An Environment for Programming with Dependent Types, Take II

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

CSEC Kick-off
Santiago, Chile
March 7th 2018

OUTLINE

1. Equations Reloaded - A primer on dependent pattern-matching (a.k.a. the end of the "return with")

2. "La Belle et la Bête"[1]: Coq's Guard Condition

3. Logic to the Rescue…

4. Putting it all together: Equations + CertiCoq

[1] **Gabrielle-Suzanne Barbot de Villeneuve** (1685 – 1755)

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, avoid 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 and **elimination** principles

Dependent Pattern-Matching 101

UIP, K and Univalence

+

Demo

OUTLINE

1. Equations Reloaded
2. **Beauty & The Beast: Coq's Guard Condition**
3. Logic to the rescue..
4. Putting it all together: Equations + CertiCoq

Coq's Guard Condition

- **Goal**: ensure termination statically
- Relatively concise **syntactic** check (compared to SCT)
- Handles naturally mutual and **nested** fixpoints, e.g:

```
Inductive t : Set :=
| leaf (a : A) : t
| node (l : list t) : t.

Fixpoint size (r : t) :=
  match r with
  | leaf a ⇒ 1
  | node l ⇒ S (list_size size l)
  end.
```

- Handles `fix-match` decomposition of eliminators, hard with sized-types (A. Abel, B. Grégoire, …)

Trouble with the Guard Condition

- Guard Condition (should) ensure termination
- Slightly hard to understand syntactic criterion. Initial formal justification: Gimenez'94, gradually "sophisticated" since, **without formal proof**.
- Guard check needs to reduce definitions (!??!) (SN for call-by-name reduction **only**, WIP fix)
- Buggiest part of the system Last bug & fix: **#6649** - 24/1/18
- DPM-elimination involves equality manipulations, ...

🔥 A Recipe for Disaster 🔥

Commuting conversions, anyone?

- Inconsistency with propext (fixed in 2013):

```
Hypothesis Heq : (False -> False) = True.
Fixpoint loop (u : True) : False :=
  loop (match Heq in (_ = T) return T with
          | eq_refl => fun f : False => match f with end
          end).
```

- Typical DPM compilation:

```
Inductive Split {X : Type}{m n : nat} : vector X (m + n) → Type :=
  append : ∀ (xs : vector X m)(ys : vector X n), Split (vapp xs ys).

Equations  split_struct {X} {m n} (xs : vector X (m + n)) : Split m n xs :=
split_struct {m:=0} xs := append nil xs ;
split_struct {m:=(S m)} (cons x _ xs) ⇐ split_struct xs ⇒ {
  | append xs' ys' := append (cons x xs') ys' }.
```

🙁 **Not** structural on vectors, due to uses of J, structural on **index**, which hence **matters**...

Still, we can handle mutual & nested rec!

http://mattam82.github.io/Coq-Equations/examples/nested_mut_rec.html

Functional elimination is good for you!

OUTLINE

1. Equations Reloaded
2. Beauty & The Beast: Coq's Guard Condition
3. **Logic to the Rescue…**
4. Putting it all together: Equations + CertiCoq

Logic to the Rescue: Acc is not a Hack

structurally recursive

$\subset$

well-founded on subterm relation

1) `Derive Subterm for I` relation on (computational/hType) inductive families
2) Prove well-foundedness by structural rec
3) Profit! "`by rec I_subterm x`"

- Define split on vectors by rec on the vector **or the index**!
- **Extracts** to general fixpoints

The Beauty of Logic

```
Equations elements' (r : t) : list A :=
elements' l by rec r (MR lt size) :=
elements' (leaf a) := [a];
elements' (node l) := fn l hidebody
  where fn (x : list t) (H : list_size size x < size (node l)) : list A :=
  fn x H by rec x (MR lt (list_size size)) :=
  fn nil _ := nil;
  fn (cons x xs) _ := elements' x ++ fn xs hidebody.
```

- Use the weapon of your choice
- Equations generates unfolding lemma
- Eliminator abstracts away from the w.f. relation: do the work only once.

Computational content

- Closed calls still reduce to the same normal forms: `I_subterm` is **closed**
- Make it **fast** by adding 2^n `Acc_intro`'s to the well-foundedness proof.
- For calls on **open** terms:
  - **Proofs**: unfolding lemma or derived equalities (more control)
  - **Programs:** still reduces, unfolding might be unwieldy though.
- **Functional extensionality** is used to prove the unfolding lemma (easier to automate)

Playtime: Regexp matching

- Implement regexp matching using continuations instead of derivatives or automata (Harper'99 - "Proof-directed debugging")
- Needs dependent types, well-founded recursion, and eliminator for recursive calls "under binders"...

