



A Gentle Introduction to Equations Or How to Match Regexp with Dependently-Typed Continuations

Matthieu Sozeau & Cyprien Mangin
Inria Paris & IRIF, Université Paris 7 Diderot

Saarland University
Saarbrücken
April 26th 2018

OUTLINE

1. Dependent Pattern-Matching 101
 - a. History & Examples
 - b. Unification
 - c. Axiomatic vs Definitional Extensions
 - d. Uniqueness of Identity Proofs

2. Equations Tutorial: Regex Matching
 - a. Tool presentation
 - b. A simply-typed version
 - c. Indexing to the rescue
 - d. Boilerplate-free proofs on dependently-typed programs

DPM 101

“Pattern-Matching with Dependent Types”, Coquand, 1992

The type-value dependency circle:

Inductive $\text{vector } A : \text{nat} \rightarrow \text{Type} :=$
| $\text{Vnil} : \text{vector } A 0$
| $\text{Vcons} : \text{forall } (h:A) (n:\text{nat}), \text{vector } A n \rightarrow \text{vector } A (S n).$

Equations $\text{vtail } \{A n\} (v : \text{vector } A (S n)) : \text{vector } A n :=$
 $\text{vtail } (\text{Vcons } a n v') := v'.$

DPM 101

“The view from the left”, McBride, McKinna - JFP, 2004

The “with” rule:

```
Equations equal (n m : nat) : { n = m } + { n ≠ m } :=  
equal O O := left eq_refl ;  
equal (S n) (S m) with equal n m := {  
  equal (S n) (S ?(n)) (left eq_refl) := left eq_refl ;  
  equal (S n) (S m) (right p) := right _ } ;  
equal x y := right _.
```

⇒ Later adopted in Agda 2/Epigram

DPM 101 - Splitting and Matching

When checking that a definition with clauses

$$f_j \text{ p}_{j1} \dots \text{ p}_{jn} := \dots$$

covers the typing:

$$f : \forall (x_1 : \tau_1 \dots x_n : \tau_n), \tau$$

We **match** $(p_1 \dots p_n, x_1 \dots x_n)$

$$\begin{aligned} \text{match}(p_1..p_n, t_1..t_n) &\sim \mathbf{Unif} \ \sigma \quad (\sigma \text{ substitution}) \\ &| \mathbf{Fail} \quad | \mathbf{Stuck} \ x \quad (x \text{ variable}) \end{aligned}$$

- Standard first-match semantics
- $\mathbf{Stuck} \ x_i$ tells which variable to split next, refining x_i to constructors

DPM 101 - Unification

When checking that a clause

$$f(C_i p_1 \dots p_n) q_2 \dots q_m := \dots$$

matches the prototype:

$$f : \forall (x_1 : \tau_1 \dots x_n : \tau_n), \tau$$

we unify: $\tau_1 \dots \tau_n$ **and** $\tau_1 \dots \tau_n$

$$t =? u \rightsquigarrow \begin{array}{l} \mathbf{Unif} \ \sigma \quad (\sigma \text{ substitution}) \\ | \mathbf{Fail} \ | \ \mathbf{Stuck} \end{array}$$

Let's write the rules on the whiteboard!

DPM 101 - Axiomatic vs Definitional Extension

- **Axiomatic** extension: matching and unification part of the kernel (Alfa, Agda 2), trusted
- **Definitional** extension: compile pattern-matching to primitive eliminators (Epigram, Equations, Lean, GHC)
E.g. in Coq:

```
Definition vtail (x : vector A (S n)) : vector A n :=  
  match x in vec y return match y with 0 => unit | S n => vector A n end with  
  | Vnil => tt  
  | Vcons a n v => v  
end
```

This kind of “small” inversions is however limited
(linearity, pure pattern structure, limited dependencies)

DPM 101 - Compilation

“Eliminating Dependent Pattern-Matching”

