The Optimal Fixed Point Combinator

Arthur Charguéraud

INRIA – Gallium

Gallium Team Seminar Rocquencourt, 2010/02/22

Recursive definitions in the logic

1) To use Coq as a programming language

→ used to reflect the "let-rec" construct

```
Fixpoint length l := match l with
  | nil => 0
  | a::l' => 1 + length l'
```

2) To use Coq as a specification / proof language

a) To define recursive predicates (sometimes more appropriate than a corresponding inductive definition)

```
Fixpoint In x l := match l with
  | nil => False
  | a::l' => x = a \/ In x l'
```

b) To define co-inductive values, functions, predicates (e.g. traces of diverging programs, recursive types)

```
CoFixpoint seq n : stream nat := n:::(seq (n+1))
```

Restrictions on recursive definitions

1) Cannot allow arbitrary recursive functions

```
Fixpoint f x := 1 + f x.
```

$\rightarrow$ Accepting this definition would be unsound:
`f x ≡ 1 + f x` implies `f x = 1 + f x` implies `0 = 1`

2) Cannot allow arbitrary co-recursive functions

```
CoFixpoint f x := f x.
```

$\rightarrow$ If accepted, the function `f` would basically be a proof term for any co-inductive proposition

What are the restrictions implemented in Coq?

Restrictions on recursive functions

Recursive calls allowed on strict subterms only

$\rightarrow$ This is a very basic syntactic check. The argument of a recursive call must be a pattern variable bound from a case analysis on the decreasing argument.

```
Fixpoint length l := match l with
| nil => 0
| a::t => 1 + length t                       (* accepted *)

Fixpoint length l := match l with
| nil => 0
| _ => 1 + length (tail l)                   (* rejected *)

Fixpoint sorted l := match l with
| nil => True
| a::nil => True
| a::b::t => (a <= b) /\ sorted (b::t) (* rejected *)
```

Restrictions on co-recursive func.

Co-recursive calls to be guarded by constructors

$\rightarrow$ Again, a very basic syntactic check. The argument of a corecursive call must be guarded by constructors and only by constructors. Examples:

```
CoFixpoint s := 1 ::: s.

CoFixpoint seq n := n ::: seq(n+1).

CoFixpoint map f s :=
  let '(x:::t) := s in
  f x ::: map f t.


CoFixpoint s := 0 ::: map succ s. (* rejected *)

CoFixpoint f n :=
   if is_prime n then n ::: f(n+1)
                 else f (n+1).    (* rejected *)
```

Existing techniques

For recursive definitions:

1) Recursion on the structure of proofs objects

2) Same but using subset types (Sozeau)

3) The domain-predicate approach (Dubois & Donzeau-Gouge, Bove & Capretta, Krauss)

4) Contraction conditions (Matthews & Krstić)

For co-recursive definitions:

1) Transformations to make some co-recursive definitions fit the guard condition (Bertot et al.)

2) A technique based on another form of contraction conditions (Matthews)

The subset-types approach

```
let rec f x  =  if x = 0 then 0 else f (f (x - 1))

Program Fixpoint f (x:nat) {measure id x}
                                   : { y : nat | y = 0 } :=
  match x with | O => O
               | S x' => f (f x')  end.

f  :  nat -> {y : nat | y = 0}

Next Obligation.
  x : nat
  f : {x' : nat | id x' < id x} -> {y : nat | y = 0}
  x' : nat
  Heq_x : S x' = x
  ----------------------------------------------------
  `f (exist (fun x'0 => x'0 < x) x' (f_obligation_2 f Heq_x))) < x
```

invoke an appropriate tactic here.

`|- forall z,  z = 0  ->  z < x.`   *omega proves it.*

The inductive graph approach

→ In Isabelle's *Function* package, by A.Krauss

Recursive equation:

$Z\ n = (\text{if } n = 0 \text{ then } 0 \text{ else } Z\ (Z\ (n-1)))$ (user input)

Extracted calls:

$[n \neq 0 \rightsquigarrow n - 1], [n \neq 0 \rightsquigarrow Z\ (n-1)]$

Graph:

$$\frac{n \neq 0 \Longrightarrow (n-1, h\ (n-1)) \in G_Z \qquad n \neq 0 \Longrightarrow (h\ (n-1), h\ (h\ (n-1))) \in G_Z}{(n, \text{if } n = 0 \text{ then } 0 \text{ else } h\ (h\ (n-1))) \in G_Z}$$

Function: **Definition Z := (fun x => εy. (x,y) ∈ G_Z)**

Domain:

$$\frac{n \neq 0 \Longrightarrow n - 1 \in dom_Z \qquad n \neq 0 \Longrightarrow Z\ (n-1) \in dom_Z}{n \in dom_Z}$$

Simplification and induction rules:

$n \in dom_Z \Longrightarrow Z\ n = (\text{if } n = 0 \text{ then } 0 \text{ else } Z\ (Z\ (n-1)))$

$$\frac{\bigwedge n.\ n \in dom_Z \Longrightarrow (n \neq 0 \Longrightarrow P\ (n-1)) \Longrightarrow (n \neq 0 \Longrightarrow P\ (Z\ (n-1))) \Longrightarrow P\ n}{a \in dom_Z \Longrightarrow P\ a}$$

Pros/cons of the two approaches

Subset-types approach:

+ applies to a large class of recursive functions
+ constructive thus compatibile with extraction
– the function now admits a dependent type
– full specification may need to appear in the type
– proof of termination to be given at definition time

Inductive graph approach:

+ applies to a large class of recursive functions
+ clear separation between definitions and spec/proofs
+ good automation for proofs of termination
– can only process a set of top-level equations
– heavy implementation (lot of work to do it in Coq)

Motivation

I investigated the third main approach to recursive definitions: "contraction conditions"

Goal:

$\rightarrow$ support for a very large class of circular definitions

$\rightarrow$ ability to deal with co-recursive definitions

Contraints:

$\rightarrow$ no modification of the type of the functions defined

$\rightarrow$ complete separation between definitions and proofs

Towards a generic combinator

We want to define generic combinator Fix that can be applied to any functional.

```
Definition Log log x :=
  if x <= 1 then 0 else 1 + log (x/2).

Definition log := Fix Log.

Lemma log_fix : forall x, log x = Log log x.
```

The above fixed point equation can then be used to unfold the body of the recursive function at any time.

```
log x <= x              rewrite log_fix.

Log log x <= x          unfold Log.

(if x <= 1 then 0 else 1 + log(x/2)) <= x
```

How to define `Fix`? How to prove `log_fix`?

