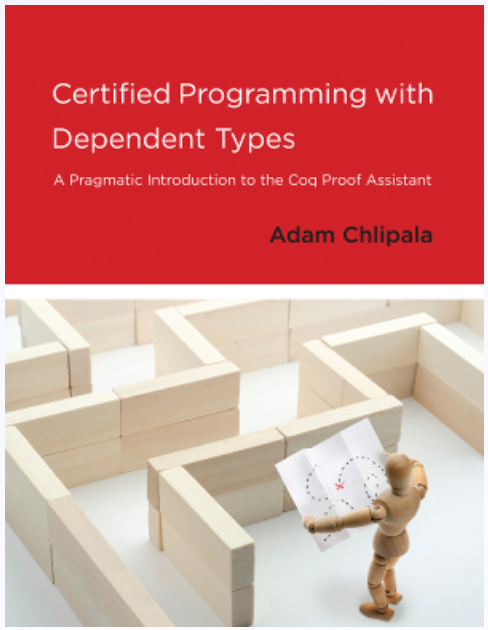
Dowodzenie twierdzeń i programowanie z typami zależnymi w Coqu

Zależny pattren-matching

12 kwietnia 2016

Przykład użycia typów zależnych

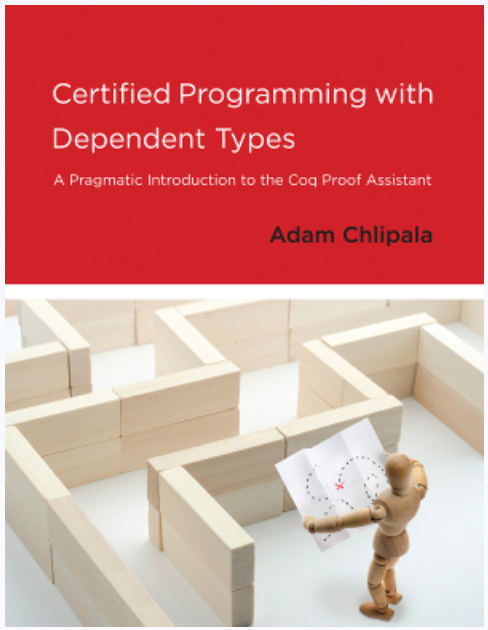
Poprawność kompilacji — przykład z książki Certified Programming with Dependent Types, Adam Chlipala (MIT)



- wydawnictwo: MIT Press 2013
- dostępne: `http://adam.chlipala.net/cpdt/`

Przykład użycia typów zależnych

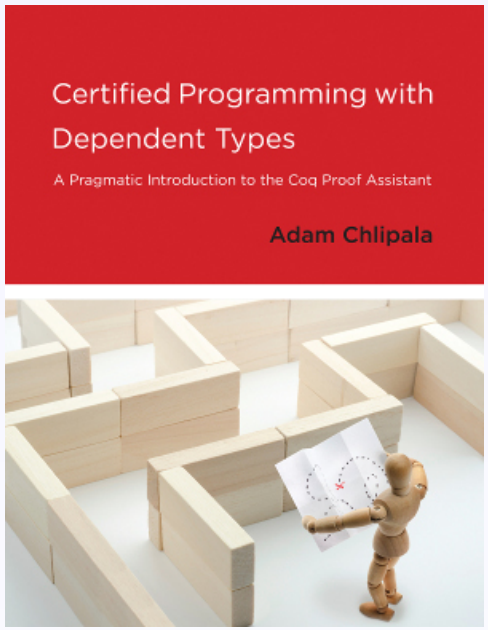
Poprawność kompilacji — przykład z książki Certified Programming with Dependent Types, Adam Chlipala (MIT)



- wydawnictwo: MIT Press 2013
- dostępne: `http://adam.chlipala.net/cpdt/`

Przykład użycia typów zależnych

Poprawność kompilacji — przykład z książki Certified Programming with Dependent Types, Adam Chlipala (MIT)



- wydawnictwo: MIT Press 2013
- dostępne: `http://adam.chlipala.net/cpdt/`

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language denotation

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Definition binopDenote (b : binop) : Z → Z → Z :=
  match b with
  | Plus ⇒ fun x y ⇒ x + y
  | Times ⇒ fun x y ⇒ x × y
  end.

Fixpoint expDenote (e : exp) : Z :=
  match e with
  | Const n ⇒ n
  | Binop b e1 e2 ⇒ (binopDenote b) (expDenote e1) (expDenote e2)
  end.
```

Single sort: Source language denotation

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Definition binopDenote (b : binop) : Z → Z → Z :=
  match b with
  | Plus ⇒ fun x y ⇒ x + y
  | Times ⇒ fun x y ⇒ x × y
  end.

Fixpoint expDenote (e : exp) : Z :=
  match e with
  | Const n ⇒ n
  | Binop b e1 e2 ⇒ (binopDenote b) (expDenote e1) (expDenote e2)
  end.
```

Single sort: Source language denotation

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Definition binopDenote (b : binop) : Z → Z → Z :=
  match b with
  | Plus ⇒ fun x y ⇒ x + y
  | Times ⇒ fun x y ⇒ x × y
  end.