Goguen, McBride & McKinna, 2006

Main Idea: unification

$$t =? u \rightsquigarrow \text{Unif } \sigma$$

is **witnessed** by a proof of:

$$t = u \rightarrow |\sigma|$$

where $|x:=t;\sigma| \equiv \Sigma p : x = t, |\sigma[p]|$

DPM 101 - Eliminating a variable

Start with: $\Gamma_0 (x : l \bar{t}) \Gamma_1$

Generalize:

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

Eliminate:

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

Simplify equalities

DPM 101 - Compilation

Supporting definitions: J-rule, NoConfusion, NoCycle, K

Specialization by Unification (McBride, PhD'99, Cockx, ICFP'14). Gradually refine/simplify:

$$t = _ (\Sigma \delta : \Delta . I \delta) u \rightarrow \tau$$

Theorem (computational soundness):

Each clause $f \ c1 \ .. \ cn := t$ turns into a

reduction $f \ c1 \ .. \ cn \rightarrow t$, even on open terms.

(When there are no overlapping patterns)

DPM 101 - Simplification

Sigma:

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

Solution:

$$\boxed{\forall \Gamma, \forall (e : x = t), P x e} \Rightarrow \boxed{\forall \Gamma', P t \text{ eq_refl}}$$

Deletion:

$$\boxed{\forall (e : t = t), P e} \Rightarrow \boxed{P \text{ eq_refl}}$$

DPM 101 - Constructor Simplification

Pack:

$$\boxed{\forall(e : C \bar{t} = D \bar{u}), P e} \Rightarrow \boxed{\forall(e : (\overline{idx_t}, C \bar{t}) = (\overline{idx_u}, D \bar{u})), \text{ind_pack_inv } P e}$$

NoConfusion:

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

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

Unpack:

$$\boxed{\text{ind_pack_inv } P \text{ eq_refl}} \Rightarrow \boxed{P \text{ eq_refl}}$$

True and False:

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

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

DPM 101 - Uniqueness of Identity Proofs

- Supported by the unification
- Incompatible with Homotopy Type Theory models
- Can remove the “deletion” rule to disallow it.
- More refined solutions possible (Cockx et al, ICFP'14, CPP'17)
- Equations allow configuration of behavior

OUTLINE

1. Dependent Pattern-Matching 101

- a. History & Examples
- b. Unification
- c. Axiomatic vs Definitional Extensions
- d. Uniqueness of Identity Proofs

2. Equations Tutorial: Regex Matching

- a. Tool presentation
- b. A simply-typed version
- c. Indexing to the rescue
- d. Boilerplate-free proofs on dependently-typed programs

Equations Reloaded

- Dependent Pattern-Matching à la Epigram, Agda
- Compiled-down to CIC using **telescope** simplification (à la Cockx circa 2016)
- Optional typeclass instances of K/decidable equality
- **Smart** case compilation: small proof terms, avoiding UIP
- Structural, nested and well-founded recursion (i.e. more than what Function/Program can handle)
- `Derive Signature NoConfusion Subterm EqDec for I`
- Generates graph, unfolding lemma, **elimination** principles

Playtime: Regexp matching

Implement regexp matching using continuations instead of derivatives or automata.

“Proof-directed debugging”, Harper, JFP’99

Tutorial

More examples

- Hereditary substitution for Predicative System F (Mangin & Sozeau, LFMTP'15)
Nested recursion, well-founded multiset ordering on types.
- Ordinal measures (Castéran)
- Reflexive ring-like tactic on polynomials. WF subterm order on indexed polynomials
- Prototyping without verifying termination using functional eliminator

mattam82.github.io/Coq-Equations/examples

Equations Summary

- Write **just what's needed** when programming with dependently-typed structures.
- **Definitional** extension of the Coq kernel
- Gives the **right** reasoning **principles** on your (mutual, nested, dependent) function.
- Good **target** for verification of total, purely functional **Haskell** programs (e.g. hs-to-coq, UPenn).

opam install coq-equations

<http://mattam82.github.io/Coq-Equations/>