Contraction conditions

– used to prove fixed point in Banach spaces

$$|| F(x) - F(y) || < \alpha \cdot || x - y || \quad \text{with } \alpha < 1$$

– used by Paulson (1992) to implement the theory of inductive definitions in Isabelle

– used by Matthews (1999) to formalize non-guarded co-recursive definitions

– used by Matthews & Krstić (2003) to formalize recursive functions with nested calls, like f(f(x))

One of our contribution is to generalize and unify the various forms of contraction conditions

→ First, let me introduce contraction conditions by explaining how I rediscovered this notion.

Bounded recursion

Bounded recursion: bounds the number of calls

```
let rec log x =
  if x <= 1 then 0 else 1 + log (x/2).

Fixpoint log' n x :=
  match n with
  | O => arbitrary
  | S n' => if x <= 1 then 0 else 1 + log'(n-1)(x/2).

Definition log x := log' x x.
```

The definition of `log` relies on the fact that `x` steps are sufficient to compute the logarithm of `x`, in the sense that the value `arbitrary` will never be returned.

Problem: the auxiliary variable `n` shows up when we unfold the definition of the function `log`.

A combinator based on measures

The idea: hide the bound inside a combinator

```
Fixpoint Fixn_run F n x :=
  match n with
  | O => arbitrary
  | S n' => F (Fixn_run F n') x

Definition Fixn F mu x := Fixn_run F (1+mu x) x.

Definition Log log x :=
  if x <= 1 then 0 else 1 + log (x/2).

Definition log := Fixn Log (fun x => x).
```

→ How to prove the fixed point equation?

```
Lemma log_fix : forall x, log x = Log log x.
```

Contraction condition for Fixn

Use this theorem to derive fixed point equations:

```
Lemma Fix_eq : forall f F mu,
  f = Fixn F mu ->
  (forall f1 f2 x,
     (forall y, mu y < mu x -> f1 y = f2 y) ->
     F f1 x = F f2 x) ->
  (forall x, f x = F f x).
```

Let's apply it to the functional Log.

```
Hypothesis: forall y, mu y < mu x -> f1 y = f2 y

Goal: Log f1 x = Log f2 x

Goal:  (if x <= 1 then 0 else 1 + f1(x/2))
     = (if x <= 1 then 0 else 1 + f2(x/2))

Subgoal: x <= 1  |-  0 = 0
Subgoal: x > 1   |-  1 + f1(x/2) = 1 + f2(x/2)

Apply the hypothesis to y = x/2, and check (x/2) < x
```

Key idea

The contraction condition captures the fact that recursive calls are made on smaller arguments.

```
forall f1 f2 x,
  (forall y, mu y < mu x -> f1 y = f2 y) ->
  F f1 x = F f2 x
```

Generalization to well-founded rec.

Replace the recursion on the structure of the bound n with a recursion on a proof of well-foundedness of the termination relation.

```
Definition Fixwf (F:(A->B)->(A->B)) (R:A->A->bool)
                 (W:well_founded R) (x:A) :=
  Acc_rect _ (fun x _ f =>
     let f' y := match sumbool_of_bool (R y x) with
                 | left H => f y H
                 | _ => arbitrary
                end in
     F f’ x) (W x).
```

Example:

```
Definition log := Fixwf Log lt lt_wf.
```

→ Cool, it's entirely constructive! (but who cares?)

→ Remark: the evaluation of R may be quite inefficient

More on contraction conditions

How can contraction conditions support:

1) Partial recursive functions, where the function may diverge on arguments outside the domain

2) Nested recursion, like

```
let rec f x = ... f(f(y)) ...
```

3) Higher-order recursion, like

```
let rec f x =  ... map f ys ...
```

Fixed point: partial functions

It suffices to restrict the values to a domain D:

```
Lemma Fix_eq' : forall f F R (W:well_founded R) D,
  f = Fixwf F R W ->
  (forall f1 f2 x, D x ->
     (forall y, D y -> y < x -> f1 y = f2 y) ->
     F f1 x = F f2 x) ->
  (forall x, D x -> f x = F f x).
```

→ All recursive calls must be made to values in **D**.

→ The guarded fixed point equation allows to unfold the definition of the fixed point **f** whenever the function is applied to an argument that belongs to **D**.