Fixpoint expDenote (e : exp) : Z :=
  match e with
  | Const n ⇒ n
  | Binop b e1 e2 ⇒ (binopDenote b) (expDenote e1) (expDenote e2)
  end.
```

Single sort: Target language

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Check iConst 42 :: nil.

Check iConst 2 :: iConst 2 :: iBinop Plus :: nil.

Check iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil.
```

Single sort: Target language

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Check iConst 42 :: nil.

Check iConst 2 :: iConst 2 :: iBinop Plus :: nil.

Check iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil.
```

Single sort: Target language

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Check iConst 42 :: nil.

Check iConst 2 :: iConst 2 :: iBinop Plus :: nil.

Check iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil.
```

Single sort: Target language

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Check iConst 42 :: nil.

Check iConst 2 :: iConst 2 :: iBinop Plus :: nil.

Check iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil.
```

Single sort: Target language

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Check iConst 42 :: nil.

Check iConst 2 :: iConst 2 :: iBinop Plus :: nil.

Check iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil.
```

Single sort: Target language denotation

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Definition stack := list Z.

Definition instrDenote (i : instr) (s : stack) : option stack :=
  match i with
  | iConst n ⇒ Some (n :: s)
  | iBinop b ⇒
    match s with
    | arg1 :: arg2 :: s' ⇒ Some ((binopDenote b) arg1 arg2 :: s')
    | _ ⇒ None
    end
  end.
```

Single sort: Target language denotation

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Definition stack := list Z.

Definition instrDenote (i : instr) (s : stack) : option stack :=
  match i with
  | iConst n ⇒ Some (n :: s)
  | iBinop b ⇒
    match s with
    | arg1 :: arg2 :: s' ⇒ Some ((binopDenote b) arg1 arg2 :: s')
    | _ ⇒ None
    end
  end.
```

Single sort: Target language denotation

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Definition stack := list Z.

Definition instrDenote (i : instr) (s : stack) : option stack :=
  match i with
  | iConst n ⇒ Some (n :: s)
  | iBinop b ⇒
    match s with
    | arg1 :: arg2 :: s' ⇒ Some ((binopDenote b) arg1 arg2 :: s')
    | _ ⇒ None
    end
  end.
```

Single sort: Target language denotation

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Definition stack := list Z.

Definition instrDenote (i : instr) (s : stack) : option stack := ...
```

Single sort: Target language denotation

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Definition stack := list Z.

Definition instrDenote (i : instr) (s : stack) : option stack := ...

Fixpoint progDenote (p : prog) (s : stack) : option stack :=
  match p with
    | nil ⇒ Some s
    | i :: p' ⇒ match instrDenote i s with
                  | None ⇒ None
                  | Some s' ⇒ progDenote p' s'
                end
  end.
```

Single sort: Compilation

```
Fixpoint compile (e : exp) : prog :=
  match e with
  | Const n ⇒ iConst n :: nil
  | Binop b e1 e2 ⇒ compile e2 ++ compile e1 ++ iBinop b :: nil
  end.
```

Single sort: Compilation correctness

```
Fixpoint compile (e : exp) : prog :=
  match e with
  | Const n ⇒ iConst n :: nil
  | Binop b e1 e2 ⇒ compile e2 ++ compile e1 ++ iBinop b :: nil
  end.
```

```
Theorem compile_correct :
  ∀ e, progDenote (compile e) nil = Some (expDenote e :: nil).
```

```
Lemma compile_correct' : ∀ e p s,
  progDenote (compile e ++ p) s = progDenote p (expDenote e :: s).
```

Single sort: Compilation correctness

```
Fixpoint compile (e : exp) : prog :=
  match e with
  | Const n ⇒ iConst n :: nil
  | Binop b e1 e2 ⇒ compile e2 ++ compile e1 ++ iBinop b :: nil
  end.
```

```
Theorem compile_correct :
  ∀ e, progDenote (compile e) nil = Some (expDenote e :: nil).
```

```
Lemma compile_correct' : ∀ e p s,
  progDenote (compile e ++ p) s = progDenote p (expDenote e :: s).
```

Single sort: Compilation correctness

```
Fixpoint compile (e : exp) : prog :=
  match e with
  | Const n ⇒ iConst n :: nil
  | Binop b e1 e2 ⇒ compile e2 ++ compile e1 ++ iBinop b :: nil
  end.
```

```
Theorem compile_correct :
  ∀ e, progDenote (compile e) nil = Some (expDenote e :: nil).
```

```
Lemma compile_correct' : ∀ e p s,
  progDenote (compile e ++ p) s = progDenote p (expDenote e :: s).
```

Many sorts: Source language

```
Inductive sort : Set := Mint | Mbool.

