.g. red-black trees.
- Type-preserving evaluators (equations_evaluator.v, with an exercise)
- See equations_exercises.v for some more!

A little history

Many flavors of inductive families and DPM.

- DML (Xi and Pfenning, 1999): ML + integer indexed types (presburger arithmetic)
- Agda (Norell, 2007), Epigram (McBride, 2005). UIP rule for non-linear cases and a higher level construction
- Agda (Cockx), Equations (Sozeau). Avoid the UIP rule, staying compatible with HoTT.
- Haskell, OCaml GADTs: indices can be types only, not arbitrary terms.
- F* (Swamy et al., 2016): indices can be values, subset types à la PVS (no proof terms)
- CoqMT (Blanqui et al., 2007): Coq Modulo Theories, conversion includes arbitrary decidable theories. No coercions!

And many others: ATS (Xi), Beluga (Pientka), Ωmega (Sheard), Trellys (Weirich), ...

Bibliography

On dependent pattern-matching and inductive families in Dependent Type Theory:

- Paulin-Mohring (1993): Inductive types in the Coq system Coq.
- Goguen et al. (2006). The notion of generalization by equalities and simplification procedure. McBride's papers include a large number of examples.
- Cockx and Devriese (2018) and Sozeau and Mangin (2019): state of the art in Agda and Coq. This allows to do pattern-matching without the K/UIP rule, incompatible with Univalence.

Bibliography

Frédéric Blanqui, Jean-Pierre Jouannaud, and Pierre-Yves Strub. Building Decision Procedures in the Calculus of Inductive Constructions. In Jacques Duparc and Thomas A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2007.

Adam Chlipala. Certified Programming with Dependent Types, volume 20. MIT Press, 2011.

Jesper Cockx and Dominique Devriese. *Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory*. *J. Funct. Program.*, 28:e12, 2018.

Pierre-Évariste Dagand. *The essence of ornaments*. *J. Funct. Program.*, 27:e9, 2017.

Pierre-Évariste Dagand and Conor McBride. A Categorical Treatment of Ornaments. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 530–539. IEEE Computer Society, 2013.

Healfdene Goguen, Conor McBride, and James McKinna. Eliminating Dependent Pattern Matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.

Conor McBride. *Epigram: Practical Programming with Dependent Types*. *Advanced Functional Programming*, pages 130–170, 2005.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.

Matthieu Sozeau and Cyprien Mangin. *Equations Reloaded: High-Level Dependently-Typed Programming and Proving in Coq*. *PACMPL*, 3(ICFP):86–115, 2019.

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 214–227, January 1999.