From guarded to well-founded

Formalizing Coq's guard condition

Cyprien Mangin
cyprien.mangin@m4x.org

Matthieu Sozeau
matthieu.sozeau@inria.fr

Inria Paris & IRIF, Université Paris-Diderot

July 8, 2018

Outline

1. Guard condition and trust
2. Translating guarded definitions
3. Current state of the implementation

Outline

1. Guard condition and trust
2. Translating guarded definitions
3. Current state of the implementation

Recursive definitions

In Coq, we have `Fixpoint` and `match` instead of recursors.

```
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
| O => m
| S n' => S (plus n' m)
end.
```

Recursive definitions

In COQ, we have `Fixpoint` and `match` instead of recursors.

```
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
| O ⇒ m
| S n' ⇒ S (plus n' m)
end.
```

The two presentations should be equivalent.

Recursive definitions

In Coq, we have `Fixpoint` and `match` instead of recursors.

```
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
| O ⇒ m
| S n' ⇒ S (plus n' m)
end.
```

The two presentations should be equivalent.

```
Fixpoint nat_rect (P : nat → Type) (f0 : P 0)
  (fS : forall n, P n → P (S n)) (n : nat) : P n :=
match n with
| O ⇒ f0
| S n' ⇒ fS n' (nat_rect P f0 fS n')
end.
```

Deep recursion

```
Inductive Even : nat → Prop :=
zeven : Even 0 | seven {n : nat} : Even n → Even (S (S n)).

Inductive Odd : nat → Prop :=
oneodd : (Odd (S 0)) | sodd {n : nat} : Odd n → Odd (S (S n)).

Definition EO (n : nat) := {Even n} + {Odd n}.

Definition aux {n : nat} (H : EO n) : EO (S (S n)) :=
match H with left p ⇒ left (seven p) | right p ⇒ right (sodd p) end.

Fixpoint evod (n : nat) : EO n :=
match n with
  | 0 ⇒ left zeven
  | S m ⇒
    match m with
      | 0 ⇒ right oneodd
      | S p ⇒ aux (evod p)
    end end.
```

The guard condition

We need to make sure that every `Fixpoint` terminates.

- Syntactic condition on the body of a `Fixpoint`.
- For each variable in the current context, track whether it is a subterm of the recursive argument.
- For each recursive call, check that the recursive argument is a subterm.

The guard condition

We need to make sure that every `Fixpoint` terminates.

- Syntactic condition on the body of a `Fixpoint`.
- For each variable in the current context, track whether it is a subterm of the recursive argument.
- For each recursive call, check that the recursive argument is a subterm.

```
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
| O ⇒ m
| S n' ⇒ S (plus n' m)
end.
```

The guard condition

We need to make sure that every `Fixpoint` terminates.

- Syntactic condition on the body of a `Fixpoint`.
- For each variable in the current context, track whether it is a subterm of the recursive argument.
- For each recursive call, check that the recursive argument is a subterm.

```
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
| O => m
| S n' => S (plus n' m)
end.
```

Issues of trust

Are the two presentations really equivalent?

Issues of trust

Are the two presentations really equivalent?

- In 1994, yes: Eduardo Giménez reduces guarded definitions to recursors.

Issues of trust

Are the two presentations really equivalent?

- In 1994, yes: Eduardo Giménez reduces guarded definitions to recursors.
- Afterwards. . . not so clear anymore.

The guard condition evolved over the years.

Issues of trust

Are the two presentations really equivalent?

- In 1994, yes: Eduardo Giménez reduces guarded definitions to recursors.
- Afterwards... not so clear anymore.

The guard condition evolved over the years.

- Commutative cuts

Issues of trust

Are the two presentations really equivalent?

- In 1994, yes: Eduardo Giménez reduces guarded definitions to recursors.
- Afterwards... not so clear anymore.

The guard condition evolved over the years.

- Commutative cuts
- A `match` can be a subterm if all branches are a subterm.

Issues of trust

Are the two presentations really equivalent?

- In 1994, yes: Eduardo Giménez reduces guarded definitions to recursors.
- Afterwards... not so clear anymore.

The guard condition evolved over the years.

- Commutative cuts
- ~~A `match` can be a subterm if all branches are a subterm.~~
- A `match` with certain restrictions can be a subterm if all branches are a subterm.

Issues of trust

Are the two presentations really equivalent?