Fixed point: nested recursion

The basic contraction condition does not suffice.

```
Definition F f x =
  if x = 0 then 0 else f(f(x-1)).

Lemma f_fix : forall x, f x = F f x.
  apply the fixed point theorem.

Hypothesis: forall y, y < x -> f1 y = f2 y

Goal: F f1 x = F f2 x

Goal:  (if x = 0 then 0 else f1(f1(x-1))
     = (if x = 0 then 0 else f2(f2(x-1))

Subgoal: x > 0  |-  f1(f1(x-1)) = f2(f2(x-1))

The hypothesis with y = x-1 gives f1(x-1) = f2(x-1).

But there is no way to prove f1 y = f2 y for
y = f1(x-1), because we don't know that f1(x-1) < x.
```

Fixed point: nested recursion

The solution is to include an invariant.

```
Lemma Fix_eq : forall f F R (W:well_founded R) Q,
  f = Fixwf F R W ->
  (forall f1 f2 x,
    (forall y, y < x -> f1 y = f2 y /\ Q y (f1 y)) ->
    F f1 x = F f2 x /\ Q x (f1 x)) ->
  (forall x, f x = F f x /\ Q x (f x)).
```

```
Invariant: Definition Q x r := (r = 0).

Hypothesis: forall y < x, f1 y = f2 y /\ Q y (f1 y)

Goal: F f1 x = F f2 x /\ Q x (f x)

Goal:  (if x = 0 then 0 else f1(f1(x-1))
     = (if x = 0 then 0 else f2(f2(x-1))
    /\ (if x = 0 then 0 else f1(f1(x-1)) = 0

Taking y = x-1, we derive f1(x-1) = f2(x-1) = 0
Taking y = 0, we derive f1(f1(x-1)) = f2(f1(x-1)) = 0
```

Fixed point: higher-order recursion

Higher-order recursion is supported right away

```
type tree = Leaf of nat | Node of list tree

Definition Succ_tree succ_tree x := match x with
  | Leaf n => Leaf (n+1)
  | Node xs => Node (List.map succ_tree xs)

Definition on the well-founded termination relation:
  Inductive (<) : tree -> tree -> Prop :=
    | subtree : forall y xs, In y xs -> y < (Node xs).

Hypothesis: forall y < x, f1 y = f2 y

Goal: Succ_tree f1 x = Succ_tree f2 x

Subgoal: Leaf (n+1) = Leaf (n+1)
Subgoal: Node (List.map f1 xs) = Node (List.map f2 xs)

Exploit this "congruence rule": forall f1 f2 l,
  (forall a, In a l -> f1 a = f2 a) ->
  List.map f1 l = List.map f2 l
```

Contraction conditions: summary

We have introduced two combinators

– `Fixn F mu` is the fixed point of `F` for a measure `mu`

– `Fixwf F R W` produces the fixed point of `F` given a decidable relation `R` and a proof of well-foundedness `W`

We used contraction conditions to prove that the fixed point equations holds on given domains

– The reasoning about termination is here carried out completely inside the logic, without any external tool

– This approach allows for the formalization of a very large class of recursive functions

Compared with previous work

– The combinators are defined constructively

– Slight improvement in the case of nested recursion

Why we want to go further

We want to write `f = Fix F`, without providing the well-founded relation in the definition of `f` (`Fixn F mu` and `Fixwf F R W` is not good enough)

Why is Fixwf not quite satisfying?

– Settling on a relation `R` at the time of definition means that one must have in mind the domain of `f` and its termination proof when definining `f`.

– Proving `R` to be decidable can be very tedious.

– Proving `R` to be well-founded before definining `f` does not allow to separate specifications from proofs.

– `Fixwf` is helpless for building cofixpoints.

→ Yet, `Fixwf F R W` for a decidable relation `R` seems to be the best we can hope for in a constructive world.

A combinator for total functions

One can define a satisfying combinator for total recursive functions, using Hilbert's epsilon.

```
Definition Fix_total F :=
  εg. (forall x, g x = F g x).
```

(exploited, e.g., by Matthews)

1) Define the fixed point: `Definition f := Fix_total F.`

2) Prove that **F** satisfies the contraction condition with respect to some well-founded relation **R** of our choice.

3) As seen earlier, it follows that **Fixwf F R W** satisfies the fixed point equation for the functional **F**.

4) Since there exists at least one function **g** such that **forall x, g x = F g x**, we can deduce that **Fix_total F** also satisfies this fixed point equation. Hence,

```
forall x, f x = F f x
```

Two solutions for partial functions

The previous approch does not work immediately apply to partial recursive functions.

First solution: make the function total by testing if the argument is inside the domain explicitly.

```
Definition Div div (x,y) :=
  if y = 0 then arbitrary else
  if x < y then 0 else 1 + div (x - y, y)
```

→ Not satisfying: the code of the function is altered

Second solution: change the fixed point combinator

```
Definition Fix_partial F D :=
   εg. (forall x, D x -> g x = F g x).
```

→ The domain need to be known at the time of definition, which is not always easy and practical

Try harder

`Fix F D` is not good enough!

We want to write `Fix F`, and nothing else...

→ The key difficulty is to find a way to define the domain **`D`** of the fixed point of **`F`** in terms of **`F`**.

→ Intuitively, we want to pick the largest domain **`D`** such that **`F`** admits a unique fixed point on **`D`**.
But does such a largest domain always exists?

→ In the following, we rely on a powerful theorem to address this question and to give a definition for **`Fix`**.

Maximal inductive fixed points

Theorem [Matthews & Krstić, 2003]: (& Gonthier?)

