Coq Manual (Section 4.4.5)

G.C. Alexandru Jochem Raat

May 26, 2020

Introduction

This section gives an overview of inductive definitions and their rules in Coq:

- Inductive definitions and their typing/correctness rules
- Template polymorphism on inductive definitions
- Destructors of inductive definitions

Structure

Notation

Ind $[p]$ $(\Gamma_I := \Gamma_C)$, where:

- p: number of parameters
- Γ_I: types of the inductive types, *"type constructors"*
- Γ_C: types of the constructors of the inductive types, *"value constructors"*

Example of parameterized lists:

$$\text{Ind } [1] \left([\text{list} : \text{Set} \to \text{Set}] := \left[\begin{array}{rcl} \text{nil} & : & \forall A : \text{Set},\ \text{list } A \\ \text{cons} & : & \forall A : \text{Set},\ A \to \text{list } A \to \text{list } A \end{array} \right] \right)$$

```
Inductive list (A:Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.
```

Example 2

$$\mathsf{Ind}\ [0]\left(\left[\begin{array}{rcl}\mathsf{even} & : & \mathsf{nat}\to\mathsf{Prop}\\ \mathsf{odd} & : & \mathsf{nat}\to\mathsf{Prop}\end{array}\right] := \left[\begin{array}{rcl}\mathsf{even}_\mathsf{O} & : & \mathsf{even}\ 0\\ \mathsf{even}_\mathsf{S} & : & \forall n,\ \mathsf{odd}\ n\to\mathsf{even}\ (\mathsf{S}\ n)\\ \mathsf{odd}_\mathsf{S} & : & \forall n,\ \mathsf{even}\ n\to\mathsf{odd}\ (\mathsf{S}\ n)\end{array}\right]\right)$$

```
Inductive even : nat -> Prop :=
| even_O : even 0
| even_S : forall n, odd n -> even (S n)
with odd : nat -> Prop :=
| odd_S :forall n, even n -> odd (S n).
```

Arity of sort

> *"A type T is an arity of sort s if it converts to the sort s or to a product $\forall x : T.U$ with U an arity of sort s." (p. 211)*

$\forall A : \text{Prop}.A \to \text{Prop}$, is an arity of sort Prop.
Set, is an arity of sort Set
$A \to B \to \text{Set}$, is an arity of sort Set.

Arity

Any type is an *arity* if there is a sort s, for which it is an arity of sort s.

So, $A \to B \to Set$ is an arity, but $A \to B$ is not.

Positivity

- Condition used to prevent logical contradiction caused by inductive definitions
- By preventing the type constructors from being used in a wrong way in the value constructors

Three definitions for a type T (all in regards to a term X):

1. Positivity condition
   - $T = X\ t_1 ... t_n$ and X does not occur freely in t_i
   - $T = \forall x : U, V$, where U is strictly positive and V satisfies the positivity condition
2. Strict positivity
   - X does not occur in T
   - T converts to $X\ t_1 ... t_n$ and X does not occur in t_i
   - T converts to $\forall x : U, V$ and X does not occur in U and strictly positively in V
   - T converts to $I\ a_1 \ldots a_m\ t_1 \ldots t_p$, where I is inductive with m parameters and X does not occur in t_i and the (instantiated) types of constructors of I satisfy nested positivity.
3. Nested positivity
   - $T = (I\ b_1 \ldots b_m\ u_1 \ldots u_p$, I is an inductive type with m parameters and X does not occur in u_i
   - $T = \forall x : U, V$ and U strictly positive and V nested positive

Correctness Rule

Let E be a global environment and Γ_P, Γ_I, Γ_C be contexts such that Γ_I is $[I_1 : \forall \Gamma_P, A_1;\ ...;\ I_k : \forall \Gamma_P, A_k]$, and Γ_C is $[c_1 : \forall \Gamma_P, C_1;\ ...;\ c_n : \forall \Gamma_P, C_n]$. Then

W-Ind

$$\frac{\mathcal{WF}(E)[\Gamma_P] \qquad (E[\Gamma_I; \Gamma_P] \vdash C_i : s_{q_i})_{i=1...n}}{\mathcal{WF}(E;\ \mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C))[]}$$

provided that the following side conditions hold:

- $k > 0$ and all of I_j and c_i are distinct names for $j = 1...k$ and $i = 1...n$,
- p is the number of parameters of $\mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C)$ and Γ_P is the context of parameters,
- for $j = 1...k$ we have that A_j is an arity of sort s_j and $I_j \notin E$,
- for $i = 1...n$ we have that C_i is a type of constructor of I_{q_i} which satisfies the positivity condition for $I_1...I_k$ and $c_i \notin E$.

