Programowanie z typami zależnymi i dowodzenie twierdzeń

Match i fix

24.03.2015

Funkcja pred

```
Print pred.
```

```
pred = fun n : nat ⇒ match n with
                     | 0 ⇒ 0
                     | S u ⇒ u
                     end
     : nat → nat
```

Funkcja pred_strong1

```
Lemma zgtz : 0 > 0 → False.
  crush.
Qed.

Definition pred_strong1 (n : nat) : n > 0 → nat :=
  match n with
    | O ⇒ fun pf : 0 > 0 ⇒ match zgtz pf with end
    | S n' ⇒ fun _ ⇒ n'
  end.
```

```
Theorem two_gt0 : 2 > 0.
  crush.
Qed.

Eval compute in pred_strong1 two_gt0.
```

```
     = 1
     : nat
```

Funkcja pred_strong1

```
Lemma zgtz : 0 > 0 → False.
  crush.
Qed.

Definition pred_strong1 (n : nat) : n > 0 → nat :=
  match n with
    | O ⇒ fun pf : 0 > 0 ⇒ match zgtz pf with end
    | S n' ⇒ fun _ ⇒ n'
  end.


Theorem two_gt0 : 2 > 0.
  crush.
Qed.

Eval compute in pred_strong1 two_gt0.
```

```
= 1
: nat
```

Funkcja pred_strong1'

```
Definition pred_strong1' (n : nat) (pf : n > 0) : nat :=
  match n with
    | O ⇒ match zgtz pf with end
    | S n' ⇒ n'
  end.
```

```
Error: In environment
n : nat
pf : n > 0
The term "pf" has type " n > 0" while it is expected to have
type "0 > 0"
```

Funkcja pred_strong1'

```
Definition pred_strong1' (n : nat) (pf : n > 0) : nat :=
  match n with
    | O ⇒ match zgtz pf with end
    | S n' ⇒ n'
  end.
```

```
Error: In environment
n : nat
pf : n > 0
The term "pf" has type " n > 0" while it is expected to have
type "0 > 0"
```

Funkcja pred_strong1''

```
Definition pred_strong1'' (n : nat) : n > 0 → nat :=
  match n return n > 0 → nat with
    | O ⇒ fun pf : 0 > 0 ⇒ match zgtz pf with end
    | S n' ⇒ fun _ ⇒ n'
  end.
```

Subset type

```
Print sig.
```

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
    exist : ∀ x : A, P x → sig P
```

```
Locate "{ _: _| _}".
```

```
  Notation
  "{ x : A | P }":= sig (fun x : A ⇒ P)
```

```
Parameter A is implicit
```

Subset type

```
Print sig.
```

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
    exist : ∀ x : A, P x → sig P
```

```
Locate "{ _: _| _}".
```

```
  Notation
  "{ x : A | P }":= sig (fun x : A ⇒ P)
```

Parameter A is *implicit*

Subset type

```
Print sig.
```

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
    exist : ∀ x : A, P x → sig P
```

```
Locate "{ _: _| _}".
```

```
  Notation
  "{ x : A | P }":= sig (fun x : A ⇒ P)
```

Parameter `A` is *implicit*

Funkcja pred_strong2

```
Definition pred_strong2 (s : {n : nat | n > 0}) : nat :=
  match s with
    | exist O pf ⇒ match zgtz pf with end
    | exist (S n') _ ⇒ n'
  end.
```

```
Eval compute in pred_strong2 (exist _ 2 two_gt0).
```

```
     = 1
     : nat
```

Funkcja pred_strong2

```
Definition pred_strong2 (s : {n : nat | n > 0}) : nat :=
  match s with
    | exist O pf ⇒ match zgtz pf with end
    | exist (S n') _ ⇒ n'
  end.
```

```
Eval compute in pred_strong2 (exist _ 2 two_gt0).
```

```
     = 1
     : nat
```

Funkcja pred_strong3

```
Definition pred_strong3 (s : {n : nat | n > 0}) : {m : nat | proj1_sig
s = S m} :=
  match s return {m : nat | proj1_sig s = S m} with
    | exist 0 pf ⇒ match zgtz pf with end
    | exist (S n') pf ⇒ exist _ n' (eq_refl _)
  end.
```

Listy z długością

```
Section ilist.
  Variable A : Set.