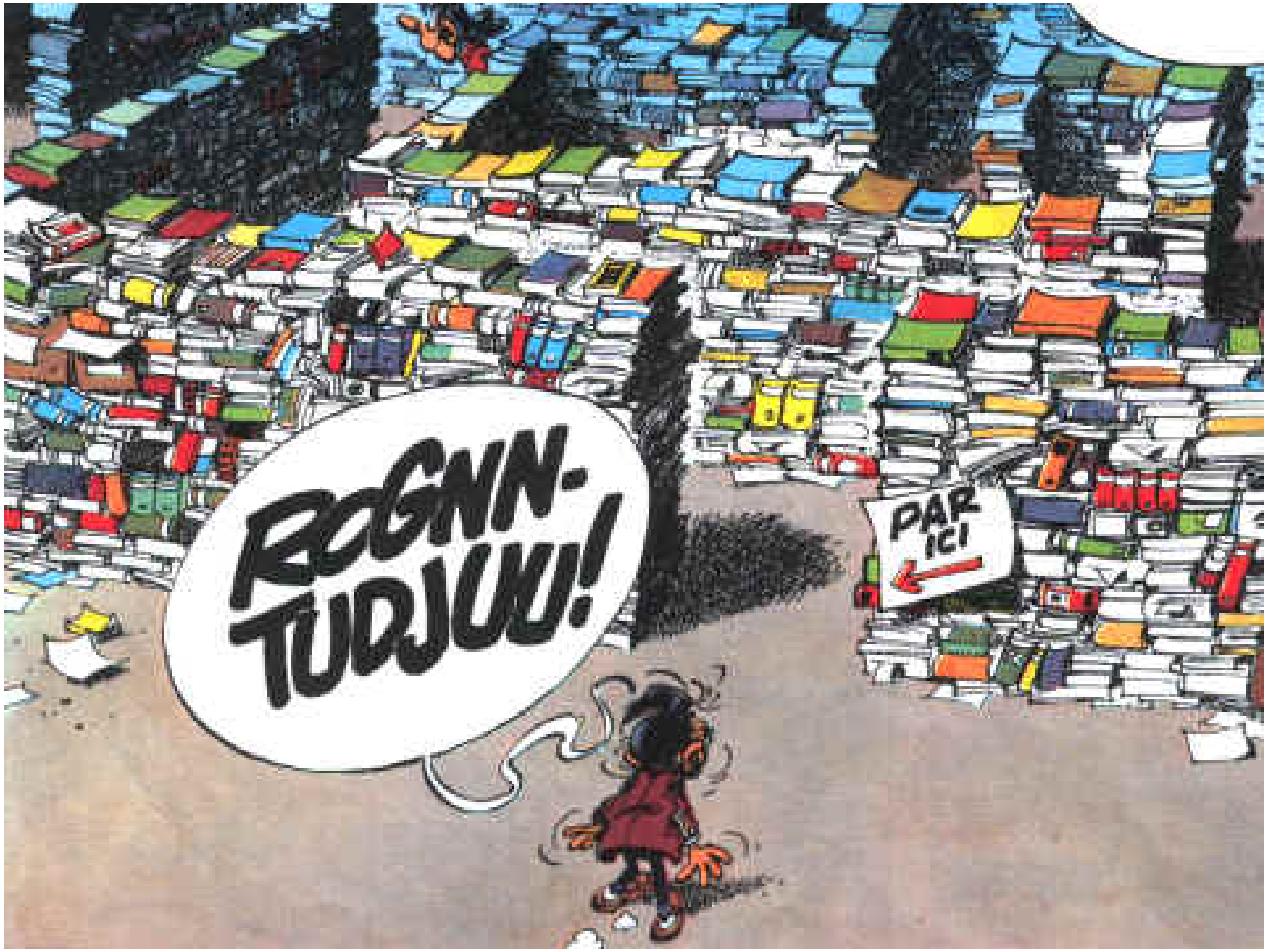
(Slightly simplified statement) For any functional **F**, there exists a largest domain **D** such that we can find a well-founded relation **R** for which **F** satisfies the contraction condition with respect to **R** on the domain **D**

→ Intuitively, it is the largest domain on which **F** can be proved to terminate.

→ This theorem could be exploited to define the domain associated with the functional **F**.

→ In fact, there exists a much older and much more general theorem defining the domain of a functional.

Good old papers can get lost...



(especially in Gaston's library!)

Theory of optimal fixed points

Theorem [Manna & Shamir*, 1975]:

Any functional **F** admits an *optimal* fixed point

Definition: a function **f** is the optimal fixed point of **F** if it is the *generally-consistent* fixed point of **F** with the largest domain.

Definition: a fixed point **f** of **F** is generally-consistent if it is consistent with any other fixed point **f'** of **F**.

Note: two partial functions **f** and **f'** are consistent if they agree on the intersection of their domains, i.e.

$\forall$**x** $\in$ **(dom f** $\cap$ **dom f'), f x = f' x**

(*) Adi Shamir is the "S" from the "RSA" protocol;
He developed "optimal fixed points" during his thesis.

Some more intuition

Properties of generally-consisted fixed points

→ The domain of a generally-consistent fixed point includes only points whose image is uniquely defined

i.e. if **f1** and **f2** are two fixed points of **F** such that **f1 x** ≠ **f2 x** for some **x**, then **x** does not belong to the domain of any generally-consistent fixed point

→ We thus exclude ambiguous points from the domain

Properties of the optimal fixed point

→ Any generally-consistent fixed point is a restriction of the optimal fixed point to a smaller domain

→ The optimal fixed point captures the maximal amount of non-ambiguous information contained in **F**

Interest of optimal fixed points

The big picture of this work:

1) We define `Fix` as a combinator that picks an optimal fixed point, using Hilbert's epsilon operator.

2) Given a functional `F`, we build `f := Fix F`.

3) We later prove `F` to satisfy the contraction condition for some well-founded relation `R` on some domain `D`.

4) We deduce that `F` admits a generally-consistent fixed point on the domain `D`.

5) Because `f` is the generally-consistent fixed point of `F` with the largest domain, the domain of `f` contains `D`.

6) Thus, `f` satisfies the fixed point equation on `D`, i.e.

`forall x, D x → f x = F f x.`

The optimal fixed point combinator

Definition of the combinator:

```
Definition Fix A B `{Inhabited B} (F:(A->B)->(A->B)) :=
  εf. (optimal_fixed_point_of F f).
```

(the typeclass `` `{Inhabited B} `` is needed for soundness)

Specification of the combinator:

```
Lemma Fix_spec : forall A B `{Inhabited B} F f R D,
  f = Fix F ->
  well_founded R ->
  (forall f1 f2 x, D x ->
    (forall y, D y -> R y x -> f1 y = f2 y) ->
     F f1 x = F f2 x) ->
  (forall x, D x -> f x = F f x).
```

(can be extended with invariants, for nested recursion)

That's it!