Demo

```
type 'alpha regexp =
| Empty
| Epsilon
| Atom of 'alpha
| Disj of bool * bool * 'alpha regexp * 'alpha regexp
| Conj of bool * bool * 'alpha regexp * 'alpha regexp
| Seq of bool * bool * 'alpha regexp * 'alpha regexp
| Star of 'alpha regexp

type 'alpha substring = 'alpha list

type 'alpha cont_type = 'alpha substring -> bool

(** val matches :
    'a1 alphabet -> bool -> 'a1 regexp -> 'a1 list -> 'a1 cont_type -> bool **)

let matches alpha null r s k =
  let hypspack = { pr1 = null; pr2 = { pr1 = r; pr2 = { pr1 = s; pr2 =
    { pr1 = k; pr2 = Tt } } } }
  in
  let rec fix_F x =
    let h = x.pr2 in
    let r0 = h.pr1 in
    let h0 = h.pr2 in
    let s0 = h0.pr1 in
    let h1 = h0.pr2 in
    let k0 = h1.pr1 in
    let matches0 = fun null0 r1 s1 k1 ->
      let y = { pr1 = null0; pr2 = { pr1 = r1; pr2 = { pr1 = s1; pr2 =
        { pr1 = k1; pr2 = Tt } } } }
      in
      (fun _ -> fix_F y)
    in
    (match r0 with
     | Empty -> False
     | Epsilon -> k0 s0
     | Atom l ->
       (match s0 with
        | Nil -> False
        | Cons (a, l0) ->
          (match equiv_dec (alphabet_dec alpha) l a with
           | Left -> k0 l0
           | Right -> False))
     | Disj (l, r1, r2, r3) ->
       (match matches0 l r2 s0 k0 __ with
        | True -> True
        | False -> matches0 r1 r3 s0 k0 __)
     | Conj (l, r1, r2, r3) ->
       matches0 l r2 s0 (fun s' ->
         matches0 r1 r3 s0 (fun s'' ->
           match equiv_dec (list_eqdec (alphabet_dec alpha)) s' s'' with
           | Left -> k0 s'
           | Right -> False) __) __
     | Seq (l, r1, r2, r3) ->
       let k1 = fun s' -> matches0 r1 r3 s' k0 __ in matches0 l r2 s0 k1 __
     | Star r1 ->
       let match_star = fun s' -> matches0 True (Star r1) s' k0 __ in
       (match k0 s0 with
        | True -> True
        | False -> matches0 False r1 s0 match_star __))
  in fix_F hypspack
```

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

OUTLINE

1. Equations Reloaded
2. Beauty & The Beast: Coq's Guard Condition
3. Logic to the Rescue...
4. **Putting it all together: Equations + CertiCoq**

Equations + CertiCoq

Certified Compilation from high-level dependent pattern-matching definitions to assembly.

Goal: do better than extraction.

Currently:

- Erases proofs (stuff in Prop)
- Erases types (abstractions, parameters)
- Does a little bit of optimization of representation optimization, e.g. unbox
  Inductive bigint :=
  | bignat (i : int63)
  | bigbig (i : BigInt.t)

Indices do not matter

But does not erase **indices!**

- Inductive fin : nat -> Type :=
  | fz (n : nat)
  | fs (n : nat) (f : fin n).
- None of the functions on fin use the index, it is just used for typing / justifying recursion arguments.
- Ideally should extract to…

  Inductive fin : Type :=
  | fz
  | fs (f : fin).

Indices do not matter

Dependent-types ensure our programs **never** go wrong, and do the right thing, **statically**. We want to get rid of the dependencies to get the (safe) code to run at full speed.

```
  head x = match x with
           | nil ⇒ assert false
           | cons x _ ⇒ x
           end
                        ⇒
function head (x : list) { return (*x).hd; }
```

Work by Eric Tanter, Pierre-Évariste Dagand and Nicolas Tabareau in this direction.

Conclusion

- Write **just what's needed** when programming with dependently-typed structures.
- Gives the **right** reasoning **principles** on your (mutual, nested, dependent) function.
- CertiCoq compiles it **maintaining** the certification **assurance**.
- Future: **run faster** that simply-typed program + correctness proof.
- Good **target** for verification of total, purely functional **Haskell** programs (e.g. hs-to-coq).

mattam82.github.io/Coq-Equations

opam install coq-equations

informatics mathematics