One can remark that there is a constraint between the sort of the arity of the inductive type and the sort of the type of its constructors which will always be satisfied for the impredicative sorts SProp and Prop but may fail to define inductive type on sort Set and generate constraints between universes for inductive types in the Type hierarchy.

Examples!

Switch to the editor, with the example Coq file! :)

Structure

Template Polymorphism

- Inductive types, polymorphic over *Sort*
- Used when arity is in `Type` hierarchy

Example

```
Inductive option (A:Type) : Type :=
| None : option A
| Some : A -> option A.
```

- Should be able to package terms of any sort. E.g. `Type(1)`, `Set`, `Prop`, etc.
- Should return the lowest applicable sort, not just `Type`

Example

For example, if `a : Set` then `option a` should be in `Set` as well.

This way a function of type `Set -> Set` that can take a value like `2`, can also take a value like `Some 2`.

Typing rule

- If A is an arity of some sort, A/s is A with its sort replaced by the sort s.
- We have r recursively uniform parameters. These are the same in all occurrences of I_j in all constructors, even in the hypotheses.
- The sorts s_i are are introduced by the inductive declaration and allow all eliminations.

$$\frac{\left\{\begin{array}{l}\mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C) \in E \\ (E[] \vdash q_l : P'_l)_{l=1\ldots r} \\ (E[] \vdash P'_l \leq_{\beta\delta\iota\zeta\eta} P_l\{p_u/q_u\}_{u=1\ldots l-1})_{l=1\ldots r} \\ 1 \leq j \leq k\end{array}\right.}{E[] \vdash I_j\ q_1 \ldots q_r : \forall [p_{r+1} : P_{r+1};\ \ldots;\ p_p : P_p], (A_j)_{/s_j}}$$

Examples

```
Check (fun A:Type => option A).
    fun A : Type => option A
         : Type -> Type

Check (fun A:Set => option A).
    fun A : Set => option A
         : Set -> Set

Check (fun A:Prop => option A).
    fun A : Prop => option A
         : Prop -> Set
```

Structure

Introduction

- Basic Question: How to *use* inductive types
- Want to retain strong normalization $\Rightarrow$ primitive recursion
- Several possible ways to go about this, in Coq the problem is factorized into:
  - pattern matching
  - recursion with guarded fixpoints

Structure

Pattern matching construct

Concrete syntax:

$$\texttt{match } m \texttt{ as } x \texttt{ in } I \texttt{ \_ } a \texttt{ return } P \texttt{ with}$$
$$\mid (c_1\ x_{11} \ldots x_{1p_1}) \Rightarrow f_1$$
$$\mid \ldots$$
$$\mid (c_n\ x_{n1} \ldots x_{np_n}) \Rightarrow f_n$$
$$\texttt{end}$$

Pattern matching construct

Concrete syntax:

$$\begin{array}{l}\texttt{match } m \texttt{ as } x \texttt{ in } I \texttt{ \_ } a \texttt{ return } P \texttt{ with} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid (c_1\ x_{11} \ldots x_{1p_1}) \Rightarrow f_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid (c_n\ x_{n1} \ldots x_{np_n}) \Rightarrow f_n \\ \texttt{end}\end{array}$$

any term, eg f, (foo bar)

Pattern matching construct

Concrete syntax:

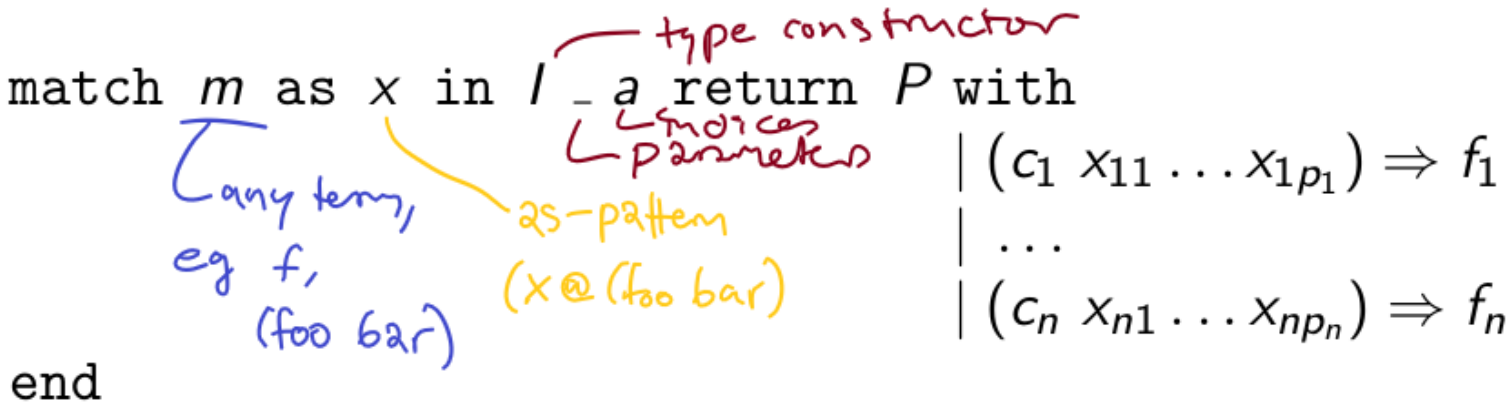
$$\texttt{match } m \texttt{ as } x \texttt{ in } I \texttt{ \_ } a \texttt{ return } P \texttt{ with}$$
$$\mid (c_1\ x_{11} \ldots x_{1p_1}) \Rightarrow f_1$$
$$\mid \ldots$$
$$\mid (c_n\ x_{n1} \ldots x_{np_n}) \Rightarrow f_n$$
$$\texttt{end}$$

any term, eg f, (foo bar)

as-pattern (x @ (foo bar))