Example with the log function

```
Definition Log log x :=
  if x <= 1 then 0 else 1 + log (x/2).

Definition log := Fix Log.

Lemma log_fix : forall x, log x = Log log x.
Proof.
   applys~ (Fix_spec lt). introv H. unfolds.
   case_if~. fequals. apply H. apply~ div2_lt.
Qed.
```

– `lt` is the relation used to argue for termination,
– `H` is the hypothesis from the contraction condition,
– `div2_lt` is the lemma used to prove `x/2 < x`.
–The symbol "`~`" stands for a call to automation.

What I haved formalized

1) Formalization of partial functions in Coq

→ represent them as pairs of type **`(A→Prop) * (A->B)`**

2) Formal definition of the notion of optimal fixed point

→ definition of order on partial functions, consistency

3) Proof that the optimal fixed point always exists

→ formalize entirely the proof of Manna and Shamir

4) Proof that contraction conditions imply the existence of a generally-consistent fixed point

→ adapted from the proof that maximal inductive fixed points are generally-consistent (Krstić, 2004).

Summary

The combinator Fix:

– can be used to define recursive functions,

– even partial functions without giving their domain nor their termination relation at time of definition,

– it supports nested recursion, higher-order recursion,

– n-ary recursive functions and mutually-recursive functions can be defined easily through encodings with pairs and sums, respectively.

Next: corecursive values and functions

C.o.f.e.'s

Developed by Matthews (1999), later polished by Di Gianantonio and Miculan (2003):

Complete Ordered Families of Equivalences

Example with the stream 0:::1:::2:::3:::4:::...

```
Definition F x := 0 ::: map succ x.
Definition x := FixVal (≈) F.
Lemma x_fix : forall x, x ≈ F x.
```

→ `s ≈ s'` means that `s` and `s'` are bisimilar streams

→ `FixVal (≈) F` picks a value `x` such that `x ≈ F x` whenever there exists a unique such value

Contraction condition for streams

The fixed point equation `x ≈ F x` is derived from the following contraction condition:

forall x1 x2 i, x1 $\approx_i$ x2 → F x1 $\approx_{i+1}$ F x2

where `s` $\approx_i$ `s'` iff `s` and `s'` agree up to their i-th item.

Let's prove the contraction condition for our functional.

```
Definition F x := 0 ::: map succ x.
```

Hypothesis: x1 $\approx_i$ x2

Goal: F x1 $\approx_{i+1}$ F x2
Goal: 0:::map succ x1 $\approx_{i+1}$ 0:::map succ x2
Goal: map succ x1 $\approx_i$ map succ x2

Conclude using the following properties of map:

x1 $\approx_i$ x2 -> map succ x1 $\approx_i$ map succ x2

(i.e. "map succ" preserves "similarity up to depth i")

Key idea

The contraction condition captures the fact that the co-recursive definition is productive.

$$\texttt{forall x1 x2 i, x1} \approx_{\texttt{i}} \texttt{x2} \rightarrow \texttt{F x1} \approx_{\texttt{i+1}} \texttt{F x2}$$

Counter-example

The next definition does not specify a stream:

```
Definition F x := 0 ::: tail x.   (* accepted *)
Definition x := FixVal (≈) F.     (* accepted *)
```

Let's see what the contraction condition would give.

Hypothesis: x1 $\approx_i$ x2

Goal: F x1 $\approx_{i+1}$ F x2

Goal: 0:::tail x1 $\approx_{i+1}$ 0:::tail x2

Goal: tail x1 $\approx_i$ tail x2

Here we are stuck, because all we can prove is that:

x1 $\approx_{i+1}$ x2 -> tail x1 $\approx_i$ tail x2

but our hypothesis x1 $\approx_i$ x2 is weaker than x1 $\approx_{i+1}$ x2

```
Lemma x_fix : forall x, x ≈ F x. (* cannot prove it *)
```

General presentation of c.o.f.e.'s

More generally, the contraction condition:

```
forall i x1 x2,
 (forall j < i, x1 ≈j x2) ->
  F x1 ≈i F x2
```

imples the existence of a unique fixed point x **modulo** (≈) **(i.e. such that** x ≈ F x **) when:**

– F has type $A \rightarrow B$

– I is a type, ordered with < (transitive well-founded)

– $(\approx_i)_{i:I}$ is a family of equivalence relations

– ≈ is the intersection of all the relations $\approx_i$

– any *coherent* sequence $(u_i)_{i:I}$ of values of type A admit a *limit* l in the sense that forall i, $u_i \approx_i$ l.

Completeness in c.o.f.e.'s

For streams, completeness asserts the existence of a limit to any coherent sequence of streams

u_1 = a ::: _ ::: _ ::: _ ::: _ ::: _ ::: ...
u_2 = a ::: b ::: _ ::: _ ::: _ ::: _ ::: ...
u_3 = a ::: b ::: c ::: _ ::: _ ::: _ ::: ...
u_4 = a ::: b ::: c ::: d ::: _ ::: _ ::: ...
u_5 = a ::: b ::: c ::: d ::: e ::: _ ::: ...
... ...
u_∞ = a ::: b ::: c ::: d ::: e ::: f ::: ...

The limit can be constructed by diagonalization

forall i, $u_i \approx_i u_\infty$

For example, u_4 is similar to the limit $\mathbf{u}_\infty$ up to length 4.

Adding invariants

The previous contraction condition is not quite strong enough to capture advanced co-recursive definitions.

→ Example: Hamming's sequence (point out by Knuth)

```
Definition F x :=
   1 ::: merge (map (mult 2) x) (map (mult 3) x).
```

where `merge` merges two sorted streams.