Inductive mbinop : sort → sort → sort → Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : ∀ s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.

Inductive mexp : sort → Set :=
| MZConst : Z → mexp Mint
| MBConst : bool → mexp Mbool
| MBinop : ∀ s1 s2 s, mbinop s1 s2 s → mexp s1 → mexp s2 → mexp s.

Check MBinop MPlus (MZConst 2) (MZConst 2) : mexp Mint.

Check MBinop (MEq Mint) (MBinop MPlus (MZConst 2) (MZConst 2))
                        (MZConst 7) : mexp Mbool.
```

Many sorts: Source language

```
Inductive sort : Set := Mint | Mbool.

Inductive mbinop : sort → sort → sort → Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : ∀ s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.

Inductive mexp : sort → Set :=
| MZConst : Z → mexp Mint
| MBConst : bool → mexp Mbool
| MBinop : ∀ s1 s2 s, mbinop s1 s2 s → mexp s1 → mexp s2 → mexp s.

Check MBinop MPlus (MZConst 2) (MZConst 2) : mexp Mint.

Check MBinop (MEq Mint) (MBinop MPlus (MZConst 2) (MZConst 2))
                        (MZConst 7) : mexp Mbool.
```

Many sorts: Source language

```
Inductive sort : Set := Mint | Mbool.

Inductive mbinop : sort → sort → sort → Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : ∀ s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.

Inductive mexp : sort → Set :=
| MZConst : Z → mexp Mint
| MBConst : bool → mexp Mbool
| MBinop : ∀ s1 s2 s, mbinop s1 s2 s → mexp s1 → mexp s2 → mexp s.

Check MBinop MPlus (MZConst 2) (MZConst 2) : mexp Mint.

Check MBinop (MEq Mint) (MBinop MPlus (MZConst 2) (MZConst 2))
                        (MZConst 7) : mexp Mbool.
```

Many sorts: Source language

```
Inductive sort : Set := Mint | Mbool.

Inductive mbinop : sort → sort → sort → Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : ∀ s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.

Inductive mexp : sort → Set :=
| MZConst : Z → mexp Mint
| MBConst : bool → mexp Mbool
| MBinop : ∀ s1 s2 s, mbinop s1 s2 s → mexp s1 → mexp s2 → mexp s.

Check MBinop MPlus (MZConst 2) (MZConst 2) : mexp Mint.

Check MBinop (MEq Mint) (MBinop MPlus (MZConst 2) (MZConst 2))
             (MZConst 7) : mexp Mbool.
```

Many sorts: Source language

```
Inductive sort : Set := Mint | Mbool.

Inductive mbinop : sort → sort → sort → Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : ∀ s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.

Inductive mexp : sort → Set :=
| MZConst : Z → mexp Mint
| MBConst : bool → mexp Mbool
| MBinop : ∀ s1 s2 s, mbinop s1 s2 s → mexp s1 → mexp s2 → mexp s.

Check MBinop MPlus (MZConst 2) (MZConst 2) : mexp Mint.

Check MBinop (MEq Mint) (MBinop MPlus (MZConst 2) (MZConst 2))
                        (MZConst 7) : mexp Mbool.
```

Many sorts: Source language denotation

```
Definition sortDenote (s : sort) : Set :=
  match s with
    | Mint ⇒ Z
    | Mbool ⇒ bool
  end.

Definition mbinopDenote arg1 arg2 res (b : mbinop arg1 arg2 res)
  : sortDenote arg1 → sortDenote arg2 → sortDenote res :=
  match b with
    | MPlus ⇒ Z.add
    | MTimes ⇒ Z.mul
    | MEq Mint ⇒ Z.eqb
    | MEq Mbool ⇒ Bool.eqb
    | MLt ⇒ Z.ltb
  end.

Fixpoint mexpDenote s (e : mexp s) : sortDenote s :=
  match e with
    | MZConst n ⇒ n
    | MBConst b ⇒ b
    | MBinop _ _ b e1 e2 ⇒
```

Many sorts: Source language denotation

```
Definition sortDenote (s : sort) : Set :=
  match s with Mint ⇒ Z | Mbool ⇒ bool end.

Definition mbinopDenote arg1 arg2 res (b : mbinop arg1 arg2 res)
  : sortDenote arg1 → sortDenote arg2 → sortDenote res :=
  match b with
    | MPlus ⇒ Z.add
    | MTimes ⇒ Z.mul
    | MEq Mint ