The guard condition of Coq

Bruno Barras

December 19, 2006
Why this talk?

- Defining functions by recursion is very common
- Logical consistency relies heavily on termination
- Reference Manual of Coq refers to Gimenez’ paper “Codifying guard definitions with recursive schemes” (94)
- This condition has been extended over the years to support more schemes
Why this talk?

- Defining functions by recursion is very common
- Logical consistency relies heavily on termination
- Reference Manual of Coq refers to Gimenez’ paper “Codifying guard definitions with recursive schemes” (94)
- This condition has been extended over the years to support more schemes
- Bugs (or scary error messages)
Why this talk?

- Defining functions by recursion is very common
- Logical consistency relies heavily on termination
- Reference Manual of Coq refers to Gimenez’ paper “Codifying guard definitions with recursive schemes” (94)
- This condition has been extended over the years to support more schemes
- Bugs (or scary error messages)
  Uncaught exception: Assert_failure("kernel/inductive.ml",_)

Overview of the talk

1. Introduction
   - Syntactic guard criterion
   - Strictly positive inductive definitions

2. A simple criterion

3. Refinements

4. Pitfalls

5. Conclusion
Overview

1. Introduction
   - Syntactic guard criterion
   - Strictly positive inductive definitions

2. A simple criterion

3. Refinements

4. Pitfalls

5. Conclusion
A long time ago...

- Recursion was made by recursors (Gödel T).
- Only allows recursive calls on *direct* subterms.
- Cumbersome in a functional programming setting.

### Example

```coq
Definition half n :=
  fst(Rec (0,false)
      (fun (k,odd) ⇒ if odd then (k+1,false)
       else (k,true))
  n)

instead of

Fixpoint half n :=
  match n with S(S k) ⇒ half k | _ ⇒ 0 end
```
Towards syntactic guard criterion

- Proposal by Coquand (92):
  recursor = pattern-matching + fixpoint

- Gimenez’ paper (94): translation towards recursors.
  For $f : I \rightarrow T$, define $I_f$ similar to $I$ such that every subterm of type $I$ comes with its image by $f$. Then write $g : I \rightarrow I_f$ and $h : I_f \rightarrow T$.

- Blanqui (05), Calculus of Algebraic Constructions: reducibility proof (CC + higher order rewriting)

- Only work for simple criterion.
Strictly positive inductive definitions

Positivity condition

- Also crucial for consistency
- Lists
  Inductive list (A:Type) : Type :=
    nil | cons (x:A) (l:list A).
- Ordinals Inductive ord:Set :=
  0 | S(o:ord) | lim(f:nat→ord).
- Useful extension: nested inductive types
  Inductive tree:Set := None(l:list tree).
  Reuse list library
Strictly positive inductive definitions

Positivity condition (more formally)

Definition (Terms)

\[ s \mid x \mid \Pi x : T. U \mid \lambda x : T. M \mid M N \]
\[ \mid \text{Ind}(X : A)\{\vec{C}\} \mid \text{Constr}(n, I) \mid \text{Fix } F_k : T := M \]
\[ \mid \text{Match } M \text{ with } \vec{p} \Rightarrow \vec{t} \text{ end} \]

Definition (strict positivity)

\[ \Pi \vec{x} : \vec{t}. C \text{ is strictly positive w.r.t. } X \text{ if forall } i \text{ either:} \]

(Norec) \( X \) does not occur free in \( t_i \), or

(Rec) \( t_i = \Pi \vec{y} : \vec{u}. X \vec{w} \) where \( X \) does not occur in \( \vec{u}\vec{w} \), or

(Nested) \( t_i = \Pi \vec{y} : \vec{u}. \text{Ind}(Y : B)\{\vec{D}\} \vec{w} \) and
  \begin{itemize}
    \item \( X \) does not occur free in \( \vec{u}\vec{w} \)
    \item \( D_i \) is strictly positive w.r.t. \( X \) forall \( i \)
Impredicativity