→ Simpler example for the sake of presentation

```
Definition F x := 2 ::: filter (≥ 1) x.
```

→ Remark: the following does not define a stream

```
Definition F x := 0 ::: filter (≥ 1) x.
```

Adding invariants

→ The generalized strengthen contraction condition:

$\forall$ **x1 x2 i, (x1 $\approx_i$ x2 $\wedge$ Q i x1 $\wedge$ Q i x2) $\rightarrow$
F x1 $\approx_{i+1}$ F x2 $\wedge$ Q i (F x1)**

Let's prove it holds for our functional.

Definition F x := 2 ::: filter (≥ 1) x.

Invariant: Q i x := (∀j ≤ i, nth j x ≥ 1)

Hypothesis: x1 $\approx_i$ x2 ∧ Q i x1 ∧ Q i x2

Goal 1: F x1 $\approx_{i+1}$ F x2
Goal 1: 2 ::: filter (≥ 1) x1 $\approx_{i+1}$ 2 ::: filter (≥ 1) x1
Goal 1: filter (≥ 1) x1 $\approx_i$ filter (≥ 1) x2

Goal 2: Q i (F x1)
Goal 2: ∀j ≤ i+1, nth j (2::x1) ≥ 1

Key idea about invariants

– Reasoning about a recursive function with nested calls requires the ability to specify results of the function (Q x (f x)).

– Reasoning on a non-trivial co-recursive value requires the ability to specify arbitrarily-long prefixes of this value (Q i x).

Contraction conditions for functions

Example of a co-recursive function:

```
Definition F f x := x ::: f (x + 1).
Definition f := FixFun (≈) F.
Lemma f_fix : forall x, f x ≈ F f x.
```

(**FixFun (≈)** is like **Fix**, but it takes **(≈)** as argument)

Contraction condition for corecursive functions:

$$\texttt{forall f1 f2 x i,}\quad \texttt{(forall y, f1 y} \approx_i \texttt{f2 y) ->}\quad \texttt{F f1 x} \approx_{i+1} \texttt{F f2 x}$$

– **s ≈ s'** means that **s** and **s'** are bisimilar streams.
– **s** $\approx_i$ **s'** iff **s** and **s'** agree up to their i-th element.

Contraction conditions for functions

One can prove the fixed point equation for f.

```
Definition F f x := x ::: f (x + 1).
```

forall f1 f2 x i,
(forall y, f1 y $\approx_i$ f2 y) ->
F f1 x $\approx_{i+1}$ F f2 x

Goal: F f1 x $\approx_{i+1}$ F f2 x

Goal: x ::: f1(x+1) $\approx_{i+1}$ x ::: f2(x+1)

Goal: f1(x+1) $\approx_i$ f2(x+1)

Conclude using the hypothesis with y = x+1

→ The fixed point equation holds modulo bisimilarity:

Lemma f_fix : forall x, f x ≈ F f x.

Partial co-recursion: stream filter

```
let rec filter x =     // where P is a given predicate
  let a:::y = x in
  if P a then a ::: filter y
  else filter y
```

Matthews could only deal with total functions:

```
Definition Filter filter x :=
  if (never P x) then arbitrary else
  let '(a:::y) := x in
  if P a then a ::: filter y else filter y.
```

With the optimal fixed point combinator Fix, we have:

```
Definition Filter filter x :=
  let '(a:::y) := x in
  if P a then a ::: filter y else filter y.

Definition filter := FixFun (≈) Filter.
```

Partial co-recursion: stream filter

The domain of the filter function is made of streams which contain infinitely many values satisfying **P**.

```
Definition Filter filter x :=
  let '(a:::y) := s in
  if P a then a ::: filter y else filter y.
```

The filter function does not produce a head value at each call: a bounded number of recursive calls may be required before the head value is exhibited.

$\rightarrow$ Generalize contraction condition to be used:

```
forall f1 f2 x i, D x ->
  (forall y j, (j,y)<(i,x) -> D y -> f1 y ≈j f2 y) ->
  F f1 x ≈i F f2 x
```

where **(j,y)<(i,x)** is a lexicographical comparison, and **y < x** holds if the next element satisfying P is closer in the stream **y** than in stream **x**.

Fixed point equation for filter

```
Definition Filter filter x :=
  let '(a:::y) := s in
  if P a then a ::: filter y else filter y.
```

Proof of the contraction condition:

Hypotheses:

- D x
- forall y j, (j,y)<(i,x) -> D y -> f1 y $\approx_j$ f2 y

Goal: F f1 s $\approx_i$ F f2 s

Goal: (if P a then a ::: f1 y else f1 y)
$\approx_i$(if P a then a ::: f2 y else f2 y)

Sugoal if (P a): a ::: f1 y $\approx_i$ a ::: f2 y
follows from f1 y $\approx_{i-1}$ f2 y taking j = i-1

Sugoal if (~ P a): f1 y $\approx_i$ f2 y
follows from the hypothesis taking j = i
and checking that y < x (next "good" item is closer).

Unifying contraction conditions

If the following hypotheses hold

- **F** is a functional of type **A->A** (where **A** is inhabited)
- **(A,I,<,$\approx_i$)** is a c.o.f.e.
- **Q** is a *continuous* property of type **I->A->Prop**
- The following contraction condition holds

$$\forall\ i\ x1\ x2,\ (\forall j < i,\ x1 \approx_j x2 \wedge Q\ j\ x1 \wedge Q\ j\ x2) \rightarrow F\ x1 \approx_i F\ x2 \wedge Q\ i\ (F\ x1)$$

Then we can deduce that

- **F** admits a unique fixed point **x** modulo $\approx$
- Moreover **x** satisfies the invariant, i.e. $\forall i,\ Q\ i\ x$

Unifying the combinators

So far we have used three combinators:

– **FixVal (≈) F** picks the unique fixed point **x** modulo **≈**

$\forall y,\ y \approx x \rightarrow y \approx F\ y$

– **FixFun (≈) F** picks the optimal fixed point **f** modulo **≈**

$\forall x,\ f\ x \approx F\ f\ x$

– **Fix F** picks the optimal fixed point for recursive functions. It is defined as **FixFun (=) F**.