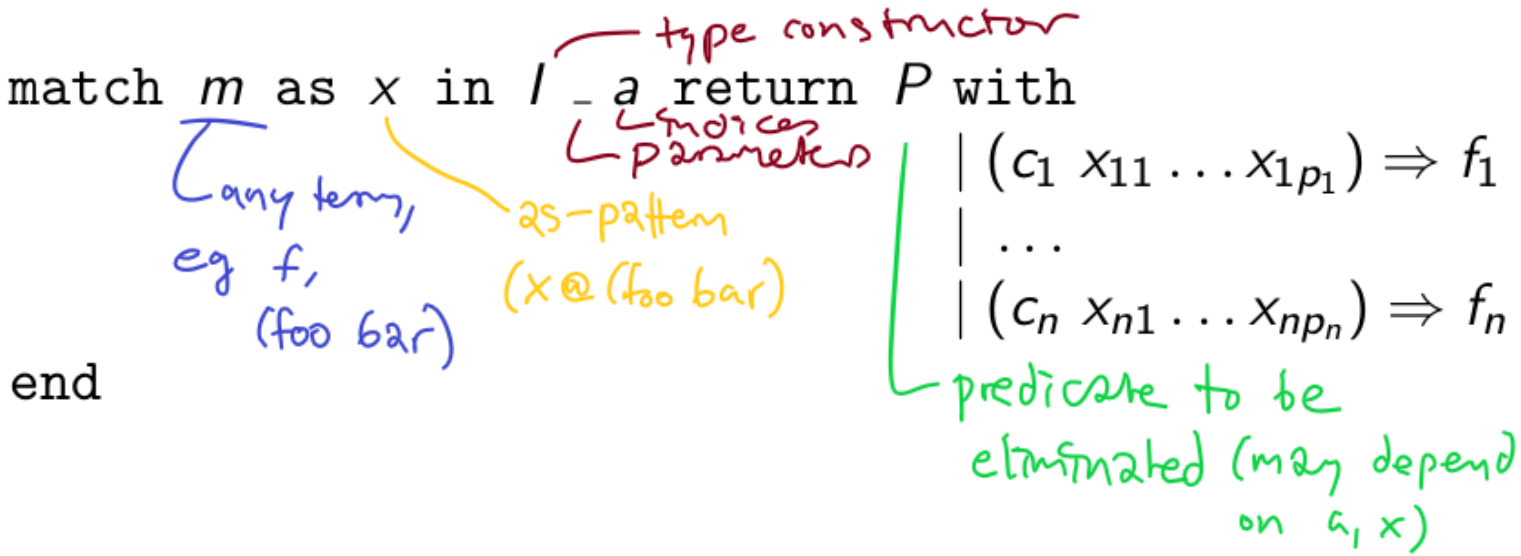
Pattern matching construct

Concrete syntax:



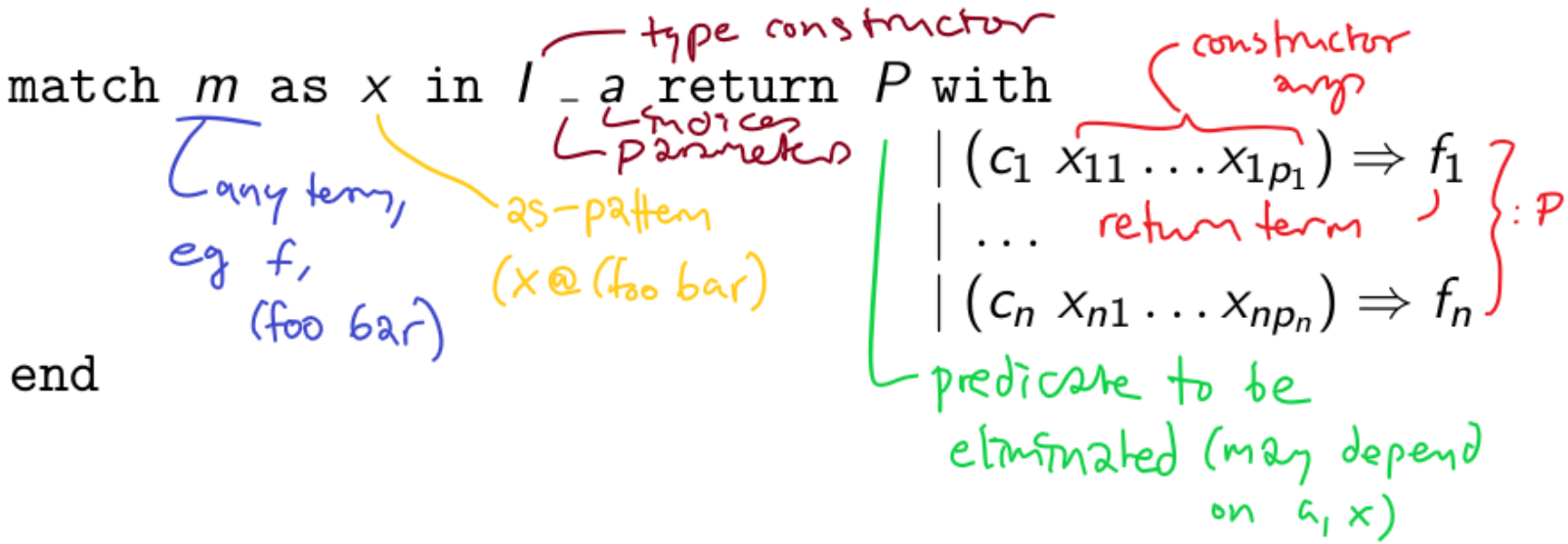
Pattern matching construct

Concrete syntax:



Pattern matching construct

Concrete syntax:



Pattern matching construct

Concrete syntax:

$$\texttt{match } m \texttt{ as } x \texttt{ in } I \texttt{ \_ } a \texttt{ return } P \texttt{ with}$$
$$\begin{array}{l} \mid (c_1\ x_{11} \ldots x_{1p_1}) \Rightarrow f_1 \\ \mid \ldots \\ \mid (c_n\ x_{n1} \ldots x_{np_n}) \Rightarrow f_n \end{array}$$
$$\texttt{end}$$

Abstract syntax:

$$\texttt{case } (m, (\lambda a x.P), \lambda x_{11} \ldots x_{1p_1}.f_1 \mid \ldots \mid \lambda x_{n1} \ldots x_{np_n}.f_n)$$

Allowed Elimination sorts

- We can't let `Prop` eliminate to `Set`, because this would mean doing a case analysis over a non-computational object
- Elimination from `Prop` to `Type` is also excluded, since it is a supertype of `Set`.
- Singleton or Empty types are exempt

Allowed Elimination sorts

Explanation of the stated rules:

Prod

$$\frac{[(I\ x) : A'|B']}{[I : \forall x : A,\ A'|\forall x : A,\ B']}$$

Set & Type

$$\frac{s_1 \in \{\mathsf{Set}, \mathsf{Type}(j)\} \qquad s_2 \in \mathcal{S}}{[I : s_1|I \to s_2]}$$

Prop

$$\frac{s \in \{\mathsf{SProp}, \mathsf{Prop}\}}{[I : \mathsf{Prop}|I \to s]}$$

Allowed Elimination sorts

Explanation of the stated rules:

Prod

$$\frac{[(I\ x) : A'|B']}{[I : \forall x : A,\ A'|\forall x : A,\ B']}$$

ensure P takes all type indices as args

Set & Type

$$\frac{s_1 \in \{\mathsf{Set}, \mathsf{Type}(j)\} \qquad s_2 \in \mathcal{S}}{[I : s_1|I \to s_2]}$$

(Note: I should have parameters already pre-applied, since P mustn't depend on them)

Prop

$$\frac{s \in \{\mathsf{SProp}, \mathsf{Prop}\}}{[I : \mathsf{Prop}|I \to s]}$$

1

[1] "P is a predicate over $n+1$ arguments: The n first ones correspond to the arguments [indices] of [the type constructor]"

Allowed Elimination sorts

Explanation of the stated rules:

Prod

$$\frac{[(I\ x) : A'|B']}{[I : \forall x : A,\ A'|\forall x : A,\ B']}$$

ensure P takes all type indices as args

Set & Type

$$\frac{s_1 \in \{\mathsf{Set}, \mathsf{Type}(j)\} \qquad s_2 \in \mathcal{S}}{[I : s_1|I \to s_2]}$$

reduce to any sort

(Note: I should have parameters already pre-applied, since P must not depend on them)

Remember, last arg of P is the term to be reduced

Prop

$$\frac{s \in \{\mathsf{SProp}, \mathsf{Prop}\}}{[I : \mathsf{Prop}|I \to s]}$$

1

[1] "P is a predicate over $n+1$ arguments [...] the last one corresponds to object m"

Allowed Elimination sorts

Explanation of the stated rules:

Prod

$$\frac{[(I\ x) : A'|B']}{[I : \forall x : A,\ A'|\forall x : A,\ B']}$$

ensure P takes all type indices as args

Set & Type

reduce to any sort

$$\frac{s_1 \in \{\mathsf{Set}, \mathsf{Type}(j)\} \qquad s_2 \in \mathcal{S}}{[I : s_1|I \to s_2]}$$

(Note: I should have parameters already pre-applied, since P must not depend on them)

Remember, last arg of P is the term to be reduced

Prop

special base case for Prop :

$$\frac{s \in \{\mathsf{SProp}, \mathsf{Prop}\}}{[I : \mathsf{Prop}|I \to s]}$$

Typing Branches of a case expression

Explanation of the stated rules:

$$\begin{array}{ll} \{c : (I\ q_1 \ldots q_r\ t_1 \ldots t_s)\}^P & \equiv (P\ t_1 \ldots\ t_s\ c) \\ \{c : \forall x : T,\ C\}^P & \equiv \forall x : T,\ \{(c\ x) : C\}^P \end{array}$$

Typing Branches of a case expression

Explanation of the stated rules:

$$\frac{\{c : (I\ q_1 \ldots q_r\ t_1 \ldots t_s)\}^P}{\{c : \forall x : T,\ C\}^P} \quad \begin{array}{l} \equiv (P\ t_1 \ldots\ t_s\ c) \\ \equiv \forall x : T,\ \{(c\ x) : C\}^P \end{array}$$

1

[1] compare base case of Def. *type of constructor*

Typing Branches of a case expression

Explanation of the stated rules:

base case

$$\{c : (I\ q_1 \ldots q_r\ t_1 \ldots t_s)\}^P \equiv (P\ t_1 \ldots\ t_s\ c)$$

$$\{c : \forall x : T,\ C\}^P \longrightarrow \equiv \forall x : T,\ \{(c\ x) : C\}^P$$

pull out products

1

[1] compare inductive case of Def. *type of constructor*

Example

Dependent Elimination

```
Inductive vect (A : Set) : nat -> Set :=
| vnil : vect A 0
| vcons (a : A) {n} (v : vect A n) : vect A (S n).

Fixpoint concat_vect0 {A} {n m} (l:vect A n) (r:vect A m) {struct l} :
  vect A (n + m) :=
  match l as x in vect _ n return vect A (n + m) with
  | vnil _ => r
  | vcons _ a l' => vcons A a (concat_vect1 l' r)
  end.
```

$$P : \forall (n : \mathbb{N}), \forall (\_ : vect\ A\ n), vect\ A\ (n + m)$$

Translation

$$
\begin{aligned}
\{(vnil\ \mathbb{N})\}^P &\equiv \{(vnil\ \mathbb{N}) : (vect\ A\ 0)\}^P \\
&\equiv P\ 0\ (vnil\ A\ 0) \equiv vect\ A\ (0 + m) \\
\{(vcons\ \mathbb{N})\}^P &\equiv \{(vcons\ \mathbb{N}) : (\forall (a : A), \forall (v : vect\ A\ n), vect\ A\ (S\ n))\}^P \\
&\equiv \forall (a : A), \{(vcons\ \mathbb{N}\ a) : (\forall (v : vect\ A\ n), (vect\ A\ (S\ n)))\}^P \\
&\equiv \forall (a : A), \forall (v : vect\ A\ n), \{(vcons\ \mathbb{N}\ a\ v) : (vect\ A\ (S\ n))\}^P \\
&\equiv \forall (a : A), \forall (v : vect\ A\ n), P\ (S\ n)\ (vect\ A\ (S\ n)) \\
&\equiv \forall (a : A), \forall (v : vect\ A\ n), vect\ A\ ((S\ n) + m)
\end{aligned}
$$

Putting it all together

$$\frac{\begin{array}{l} E[\Gamma] \vdash c : (I\ q_1 ... q_r\ t_1 ... t_s) \\ E[\Gamma] \vdash P : B \\ [(I\ q_1 ... q_r)|B] \\ (E[\Gamma] \vdash f_i : \{(c_{p_i}\ q_1 ... q_r)\}^P)_{i=1...l} \end{array}}{E[\Gamma] \vdash \mathsf{case}(c, P, f_1|...|f_l) : (P\ t_1 ... t_s\ c)}$$

Putting it all together

params

bind to identifiers

$$\frac{\begin{array}{l} E[\Gamma] \vdash c : (I\ \overbrace{q_1 \ldots q_r}\ \underbrace{t_1 \ldots t_s}) \\ E[\Gamma] \vdash P : B \\ [(I\ q_1 \ldots q_r)|B] \\ (E[\Gamma] \vdash f_i : \{(c_{p_i}\ q_1 \ldots q_r)\}^P)_{i=1\ldots l} \end{array}}{E[\Gamma] \vdash \mathsf{case}(c, P, f_1|\ldots|f_l) : (P\ t_1 \ldots t_s\ c)}$$

args / indices

Putting it all together

params

bind to identifiers

$$\frac{\begin{array}{l} E[\Gamma] \vdash c : (I\ \overbrace{q_1 \ldots q_r}\ \underbrace{t_1 \ldots t_s}) \\ E[\Gamma] \vdash P : B \\ [(I\ q_1 \ldots q_r)|B] \\ (E[\Gamma] \vdash f_i : \{(c_{p_i}\ q_1 \ldots q_r)\}^P)_{i=1\ldots l} \end{array}}{E[\Gamma] \vdash \mathsf{case}(c, P, f_1|\ldots|f_l) : (P\ t_1 \ldots t_s\ c)}$$

args / indices

check P of the right form & sort for elimination

Putting it all together

$$\frac{\begin{array}{l} E[\Gamma] \vdash c : (I\ \underbrace{q_1 \ldots q_r}_{\text{params}}\ \underbrace{t_1 \ldots t_s}_{\text{args/indices}}) \\ E[\Gamma] \vdash P : B \\ [(I\ q_1 \ldots q_r)|B] \\ (E[\Gamma] \vdash f_i : \{(c_{p_i}\ q_1 \ldots q_r)\}^P)_{i=1\ldots l} \end{array}}{E[\Gamma] \vdash \mathsf{case}(c, P, f_1|\ldots|f_l) : (P\ t_1 \ldots t_s\ c)}$$

bind to identifiers

check P of the right form & sort for elimination

check all branches well-typed

Structure

Syntax

Concrete Syntax:

$$\text{fix } f_1(\Gamma_1) : A_1 := t_1 \text{ with} \ldots \text{with } f_n(\Gamma_n) : A_n := t_m \text{ for } f_i$$

Abstract Syntax:

$$\text{Fix } f_i\{f'_1 : A'_1 := t'_1 \ldots f'_n : A'_n := t'_n\}$$

The typing rule is the expected one for a fixpoint.

Normalization

- Non-normalizing terms allow proofs of absurdity
- Allow only primitive recursion[2]
- More precisely: "One of the arguments belongs to an inductive type, the function starts with a case analysis and recursive calls are done on variables coming from patterns representing subterms"

[2] well-founded recursion with `Program Fixpoint` tactic

Extended Abstract Syntax

$$\text{Fix } f_i\{f_1/k_1 : A_1 := t_1 \dots f_n/k_n : A_n := t_n\}$$

3

- side conditions: $A_i \leq_{\beta\delta\iota\zeta\eta} \forall y_1 : B_1, \dots \forall y_n : B_n$, $n \geq k_i$, B_{k_i} an inductive type.
- "In the definition t_i, if f_j occurs then the k_jth argument should be structurally smaller than y_{k_i}"
- read: The argument meant to be decreasing in a recursive call should be a (nested) subterm of the original argument

[3] it is still $f_i : A_i$, k_i is just an index

Recursive Arguments

$$\mathsf{Ind}[r](\Gamma_I := \Gamma_C),\ c : \forall p_1 : P_1, \ldots \forall p_r : P_r, \forall x_1 : T_1 \ldots \forall x_m : T_m, (I_j\ p_1 \ldots p_r\ t_1 \ldots t_s)$$

Recursive arguments: i, I_I occurs in T_i

Subterm

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

Source: https://coq.inria.fr/refman/language/gallina-specification-language.html#coq:cmd.fixpoint

"case$(c, P, f_1 \ldots_n)$ [...] c is y [...] the variables y_j occuring in g_i corresponding to recursive arguments B_i are structurally smaller than y"

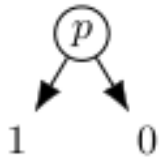
Deeper subterm

```
Fixpoint mod2 (n:nat) : nat :=
match n with
| O => O
| S p => match p with
         | O => S O
         | S q => mod2 q
         end
end.
```

Source: https://coq.inria.fr/refman/language/gallina-specification-language.html#coq:cmd.fixpoint

"case$(c, P, f_1 \ldots_n)$ [...] c is [...] structurally smaller than y [...] the variables y_j occuring in g_i corresponding to recursive arguments B_i are structurally smaller than y"

Compositionality

- $ROBDD(false) = 0$, $ROBDD(true) = 1$,
- $ROBDD(p) =$ 

- $ROBDD(\neg\phi) = ROBDD(\phi \rightarrow false)$
- $ROBDD(\phi \diamond \psi) =$ $apply(ROBDD(\phi), ROBDD(\psi), \diamond)$

no ROBDD called

Demo

Reduction

The expected one for fixpoints.

Structure