  Inductive ilist : nat → Set :=
  | Nil : ilist O
  | Cons : ∀ n, A → ilist n → ilist (S n).
```

Funkcja append na listach z długością

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2) :=
    match ls1 in (ilist n1) return (ilist (n1 + n2)) with
      | Nil ⇒ ls2
      | Cons _ x ls1' ⇒ Cons x (app' ls1' ls2)
    end.
```

Funkcja head na listach z długością cd.

```
Definition hd n (ls : ilist (S n)) : A :=
    match ls with
      | Cons _ h _ ⇒ h
    end.

Error: Non exhaustive pattern-matching: no clause found for
pattern Nil
```

W nowej wersji, przypadek `Nil` jest obsłużony przez Coqa

Funkcja head na listach z długością cd.

```
Definition hd n (ls : ilist (S n)) : A :=
    match ls with
      | Cons _ h _ ⇒ h
    end.
```

```
Error: Non exhaustive pattern-matching: no clause found for
pattern Nil
```

W nowej wersji, przypadek `Nil` jest obsłużony przez Coqa

Funkcja head na listach z długością cd.

```
Definition hd n (ls : ilist (S n)) : A :=
    match ls in (ilist (S n)) with
      | Cons _ h _ ⇒ h
    end.

Error: The reference n was not found in the current environment
```

Funkcja head na listach z długością cd.

```
  Definition hd' n (ls : ilist n) :=
    match ls in (ilist n) return (match n with O ⇒ unit | S _ ⇒ A
end) with
      | Nil ⇒ tt
      | Cons _ h _ ⇒ h
    end.

  Check hd'.

hd'
     : ∀ n : nat, ilist n → match n with
                            | 0 ⇒ unit
                            | S _ ⇒ A
                            end

  Definition hd n (ls : ilist (S n)) : A := hd' ls.

End ilist.
```

Destrukcja - match

Simple form:

$$\text{match } m \text{ with } (c_1\ x_{11}\ ...\ x_{1p_1}) \Rightarrow f_1 \mid \ \ldots \ \mid (c_n\ x_{n1}...x_{np_n}) \Rightarrow f_n \text{ end}$$

Full form:

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

For the purpose of presenting the inference rules, we use a more compact notation :

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

Destrukcja - match (c.d)

Type of branches. Let c be a term of type C, we assume C is a type of constructor for an inductive definition I. Let P be a term that represents the property to be proved. We assume r is the number of parameters. We define a new type $\{c : C\}^P$ which represents the type of the branch corresponding to the $c : C$ constructor.

$$\begin{aligned} \{c : (I_i\ p_1 \ldots p_r\ t_1 \ldots t_p)\}^P &\equiv (P\ t_1 \ldots\ t_p\ c) \\ \{c : \forall\ x : T, C\}^P &\equiv \forall\ x : T, \{(c\ x) : C\}^P \end{aligned}$$

We write $\{c\}^P$ for $\{c : C\}^P$ with C the type of c.

Examples. For List_A the type of P will be List_A $\rightarrow s$ for $s \in \mathcal{S}$.
$\{(\text{cons}\ A)\}^P \equiv \forall a : A, \forall l : \text{List\_A}, (P\ (\text{cons}\ A\ a\ l))$.

Destrukcja - match (c.d)

```
Inductive Length (A:Set) : list A -> nat -> Prop :=
| Lnil : Length A (nil A) O
| Lcons : forall (a:A) (l:list A) (n:nat),
                Length A l n -> Length A (cons A a l) (S n).
```

For Length_A, the type of P will be

$$\forall l : \text{List\_A}, \forall n : \text{nat}, (\text{Length\_A } l\ n) \to \text{Prop}$$

$$\{(\text{Lcons } A)\}^P = \forall a : A, \forall l : \text{List\_A}, \forall n : \text{nat}, \forall h : (\text{Length\_A } l\ n), \\ (P\ (\text{cons } A\ a\ l)\ (\text{S } n)\ (\text{Lcons } A\ a\ l\ n\ l)).$$

If P does not depend on its third argument:
$\{(\text{Lcons } A)\}^P =$
$\forall a : A, \forall l : \text{List\_A}, \forall n : \text{nat}, (\text{Length\_A } l\ n) \to (P\ (\text{cons } A\ a\ l)\ (\text{S } n))$.