- In 1994, yes: Eduardo Giménez reduces guarded definitions to recursors.
- Afterwards. . . not so clear anymore.

The guard condition evolved over the years.

- Commutative cuts
- ~~A `match` can be a subterm if all branches are a subterm.~~
- A `match` with certain restrictions can be a subterm if all branches are a subterm.
- . . . ?

$\Rightarrow$ We want a current justification of the guard condition.

Our proposal: relying on well-foundedness

The code that checks for the guard condition tracks whether each variable is a subterm.

Our proposal: relying on well-foundedness

The code that checks for the guard condition tracks whether each variable is a subterm.

Use the subterm relation as a well-founded relation.
For each such subterm, add a Coq proof that it is a subterm.

Our proposal: relying on well-foundedness

The code that checks for the guard condition tracks whether each variable is a subterm.

Use the subterm relation as a well-founded relation.
For each such subterm, add a Coq proof that it is a subterm.

```
Definition nat_subterm : relation nat.
Theorem wf_nat_subterm : well_founded nat_subterm.

(* Large subterms. *)
Definition nat_subterm_eq : relation nat.

Definition nat_case (N : nat) (x : nat) (Hsub : nat_subterm_eq x N)
  (P : nat → Type) (f0 : P 0)
  (fS : forall (y : nat), nat_subterm y N → P (S y)) : P x.

Definition plus_body (n m : nat)
  (F : forall (n' m' : nat), nat_subterm n' n → nat) : nat.
```

Outline

1 Guard condition and trust

2 Translating guarded definitions

3 Current state of the implementation

Back to the guard condition

Seen from the top, rather straightforward:

- Recursively go through the function body, while collecting subterm information in the context.
- At a recursive call, check that we have a subterm in the recursive position.

Back to the guard condition

Seen from the top, rather straightforward:

- Recursively go through the function body, while collecting subterm information in the context.
- At a recursive call, check that we have a subterm in the recursive position.

To check the guard on...

```
match c return p with
| C_i x y z ⇒ t
end
```

...check that `c` and `p` are guarded, then compute the recursive information on `x`, `y`, `z` depending on the information on `c`, and check `t` under this context.

Back to the guard condition

Seen from the top, rather straightforward:

- Recursively go through the function body, while collecting subterm information in the context.
- At a recursive call, check that we have a subterm in the recursive position.

To check the guard on...

```
(fun x ⇒ t)
```

...check that `t` is guarded under a context where `x` is not a subterm

Back to the guard condition

Seen from the top, rather straightforward:

- Recursively go through the function body, while collecting subterm information in the context.
- At a recursive call, check that we have a subterm in the recursive position.

To check the guard on...

```
match c with
| C f ⇒ (fun x ⇒ t)
end y
```

...check that `t` is guarded under a context where `x` is not a subterm...unless there was some term applied to a `match` previously.

Back to the guard condition

Seen from the top, rather straightforward:

- Recursively go through the function body, while collecting subterm information in the context.
- At a recursive call, check that we have a subterm in the recursive position.

To check the guard on...

```
f x y z
```

...when `f` is the function being defined, check that the argument in recursive position is a subterm.

The subterm relation

For each inductive type, we can systematically define a direct subterm relation.

```
Inductive nat : Set :=
O : nat | S : nat → nat.

Inductive nat_direct_subterm : relation nat :=
  nat_ds_1 : forall (n : nat), nat_direct_subterm n (S n).
```

[1] See for instance EQUATIONS

The subterm relation

For each inductive type, we can systematically define a direct subterm relation.

```
Inductive nat : Set :=
O : nat | S : nat → nat.

Inductive nat_direct_subterm : relation nat :=
  nat_ds_1 : forall (n : nat), nat_direct_subterm n (S n).
```

The subterm relation is its transitive closure.

```
Definition nat_subterm : relation nat :=
  clos_trans nat nat_direct_subterm.
```

This is already done automatically by some tools[1], as well as proving its well-foundedness.

[1] See for instance EQUATIONS

Translation

```
Fixpoint evod (n : nat) : EO n :=
match n with
  | O ⇒ left zeven
  | S m ⇒
    match m with
      | O ⇒ right oneodd
      | S p ⇒ aux (evod p)
    end end.

Definition evod_body (n : nat)
  (F : forall (m : nat), nat_subterm m n → EO m) : EO n :=
  nat_case n n r_refl EO
    (left zeven)
    (fun m Hsub ⇒ nat_case n m (r_step Hsub) (fun m ⇒ EO (S m))
      (right oneodd)
      (fun p Hsub ⇒ aux (F p Hsub))).
```

Translation

```
Fixpoint evod (n : nat) : EO n :=
match n with
  | O ⇒ left zeven
  | S m ⇒
    match m with
      | O ⇒ right oneodd
      | S p ⇒ aux (evod p)
    end end.

Definition evod_body (n : nat)
  (F : forall (m : nat), nat_subterm m n → EO m) : EO n :=
  nat_case n n r_refl EO
    (left zeven)
    (fun m Hsub ⇒ nat_case n m (r_step Hsub) (fun m ⇒ EO (S m))
      (right oneodd)
      (fun p Hsub ⇒ aux (F p Hsub))).
```

Translation

```
Fixpoint evod (n : nat) : EO n :=
match n with
  | O ⇒ left zeven
  | S m ⇒
    match m with
      | O ⇒ right oneodd
      | S p ⇒ aux (evod p)
    end end.

Definition evod_body (n : nat)
  (F : forall (m : nat), nat_subterm m n → EO m) : EO n :=
  nat_case n n r_refl EO
    (left zeven)
    (fun m Hsub ⇒ nat_case n m (r_step Hsub) (fun m ⇒ EO (S m))
      (right oneodd)
      (fun p Hsub ⇒ aux (F p Hsub))).
```

Translation

```
Fixpoint evod (n : nat) : EO n :=
match n with
  | O ⇒ left zeven
  | S m ⇒
    match m with
      | O ⇒ right oneodd
      | S p ⇒ aux (evod p)
    end end.

Definition evod_body (n : nat)
  (F : forall (m : nat), nat_subterm m n → EO m) : EO n :=
  nat_case n n r_refl EO
    (left zeven)
    (fun m Hsub ⇒ nat_case n m (r_step Hsub) (fun m ⇒ EO (S m))
      (right oneodd)
      (fun p Hsub ⇒ aux (F p Hsub))).
```

What about mutual inductive types?

We have two choices:

- Define heterogeneous subterm relations for each pair of mutually inductive types.

What about mutual inductive types?

We have two choices:

- Define heterogeneous subterm relations for each pair of mutually inductive types.
  $\Rightarrow$ simple enough
  $\Rightarrow$ need to introduce a more complicated `Fix` to handle several types at once

What about mutual inductive types?

We have two choices:

- Define heterogeneous subterm relations for each pair of mutually inductive types.
  $\Rightarrow$ simple enough
  $\Rightarrow$ need to introduce a more complicated `Fix` to handle several types at once
- Fallback to the single inductive case with an index.

What about mutual inductive types?

We have two choices:

- Define heterogeneous subterm relations for each pair of mutually inductive types.
  $\Rightarrow$ simple enough
  $\Rightarrow$ need to introduce a more complicated `Fix` to handle several types at once
- Fallback to the single inductive case with an index.
  $\Rightarrow$ a bit more verbose and complicated
  $\Rightarrow$ it reuses the standard `Fix`

What about mutual inductive types?

We have two choices:

- Define heterogeneous subterm relations for each pair of mutually inductive types.
  $\Rightarrow$ simple enough
  $\Rightarrow$ need to introduce a more complicated `Fix` to handle several types at once
- Fallback to the single inductive case with an index.
  $\Rightarrow$ a bit more verbose and complicated
  $\Rightarrow$ it reuses the standard `Fix`

In any case, there are some problems to solve with sorts.

What about nested inductive types?

```
Inductive rose : Set :=
| node : list rose → rose.
```

What about nested inductive types?

```
Inductive rose : Set :=
| node : list rose → rose.
```

- The correct solution is not clear yet...

What about nested inductive types?

```
Inductive rose : Set :=
| node : list rose → rose.
```

- The correct solution is not clear yet...
- For now, we inline the definition of the nested type.

```
Inductive Rose' : Set :=
| node' : listRose' → Rose'
with listRose' : Set :=
| nil' : listRose'
| cons' : Rose' → listRose' → listRose'.
```

- Ideally, we would like to reuse generically what was built for lists.

