Function to Equations and let-ins

Matthieu Sozeau
Gallinette, Inria

Coq Working Group
Paris
January 31th 3023

Motivation

Translating Function definitions to Equations:

- match translates to with
- Well-founded recursion: Acc vs. the iterating construction of Function. Equally expressive (?).
- let-bindings: not direct supported in Equations.

`let` to `where` would give a very different elimination principle (one motive per let-binding!)

⇒ Native support for `let`

OUTLINE

1. Dependent pattern-matching 101
   a. History & Examples
   b. Unification

2. Let-ins!
   a. Dependent let-ins
   b. Non-dependent let-ins?

DPM 101

"Pattern-Matching with Dependent Types", Coquand, 1992

The type-value dependency circle:

```
Inductive vector A : nat -> Type :=
| Vnil : vector A 0
| Vcons : forall (h:A) (n:nat), vector A n → vector A (S n).

Equations vtail {A n} (v : vector A (S n)) : vector A n :=
vtail (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 _.
```

?(n) is an inaccessible pattern, forced by the rest of the patterns and typing constraints.

⇒ Later adopted in Agda 2/Epigram

DPM 101 - Splitting and Matching

When checking that a definition with clauses

```
fj pj1 … pjn := …
```

covers the typing:

```
f : ∀ (x1 : I u1 … un) (x2 : τ2) … (xn : τn), τ
```

We **match**`(p1 … pn, x1 … xn)`

```
match(p1..pn, t1..tn) ⇝ Unif σ   (σ substitution)
          | Fail | Stuck x        (x variable)
```

- Standard first-match semantics
- Stuck `xi` tells which variable to split next, refining `xi` to constructors

DPM 101 - Unification

When checking that a clause

```
f (C_i p1 .. pn) q2 .. qm := …
```

matches the prototype:

```
f : ∀ (x1 : I u1 .. un) (x2 : τ2)…(xn : τn), τ
```

where `C_i p1 .. pn` has **type** `I t1 .. tn`,
we **unify:** `I t1 .. tn` and `I u1 .. un`

```
t =? u ⇝ Unif σ | Fail | Stuck
```

Usual rules of unification for constructors: no-confusion.

DPM 101 - Compilation

"Eliminating Dependent Pattern-Matching"
Goguen, McBride & McKinna, 2006

Main Idea: unification

```
t =? u ⇝ Unif σ
```

is **witnessed** by a proof of:

```
t = u → |σ|
```

where `|x:=t;σ| ≡ Σ p : x = t, |σ[p]|`

DPM 101 - Eliminating a variable

Start with: $\Gamma_0\ (x : \mathsf{I}\ \overline{t})\ \Gamma_1$

Generalize:

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

Eliminate:

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

Simplify equalities

DPM 101 - Main structure

We manipulate context maps:

$$\Gamma \vdash \sigma : \Delta$$

σ is a substitution of size $\#|\Delta|$ well-typed in Γ. We start with the identity substitution on Γ, the binders of the definition.

For example splitting a nat corresponds to producing two context maps:

$\Gamma, x : \mathsf{nat} \vdash \sigma, x : (\Delta, x : \mathsf{nat}) \Rightarrow$
$\Gamma \vdash \sigma, 0 : (\Delta, x : \mathsf{nat})$ and $\Gamma, n : \mathsf{nat} \vdash \sigma, S\ n : (\Delta, x : \mathsf{nat})$

DPM 101 - Main structure

Starting with these two context maps:

$$\Gamma \vdash \sigma, 0 : (\Delta, x : \mathrm{nat}) \text{ and } \Gamma, n : \mathrm{nat} \vdash \sigma, S\ n : (\Delta, x : \mathrm{nat})$$

We match with the user clauses to see if they have some clause of the shape:

f x1 … xn 0 := … or f x1 … xn (S _) where the xi match σi

Originally this meant that each term of σ should match a pattern in the clauses.

DPM 101 - Hidden patterns

This is impractical for nested, recursive definitions, so we introduce "hidden" patterns in context maps for passing around **variables**.

$$\Gamma, x : \mathrm{nat} \vdash \sigma, \mathrm{hide}(x) : (\Delta, x : \mathrm{nat})$$

So we actually match with the user clauses where the xi match σi, **skipping hidden patterns** in sigma.

Recursive function prototypes are passed that way in the context map. Note that due to DPM, the position of the binding for the function in Γ can change due to dependencies.

DPM 101 - Main structure

The transformation from splitting trees to terms just composes context maps.

$$\Gamma \vdash \sigma : \Delta \ \wedge \ \Delta \vdash \sigma' : \phi \Rightarrow \ \Gamma \vdash \sigma \circ \sigma' : \phi$$

A leaf of a pattern-matching will have its variables "refined" from the initial function arguments, but the return type of the function lives in the initial context, we can apply the substitution to specialize it to the branch context.

OUTLINE

1. Dependent pattern-matching 101
   a. History & Examples
   b. Unification

2. **Let-ins!**
   a. Dependent let-ins
   b. Problems, issues?

Let-ins!

- Remember let-ins in CIC are **dependent,** so they are essentially transparent: if you mention a let-bound variable it's exactly the same as mentioning the body of the let, in the theory.
- In practice we use lets for abbreviations and sharing.

```
Equations foo (x : nat) : bool :=
 foo x :=
   let y := true in
   let z := false in
   with y :=
    { | true => z
      | false => y }.
```

Let-ins!

- How to model them in context maps? We have to carry them around: $(\Gamma, x := t) \vdash \sigma, x : (\Delta, x := t)$

- However we don't want users to write clauses that "rebind" the let: so we use hide(x) instead.

- With DPM, the body of a let can get refined by pattern-matching. Just like with "destruct".

- Very different from the usual use of let-ins in Coq definitions. We have to apply the convoy pattern to them now (rebinding the let at each dependent elimination).

Problems?

This causes a problem when we compile to terms:

a let-binding can be used at different stages of refinement: duplication can happen, on the same code path and across.

Two solutions:

- Accept it: the compiled terms has the same "convertibility" behavior as the "expected" one but operationally different.
- Allow configurability: we can have non-dependent lets that behave like abstractions and are really shared.

Non-dependent lets

Abstract lets are treated like lambda-abstractions: one cannot see their body anymore, e.g. ML-like lets.

- keeps the "intuitive" operational behavior of lets
- `inspect` idiom to keep an equality with the body if needed for typing/solving obligations or later reasoning? Otherwise, lets stay abstract in the elimination principle…

Current Conclusion

Both dependent lets and non-dependent/abstract let-bindings can be supported but there is a tradeoff between expressivity/reasoning power and faithfulness to the intuitive operational semantics of let.