Recursive calls cannot be allowed on all constructor arguments

Fixpoint F (x:I) : False :=
  match x with
    C f => F (f I x)
  end

Definition (recursive positions)

constructors arguments that satisfy (Rec) or (Nested) clause of positivity.
Different instances of the same inductive type may have different sets of recursive positions

Example (Str(list) and Str(tree))
While checking positivity, we build a regular tree that identifies recursive positions. But: parameters not instanciated

**Lemma**

*The computed tree is the set of paths that cannot contain an infinite number of inductive objects.*
Overview

1. Introduction
   - Syntactic guard criterion
   - Strictly positive inductive definitions

2. A simple criterion

3. Refinements

4. Pitfalls

5. Conclusion
Size information

(strict) \( \sigma^- ::= \top | \tau^- \)

(non-strict) \( \sigma^+ ::= \bot | \tau^+ \)

(size info) \( \sigma ::= \sigma^+ \cup \sigma^- \)

A map \( \rho \) associates size information to every variable
Guard condition in short

- A judgement $\rho \vdash^S M \Rightarrow \sigma$ meaning that $M$ has size information $\sigma$, where $\rho$ associates size information to variables.
- A judgement $M \in \text{Check}_{\rho}^{F,k}$ meaning that $M$ does recursive calls to $F$ only on strict subterms, as specified by $\rho$.
- Pattern-matching propagates information on pattern variables
  $$\text{Constr}(i, I) \ x_1 \ldots x_k \mid \sigma = \{(x_j, \sigma.i.\ j^-) \mid j \leq k\}$$

Remarks

- Easy encoding of recursors as fix+match (non regression).
- Allow recursive calls on deep subterms.
Definition of the condition (1)

Typing rule:

\[
\Gamma (F : T) \vdash M : T \quad M \in \text{Guard}^F_k \\
\Gamma \vdash (\text{Fix} F_k : T := M) : T
\]

\[
t_k = \text{Ind}(X : A)\{\vec{C}\} \bar{u} \quad \text{Str}(X, \vec{C}) = \tau \quad M \in \text{Check}^{F,k}_{\{(x_k, \tau^+)\}} \\
\lambda \vec{x} : \vec{t}. M \in \text{Guard}^F_k
\]
Definition of the condition (2)

\[ M \in \text{Check}_{\rho}^{f,k} \quad \rho \vdash^S M \Rightarrow \sigma \quad \forall i. b_i \in \text{Check}_{\rho \cup (p_i|\sigma)}^{f,k} \]

\[ \text{Match } M \text{ with } \vec{p} \Rightarrow \vec{b} \text{ end } \in \text{Check}_{\rho}^{f,k} \]

\[ \rho \vdash^S t_k \Rightarrow \sigma^- \quad \forall i, t_i \in \text{Check}_{\rho}^{f,k} \]

\[ f \vec{t} \in \text{Check}_{\rho}^{f,k} \]
Definition of the condition (boring cases)

Simply check recursively that subexpressions are guarded

\[
\begin{align*}
& f \notin FV(M) \quad T \in \text{Check}^f \kappa \quad U \in \text{Check}^f \kappa \\
\Rightarrow & \quad M \in \text{Check}^f \kappa \\
\end{align*}
\]

\[
\begin{align*}
& T \in \text{Check}^f \kappa \quad U \in \text{Check}^f \kappa \\
\Rightarrow & \quad \prod x : T \ U \in \text{Check}^f \kappa \\
\end{align*}
\]

\[
\begin{align*}
& T \in \text{Check}^f \kappa \quad U \in \text{Check}^f \kappa \\
\Rightarrow & \quad \lambda x : T \ U \in \text{Check}^f \kappa \\
\end{align*}
\]

\[
\begin{align*}
& M \in \text{Check}^f \kappa \quad N \in \text{Check}^f \kappa \\
\Rightarrow & \quad M \ N \in \text{Check}^f \kappa \\
\end{align*}
\]
Subterms

\[(x, \sigma) \in \rho \quad \frac{\rho \vdash^S x \bar{t} \Rightarrow \sigma}{\rho \vdash^S \bar{t} \Rightarrow \sigma} \quad \frac{\rho \vdash^S M \Rightarrow \sigma}{\rho \vdash^S \lambda x : A. M \Rightarrow \sigma}\]
Overview