Destrukcja - match (c.d)

```
Inductive Length (A:Set) : list A -> nat -> Prop :=
| Lnil : Length A (nil A) O
| Lcons : forall (a:A) (l:list A) (n:nat),
                Length A l n -> Length A (cons A a l) (S n).
```

For Length_A, the type of P will be

$$\forall l : \text{List\_A}, \forall n : \text{nat}, (\text{Length\_A } l\ n) \rightarrow \text{Prop}$$

$$\{(\text{Lcons } A)\}^P = \forall a : A, \forall l : \text{List\_A}, \forall n : \text{nat}, \forall h : (\text{Length\_A } l\ n), (P\ (\text{cons } A\ a\ l)\ (\text{S } n)\ (\text{Lcons } A\ a\ l\ n\ l)).$$

If P does not depend on its third argument:
$\{(\text{Lcons } A)\}^P =$
$\forall a : A, \forall l : \text{List\_A}, \forall n : \text{nat}, (\text{Length\_A } l\ n) \rightarrow (P\ (\text{cons } A\ a\ l)\ (\text{S } n))$.

Destrukcja - match (c.d)

```
Inductive Length (A:Set) : list A -> nat -> Prop :=
| Lnil : Length A (nil A) O
| Lcons : forall (a:A) (l:list A) (n:nat),
                Length A l n -> Length A (cons A a l) (S n).
```

For Length_A, the type of P will be

$$\forall l : \text{List\_A}, \forall n : \text{nat}, (\text{Length\_A } l\ n) \rightarrow \text{Prop}$$

$$\{(\text{Lcons } A)\}^P = \forall a : A, \forall l : \text{List\_A}, \forall n : \text{nat}, \forall h : (\text{Length\_A } l\ n), (P\ (\text{cons } A\ a\ l)\ (\text{S } n)\ (\text{Lcons } A\ a\ l\ n\ l)).$$

If P does not depend on its third argument:
$\{(\text{Lcons } A)\}^P =$
$\forall a : A, \forall l : \text{List\_A}, \forall n : \text{nat}, (\text{Length\_A } l\ n) \rightarrow (P\ (\text{cons } A\ a\ l)\ (\text{S } n))$.

Destrukcja - match (c.d)

```
Inductive Length (A:Set) : list A -> nat -> Prop :=
| Lnil : Length A (nil A) O
| Lcons : forall (a:A) (l:list A) (n:nat),
                Length A l n -> Length A (cons A a l) (S n).
```

For Length_A, the type of P will be

$$\forall l : \text{List\_A}, \forall n : \text{nat}, (\text{Length\_A } l\ n) \rightarrow \text{Prop}$$

$$\{(\text{Lcons } A)\}^P = \forall a : A, \forall l : \text{List\_A}, \forall n : \text{nat}, \forall h : (\text{Length\_A } l\ n), \\ (P\ (\text{cons } A\ a\ l)\ (\text{S } n)\ (\text{Lcons } A\ a\ l\ n\ l)).$$

If P does not depend on its third argument:
$\{(\text{Lcons } A)\}^P =$
$\forall a : A, \forall l : \text{List\_A}, \forall n : \text{nat}, (\text{Length\_A } l\ n) \rightarrow (P\ (\text{cons } A\ a\ l)\ (\text{S } n))$.

Allowed elimination sorts

`[I:A |B]` is the smallest relation satisfying the following rules:

- if `[(I x):A'|B']` then `[I:forall x:A, A'|forall x:A, B']`
- `[I:s1|I -> s2]` for any $s1 \in$ `Set`, `Type`(j), $s2 \in S$
- `[I:Prop|I -> Prop]`
- `[I:Prop|I -> s]` for `I` empty or singleton definition, $s \in S$

Allowed elimination sorts

`[I:A |B]` is the smallest relation satisfying the following rules:

- if `[(I x):A'|B']` then `[I:forall x:A, A'|forall x:A, B']`
- `[I:s1|I -> s2]` for any $s1 \in$ `Set`, `Type`(j), $s2 \in S$
- `[I:Prop|I -> Prop]`
- `[I:Prop|I -> s]` for `I` empty or singleton definition, $s \in S$

Allowed elimination sorts

`[I:A |B]` is the smallest relation satisfying the following rules:

- if `[(I x):A'|B']` then `[I:forall x:A, A'|forall x:A, B']`
- `[I:s1|I -> s2]` for any $s1 \in$ `Set`, `Type`(j), $s2 \in S$
- `[I:Prop|I -> Prop]`
- `[I:Prop|I -> s]` for `I` empty or singleton definition, $s \in S$

Allowed elimination sorts

`[I:A |B]` is the smallest relation satisfying the following rules:

- if `[(I x):A'|B']` then `[I:forall x:A, A'|forall x:A, B']`
- `[I:s1|I -> s2]` for any $s1 \in$ `Set`, `Type`(j), $s2 \in S$
- `[I:Prop|I -> Prop]`
- `[I:Prop|I -> s]` for `I` empty or singleton definition, $s \in S$

Destrukcja - match (c.d)

Typing rule.
Our very general destructor for inductive definition enjoys the following typing rule

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

provided I is an inductive type in a declaration $\mathsf{Ind}(\Delta)[r](\Gamma_I := \Gamma_C)$ with $\Gamma_C = [c_1 : C_1; \ldots; c_n : C_n]$ and $c_{p_1} \ldots c_{p_l}$ are the only constructors of I.

Destrukcja - match (redukcja)

A ι-reduction has the following form

$$\mathsf{case}((c_{p_i}\ q_1 \ldots q_r\ a_1 \ldots a_m), P, f_1 | \ldots | f_n) \triangleright_\iota (f_i\ a_1 \ldots a_m)$$

with c_{p_i} the i-th constructor of the inductive type I with r parameters.

Przykłady typowania - pred_strong1

```
Definition pred_strong1 (n : nat) : n > 0 → nat :=
  match n with
    | O ⇒ fun pf : 0 > 0 ⇒ match zgtz pf with end
    | S n' ⇒ fun _ ⇒ n'
  end.
```

$$P = fun\ n : nat\ \Rightarrow\ (n > 0) \rightarrow nat$$

Powinno być $f1 : P0$ i $f2 : \forall n' : nat, P(Sn')$, czyli

$$f1 : (0 > 0) \rightarrow nat$$

$$f2 : \forall\ n' : nat, (Sn' > 0) \rightarrow nat$$

Dobre będą:

$$f1 = fun\ pf : 0 > 0\ \Rightarrow\ False\_rec\ \_\ \_$$

$$f2 = fun\ (n' : nat)(pf : (Sn' > 0))\ \Rightarrow\ n'$$

Przykłady typowania - pred_strong1

```
Definition pred_strong1 (n : nat) : n > 0 → nat :=
  match n with
    | O ⇒ fun pf : 0 > 0 ⇒ match zgtz pf with end
    | S n' ⇒ fun _ ⇒ n'
  end.
```

$$P = fun\ n : nat\ \Rightarrow\ (n > 0) \rightarrow nat$$

Powinno być $f1 : P0$ i $f2 : \forall n' : nat, P(Sn')$, czyli

$$f1 : (0 > 0) \rightarrow nat$$

$$f2 : \forall\ n' : nat, (Sn' > 0) \rightarrow nat$$

Dobre będą:

$$f1 = fun\ pf : 0 > 0\ \Rightarrow\ False\_rec\ \_\ \_$$

$$f2 = fun\ (n' : nat)(pf : (Sn' > 0))\ \Rightarrow\ n'$$

Przykłady typowania - pred_strong1

```
Definition pred_strong1 (n : nat) : n > 0 → nat :=
  match n with
    | O ⇒ fun pf : 0 > 0 ⇒ match zgtz pf with end
    | S n' ⇒ fun _ ⇒ n'
  end.
```

$$P = fun\ n : nat\ \Rightarrow\ (n > 0) \rightarrow nat$$

Powinno być $f1 : P0$ i $f2 : \forall n' : nat, P(Sn')$, czyli

$$f1 : (0 > 0) \rightarrow nat$$

$$f2 : \forall\ n' : nat, (Sn' > 0) \rightarrow nat$$

Dobre będą:

$$f1 = fun\ pf : 0 > 0\ \Rightarrow\ False\_rec\ \_\ \_$$

$$f2 = fun\ (n' : nat)(pf : (Sn' > 0))\ \Rightarrow\ n'$$

Przykłady typowania - pred_strong1

```
Definition pred_strong1 (n : nat) : n > 0 → nat :=
  match n with
    | O ⇒ fun pf : 0 > 0 ⇒ match zgtz pf with end
    | S n' ⇒ fun _ ⇒ n'
  end.
```

$$P = fun\ n : nat\ \Rightarrow\ (n > 0) \rightarrow nat$$

Powinno być $f1 : P0$ i $f2 : \forall n' : nat, P(Sn')$, czyli

$$f1 : (0 > 0) \rightarrow nat$$

$$f2 : \forall\ n' : nat, (Sn' > 0) \rightarrow nat$$

Dobre będą:

$$f1 = fun\ pf : 0 > 0\ \Rightarrow\ False\_rec\ \_\ \_$$

$$f2 = fun\ (n' : nat)(pf : (Sn' > 0))\ \Rightarrow\ n'$$

Przykłady typowania - app

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2)
:=
    match ls1 in (ilist n1) return (ilist (n1 + n2)) with
      | Nil ⇒ ls2
      | Cons _ x ls1' ⇒ Cons x (app' ls1' ls2)
    end.
```

$$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow ilist(i + n2)$$

Powinno być $f1 : P\ 0\ Nil$ i
$f2 : \forall (n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$, czyli

$$f1 : ilist(0 + n2)$$

$$f2 : \forall\ (n' : nat)(a : A)(l : ilist\ n'), ilist(Sn')$$

Dobre będą:

$$f1 = ls2$$

$$f2 = fun\ (n' : nat)(a : A)(l : ilist\ n') \Rightarrow Cons\ n'\ a\ (app'\ l\ ls2)$$

Przykłady typowania - app

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2)
:=
    match ls1 in (ilist n1) return (ilist (n1 + n2)) with
      | Nil ⇒ ls2
      | Cons _ x ls1' ⇒ Cons x (app' ls1' ls2)
    end.
```

$$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow ilist(i + n2)$$

Powinno być $f1 : P\ 0\ Nil$ i
$f2 : \forall (n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$, czyli

$$f1 : ilist(0 + n2)$$

$$f2 : \forall\ (n' : nat)(a : A)(l : ilist\ n'), ilist(Sn')$$

Dobre będą:

$$f1 = ls2$$

$$f2 = fun\ (n' : nat)(a : A)(l : ilist\ n') \Rightarrow Cons\ n'\ a\ (app'\ l\ ls2)$$

Przykłady typowania - app

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2)
:=
    match ls1 in (ilist n1) return (ilist (n1 + n2)) with
      | Nil ⇒ ls2
      | Cons _ x ls1' ⇒ Cons x (app' ls1' ls2)
    end.
```

$$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow ilist(i + n2)$$

Powinno być $f1 : P\ 0\ Nil$ i
$f2 : \forall (n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$, czyli

$$f1 : ilist(0 + n2)$$

$$f2 : \forall\ (n' : nat)(a : A)(l : ilist\ n'), ilist(Sn')$$

Dobre będą:

$$f1 = ls2$$

$$f2 = fun\ (n' : nat)(a : A)(l : ilist\ n') \Rightarrow Cons\ n'\ a\ (app'\ l\ ls2)$$

Przykłady typowania - app

```
Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2)
:=
    match ls1 in (ilist n1) return (ilist (n1 + n2)) with
      | Nil ⇒ ls2
      | Cons _ x ls1' ⇒ Cons x (app' ls1' ls2)
    end.
```

$$P = fun\ (i : nat)(ls : ilist\ i) \Rightarrow ilist(i + n2)$$

Powinno być $f1 : P\ 0\ Nil$ i
$f2 : \forall (n' : nat)(a : A)(l : ilist\ n'), P\ (Sn')\ (Cons\ n'\ a\ l)$, czyli

$$f1 : ilist(0 + n2)$$

$$f2 : \forall\ (n' : nat)(a : A)(l : ilist\ n'), ilist(Sn')$$

Dobre będa:

$$f1 = ls2$$

$$f2 = fun\ (n' : nat)(a : A)(l : ilist\ n') \Rightarrow Cons\ n'\ a\ (app'\ l\ ls2)$$

Przykłady typowania - silna eliminacja

```
Definition sel (n:nat) := match n with
0 => False
| S _=> True
```

```
Goal 0=1 -> False.
intro H.
change (sel 1).
rewrite <- H.
red.
constructor.
Qed.
```

Przykłady typowania - silna eliminacja

```
Definition sel (n:nat) := match n with
0 => False
| S _=> True
```

```
Goal 0=1 -> False.
intro H.
change (sel 1).
rewrite <- H.
red.
constructor.
Qed.
```

Przykłady typowania - eliminacja równości

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist n): ilist m :=
match h in _=m with return (ilist m)
eq_refl => l
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist(m)$$

Powinno być $f\ :\ P\ n\ (eq\_refl\ nat\ n)$, czyli

$$f\ :\ ilist\ n$$

Dobra będzie:

$$f = l$$

- To była eliminacja z `Prop` w `Set` dla typu singletonowego
- Tak działa taktyka `rewrite`

Przykłady typowania - eliminacja równości

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist n): ilist m :=
match h in _=m with return (ilist m)
eq_refl => l
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist(m)$$

Powinno być $f : P\ n\ (eq\_refl\ nat\ n)$, czyli

$$f : ilist\ n$$

Dobra będzie:

$$f = l$$

- To była eliminacja z `Prop` w `Set` dla typu singletonowego
- Tak działa taktyka `rewrite`

Przykłady typowania - eliminacja równości

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist n): ilist m :=
match h in _=m with return (ilist m)
eq_refl => l
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist(m)$$

Powinno być $f\ :\ P\ n\ (eq\_refl\ nat\ n)$, czyli

$$f\ :\ ilist\ n$$

Dobra będzie:

$$f = l$$

- To była eliminacja z `Prop` w `Set` dla typu singletonowego
- Tak działa taktyka `rewrite`

Przykłady typowania - eliminacja równości

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist n): ilist m :=
match h in _=m with return (ilist m)
eq_refl => l
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist(m)$$

Powinno być $f\ :\ P\ n\ (eq\_refl\ nat\ n)$, czyli

$$f\ :\ ilist\ n$$

Dobra będzie:

$$f = l$$

- To była eliminacja z `Prop` w `Set` dla typu singletonowego
- Tak działa taktyka `rewrite`

Przykłady typowania - eliminacja równości

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist n): ilist m :=
match h in _=m with return (ilist m)
eq_refl => l
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist(m)$$

Powinno być $f\ :\ P\ n\ (eq\_refl\ nat\ n)$, czyli

$$f\ :\ ilist\ n$$

Dobra będzie:

$$f = l$$

- To była eliminacja z `Prop` w `Set` dla typu singletonowego
- Tak działa taktyka `rewrite`

Przykłady typowania - eliminacja równości