Evolution of the guard condition

```
Inductive True2 : Prop :=
I2: (False→ True2) → True2.

(* Using prop_ext : forall P Q, (P ↔ Q) → P = Q. *)
Theorem Heq: (False → True2) = True2.
```

Evolution of the guard condition

```
Inductive True2 : Prop :=
I2: (False→ True2) → True2.

(* Using prop_ext : forall P Q, (P ↔ Q) → P = Q. *)
Theorem Heq: (False → True2) = True2.

Fixpoint con (x : True2) : False :=
match x with
I2 f ⇒ con (match Heq in _=T return T with eq_refl ⇒ f end)
end.
```

Outline

Template Coq

- Quoting library for Coq.
- Allows the user to manipulate quotations and the global environment.
- Will include a certified checker for Coq.

Assuming completion of this work...

You only need to trust the well-foundedness of `Acc` to trust any recursive definition accepted by Template Coq.

Regular trees

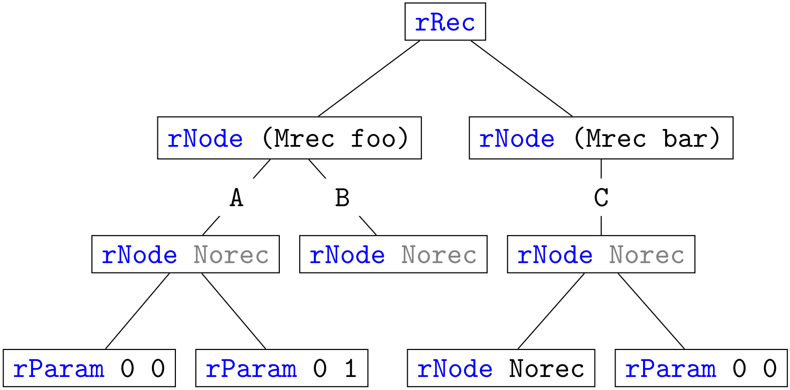
- Used to represent the recursive structure of a term.
- Infinite trees with finitely many subtrees.
- Represented in Coq's kernel by a finite datastructure with de Bruijn indices.

```
Inductive rtree (X : Set) : Set :=
| rNode (x : X) (sons : list (rtree X)) : rtree X
| rParam (i : nat) (j : nat) : rtree X
| rRec (j : nat) (defs : list (rtree X)) : rtree X.
```

From a client viewpoint, only see it as a tree of `rNode` with some payload of type `X`.

An example of regular tree

```
Inductive foo : Set :=
| A : foo → bar → foo
| B : foo
with bar : Set :=
| C : nat → foo → bar.
```



Unfolding of regular trees

A closed tree is either:

- some `rNode x sons` : the payload is `x` and its children are `sons`
- some `rRec j defs` : in that case we can consider the unfolding of this recursive node, that is `defs.(j)` where each `defs.(i)` is substituted for `rParam 0 i`.

This unfolding operation guarantees that we can always provide an actual `rNode`

Unfolding of regular trees

A closed tree is either:

- some `rNode x sons` : the payload is `x` and its children are `sons`
- some `rRec j defs` : in that case we can consider the unfolding of this recursive node, that is `defs.(j)` where each `defs.(i)` is substituted for `rParam 0 i`.

This unfolding operation guarantees that we can always provide an actual `rNode` if the tree verifies some acyclicity condition.

Positivity, guard and the rest

The next steps are:

- Implement the positivity criterion.
  $\Rightarrow$ Not too difficult, except that it relies itself on the strong normalization of Coq terms.

Positivity, guard and the rest

The next steps are:

- Implement the positivity criterion.
  $\Rightarrow$ Not too difficult, except that it relies itself on the strong normalization of Coq terms.
- Implement the guard condition.
  $\Rightarrow$ This will be a test of the usability of the implementation of regular trees.

Positivity, guard and the rest

The next steps are:

- Implement the positivity criterion.
  $\Rightarrow$ Not too difficult, except that it relies itself on the strong normalization of Coq terms.
- Implement the guard condition.
  $\Rightarrow$ This will be a test of the usability of the implementation of regular trees.
- Implement the translation of guarded functions.

Positivity, guard and the rest

The next steps are:

- Implement the positivity criterion.
  $\Rightarrow$ Not too difficult, except that it relies itself on the strong normalization of Coq terms.
- Implement the guard condition.
  $\Rightarrow$ This will be a test of the usability of the implementation of regular trees.
- Implement the translation of guarded functions.

Of course, prove all this correct all along.