$\forall x,\ f\ x = F\ f\ x$

Can we unify FixVal and FixFun somehow?

The "best fixed point" combinator

We define a combinator `FixBest` such that `FixVal` and `FixFun` are both instances of it.

– **`FixBest (◁) (≈) F`** picks the greatest "fixed point **x** modulo **≈**" with respect to **◁**.

`greatest (◁) (fun x => ∀y, y ≈ x → y ≈ F y)`

– **`FixVal (≈) F := FixBest (≈) (≈) F`**

returns the unique **x** s.t. **`(∀y, y ≈ x → y ≈ F y)`**

– **`FixFun (≈) F := FixBest (≈)`** ($\angle_F$) **`F`**

where $\angle_F$ is a comparison function on fixed points of **F** designed such that its greatest element is exactly the optimal fixed point of **F**.

Recovering extraction

By moving to a non-constructive logic,

we break the extraction mechanism.

How can we recover extraction?

Extraction of fixed points

Fix is not constructive: it relies on Hilbert's epsilon. Yet, we can manually extract fixed points towards executable code using a let-rec construct. Intuitively:

From Coq:

```
Definition Log log x :=
  if x <= 1 then 0 else 1 + log (x/2).

Definition log := Fix Log.
```

To Caml: (assuming uppercase identifiers are accepted by Caml)

```
let Log log x =
  if x <= 1 then 0 else 1 + log (x/2)

let rec log = Log log
```

$\rightarrow$ How can we implement this in a systematic manner?

Extraction of the combinators

In Haskell:

```
Extract Constant FixBest =>
  "(\F -> let x = F x in x)".
```

In Caml: (where lazy types are explicit)

```
Extract Constant FixFun =>
  "(fun F -> let rec f x = F f x in f)".
```

```
Extract Constant FixVal =>
  "(fun F -> let rec x = lazy (Lazy.force (F x)) in x)".
```

Remark: proof that types are inhabited are all erased through the extraction process.

Summary and conclusion

Some examples formalized

Recursion:	**Lines of proofs**
– log function	2
– gcd function	3
– div function	3
– nested zero function	3
– trees with list of subtrees	4
– Ackermann's function	3
– McCarthy's function	8

Co-recursion: (≈ 100 lines to establish a new c.o.f.e.)

– constant stream	3
– mutually-defined streams	9
– filter on streams	13
– "product" of infinite trees	3+14+7

Contribution

1) Spot the interest of the optimal fixed point

$\rightarrow$ and implement the first formal proof of this theory

$\rightarrow$ first proper support for partial corecursive functions

2) Invariants in contraction condition for c.o.f.e.'s

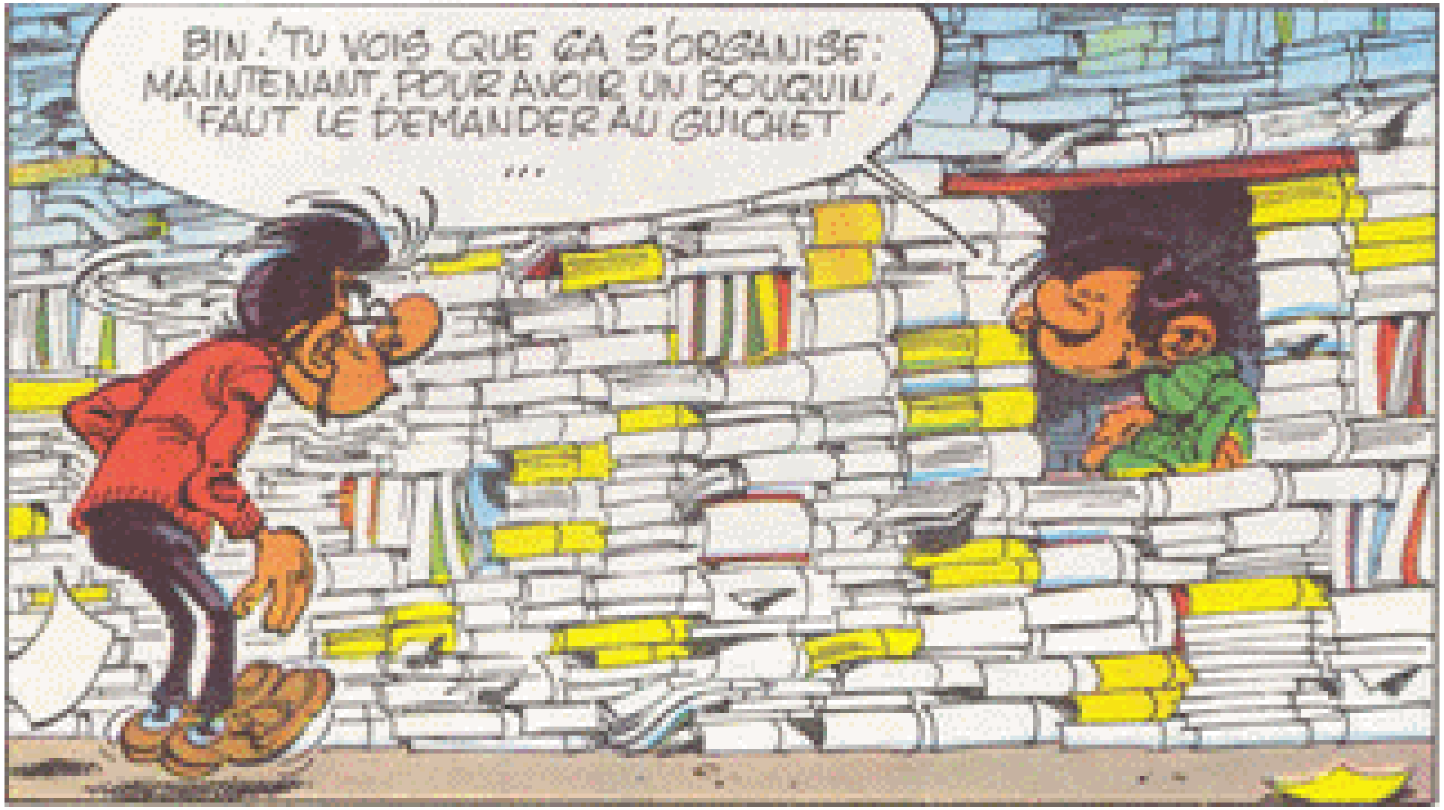
$\rightarrow$ many more co-inductive definitions are supported