```
Definition rowne (n,m:nat)(h:n=m)(l:ilist n): ilist m :=
match h in _=m with return (ilist m)
eq_refl => l
```

$$P = fun\ (m : nat)(h : n = m) \Rightarrow ilist(m)$$

Powinno być $f\ :\ P\ n\ (eq\_refl\ nat\ n)$, czyli

$$f\ :\ ilist\ n$$

Dobra będzie:

$$f = l$$

- To była eliminacja z `Prop` w `Set` dla typu singletonowego
- Tak działa taktyka `rewrite`

Rekursja - fix

The basic concrete syntax for a recursive set of mutually recursive declarations is (with Γ_i contexts) :

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

In the inference rules, we represent such a term by

$$\texttt{Fix } f_i\{f_1 : A'_1 := t'_1 \ldots f_n : A'_n := t'_n\}$$

with t'_i (resp. A'_i) representing the term t_i abstracted (resp. generalized) with respect to the bindings in the context Γ_i, namely $t'_i = \lambda\Gamma_i, t_i$ and $A'_i = \forall\Gamma_i, A_i$.

Rekursja - fix (c.d)

Typing rule:

$$\frac{(E[\Gamma] \vdash A_i : s_i)_{i=1\ldots n} \quad (E[\Gamma, f_1 : A_1, \ldots, f_n : A_n] \vdash t_i : A_i)_{i=1\ldots n}}{E[\Gamma] \vdash \texttt{Fix}\ f_i\{f_1 : A_1 := t_1 \ldots f_n : A_n := t_n\} : A_i}$$

But we need to specify the argument on which we recurse.

For doing this the syntax of fixpoints is extended and becomes

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

where k_i are positive integers. Each A_i should be a type starting with at least k_i products $\forall y_1 : B_1, \ldots \forall y_{k_i} : B_{k_i}, A'_i$ and B_{k_i} being an instance of an inductive definition.
Now in the definition t_i, if f_j occurs then it should be applied to at least k_j arguments and the k_j-th argument should be syntactically recognized as *structurally smaller* than y_{k_i}

Rekursja - fix (c.d)

Given a variable y of an inductive type from a declaration $\mathsf{Ind}(\Gamma)[r](\Gamma_I := \Gamma_C)$ where Γ_I is $[I_1 : A_1; \ldots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \ldots; c_n : C_n]$. The terms structurally smaller than y are:

- $(t\ u), \lambda x : u, t$ when t is structurally smaller than y .
- $\mathsf{case}(c, P, f_1 \ldots f_n)$ when each f_i is structurally smaller than y. If c is y or is structurally smaller than y, its type is an inductive definition I_p part of the inductive declaration corresponding to y. Each f_i corresponds to a type of constructor $C_q \equiv \forall p_1 : P_1, \ldots, \forall p_r : P_r, \forall y_1 : B_1, \ldots \forall y_k : B_k, (I\ a_1 \ldots a_k)$ and can consequently be written $\lambda y_1 : B'_1, \ldots \lambda y_k : B'_k, g_i$. ($B'_i$ is obtained from B_i by substituting parameters variables) the variables y_j occurring in g_i corresponding to recursive arguments B_i (the ones in which one of the I_l occurs) are structurally smaller than y.

Rekursja - fix (redukcja)

Let F be the set of declarations: $f_1/k_1 : A_1 := t_1 \ldots f_n/k_n : A_n := t_n$.
The reduction for fixpoints is:

$$(\texttt{Fix}\ f_i\{F\}\ a_1 \ldots a_{k_i}) \triangleright_\iota t_i\{(f_k/\texttt{Fix}\ f_k\{F\})_{k=1\ldots n}\}\ a_1 \ldots a_{k_i}$$

when a_{k_i} starts with a constructor.
This last restriction is needed in order to keep strong normalization and corresponds to the reduction for primitive recursive operators.

Przykład strukturalnej rekursji

```
Fixpoint split (ls : list A) : list A × list A :=
  match ls with
    | nil ⇒ (nil, nil)
    | h :: nil ⇒ (h :: nil, nil)
    | h1 :: h2 :: ls' ⇒
      let (ls1, ls2) := split ls' in
        (h1 :: ls1, h2 :: ls2)
  end.
```

Przykład niestrukturalnej rekursji

```
Fixpoint mergeSort (ls : list A) : list A :=
  if leb (length ls) 1
    then ls
    else let lss := split ls in
      merge (mergeSort (fst lss)) (mergeSort (snd lss)).
```

```
Recursive call to mergeSort has principal argument equal to
"fst (split ls)" instead of a subterm of " ls".
```