1. Introduction
   - Syntactic guard criterion
   - Strictly positive inductive definitions

2. A simple criterion

3. Refinements

4. Pitfalls

5. Conclusion
 Checking guard modulo reduction

In fact, the typing rule for fixpoints is:

\[
\begin{align*}
\Gamma (F : T) & \vdash M : T \\
M & \rightarrow^* M' \\
M' & \in \text{Guard}^F_k
\end{align*}
\]

\[\Gamma \vdash (\text{Fix } F_k : T := M) : T\]

Breaks strong normalization!

Example

Fixpoint \( F \ n := \text{let } x := F \ n \text{ in } 0 \).
Eval compute in \((F \ 0)\).
Pattern-matching

\[ \forall i, \rho \vdash_S b_i \Rightarrow \sigma_i \]
\[ \rho \vdash_S \text{Match } M \text{ with } \vec{p} \Rightarrow \vec{b} \text{ end} \Rightarrow \bigcap \vec{\sigma} \]

Example

Definition \( \text{pred } n \ (H : n <> 0) := \)

\[
\begin{align*}
\text{match } n \text{ with} \\
0 & \Rightarrow \text{match } H \_ \text{ with end} \\
| S \ k & \Rightarrow k \\
\text{end.}
\end{align*}
\]

Fixpoint \( F x := \)

\[
\text{if eq_nat_dec } x \ 0 \ \text{then } 0 \ \text{else } F \ (\text{pred } x)
\]
Fixpoints as argument of $F$

- A fix returns a strict subterm if its body does
- Size information of recursive argument is propagated

$$\rho \vdash^{S} u_n \Rightarrow \sigma \quad \rho \cup \{(G, \tau^-), (x_n, \sigma)\} \vdash^{S} M \Rightarrow \tau^-$$

$$\rho \vdash^{S} (\text{Fix } G_n : T := \lambda \vec{x} : \vec{t} \cdot M) \; \vec{u} \Rightarrow \tau^-$$

**Example**

Fixpoint $F \ x \ y :=$
if ‘‘$x \leq y$’’ then $x$ else $F \ (x-S(y)) \ y$
Nested fixpoints

\[ \rho \vdash^S u_n \Rightarrow \sigma \quad M \in \text{Check}_{\rho\{x_k,\sigma\}}^F,k \quad T \in \text{Check}_{\rho}^F,k \quad \vec{u} \in \text{Check}_\rho^{F,k} \]

\[ \frac{}{(\text{Fix } G_n : T := M) \quad \vec{u} \in \text{Check}_\rho^{F,k}} \]

Example (size of a tree)

Fixpoint size (t:tree) :=
  match t with
    Node l \Rightarrow \text{fold_right \ (fun t' \ n \Rightarrow n+size t')} 1 l
  end.
Overview

1. Introduction
   - Syntactic guard criterion
   - Strictly positive inductive definitions

2. A simple criterion

3. Refinements

4. Pitfalls

5. Conclusion
Nested vs. mutual inductive types

Example (Guard violated)

Fixpoint size (t:tree) :=
  match t with
    Node l ⇒ S(size_forest l)
  end
with size_forest (l:list tree) :=
  match l with
    nil ⇒ 0
  | t::l’ ⇒ size t + size l’
  end.

Mutual inductive types can be used in the context of both mutual fixpoints and nested fixpoints.
Nested inductive types cannot be used in the context of mutual fixpoints.
Introduction

- Syntactic guard criterion
- Strictly positive inductive definitions

A simple criterion

Refinements

Pitfalls

Conclusion
• Many extensions already,
• Many are still missing (syntactic criterion)
So why this talk?

- An opportunity to stop and think
- A highly critical (implementation) bug found: apply the patch!
- Syntactic criterions are dead: Gimenez, Blanqui, Barthe (and...) moved to type-based guard verification (size annotation)