3) Unify the theory of contraction conditions

$\rightarrow$ proved that all contraction conditions can be derived from the contr. condition for c.o.f.e.'s with invariants

4) Unify the generic fixed point combinators

$\rightarrow$ FixFun and FixVal derivable from FixGreatest



Things are now sorted out!

Conclusion

Optimal fixed points:

– little use as a theory of circular *program* definitions

– tool of choice to justify circular *logical* definitions

Contraction conditions:

– all contraction conditions derivable from a single one

– support a very large scope of circular definitions

– while reasoning entirely within the logic of the prover

Generic fixed point combinators:

– allow to separate definitions from their justification

– allow to encode let-rec in a systematic manner

– extraction is simple because the functional is explicit

Future Work

Generate corollaries automatically

– given the arity of the function
– given the number of mutually recursive values
– with or without invariant
– for partial or for total functions

Tactics to help proving contraction conditions

– proofs typically follows the structure of the code
– automation possible if Ltac could analyse "match"
– automate the construction of a c.o.f.e.

Applications of the combinator

– release a Coq library exporting the combinators
– implement a tool to convert from pure-Caml to Coq

Thanks!

For more information: *The Optimal Fixed Point Combinator*
http://arthur.chargueraud.org/research/2010/fix

Restrictions implemented in Coq

→ There is one little exception to termination criteria: higher-order functions can be unfolded on-the fly. This allows Coq to accept definitions such as:

```
Fixpoint size t := match t with
| Leaf => 1
| Node ts =>
    List.fold_right (fun t' a => a + size t') 1 ts
```

→ While this is convenient, it also has a drawback:

```
Definition ignore (n:nat) := 0.
Fixpoint f x := ignore (f x). (* accepted *)
let rec f x = ignore (f x).   (* extracted code *)
```

→ The extracted code can diverge in call-by-value!

Strong normalization is preserved only with lazy eval.

Contr. condition and divergence

Contraction conditions don't ensure termination.

```
Definition F f x := ignore (f x).

Contraction condition:
   forall f1 f2 x,
     (forall y, y < x -> f1 y = f2 y) ->
     F f1 x = F f2 x

Hypothesis: forall y, y < x -> f1 y = f2 y
Goal: F f1 x = F f2 x
Goal: ignore (f1 x) = ignore (f2 x)
Goal: 0 = 0
```

$\rightarrow$ It is not possible to state, inside the logic of Coq, a proposition that characterizes only terms that terminate under call-by-value evaluation.

Interest of recursive predicates

1) Can be more compact than an inductive def.

```
Inductive sorted : list A -> Prop :=
| sorted_nil : sorted nil
| sorted_one : forall x, sorted (x::nil)
| sorted_two : forall x y l,
    (x <= y) -> sorted (y::l) -> sorted (x::y::l).
```

```
Fixpoint sorted l := match l with
| x::y::l' => (x <= y) /\ sorted (x::l') | _ => True.
```

2) Fixpoints support negative occurences

```
Inductive models :=
| models_arrow : forall i v T1 T2,
   (∀x, ∀j<i, models j x T1 -> models j (v x) T2) ->
   models f (Arrow T1 T2).                 (* rejected *)
```

```
Fixpoint models i v T := match T with
| Arrow T1 T2 => forall x, forall j < i,
   models j x T1 -> models j (v x) T2. (* accepted *)
```

Origins of the contraction condition

How to come up with the contraction condition?

```
(forall f1 f2 x,
  (forall y, mu y < mu x -> f1 y = f2 y) ->
  F f1 x = F f2 x)
```

Start from the fixed point equation:

```
log x = Log log x
```

Unfolding the definition of `log`, we have to prove:

```
  Fixn_run Log (1 + mu x) x
= Log (fun y => Fixn_run Log (1 + mu y) y) x
```

Unfolding the definition of `Fixn_run`, it becomes:

```
  Log (fun y => Fixn_run Log (mu x) y) x
= Log (fun y => Fixn_run Log (1 + mu y) y) x
```

This suggests a proof by induction, on something like Log f1 x = Log f2 x, with hypotheses on `f1` and `f2`.

Another presentation

Fix can be applied on the fly to any functional

```
Definition log := Fix (fun log x =>
  if x <= 1 then 0 else 1 + log (x/2)).

Lemma log_fix : forall x,
   log x  =  if x <= 1 then 0 else 1 + log (x/2).
```

The unfolding of the definition is more direct. However, the price to pay is a duplication of the source code.

```
x > 0  |-  log x < x
```

```
rewrite log_fix.
```

```
x > 0  |-  (if x <= 1 then 0 else 1 + log(x/2)) < x
```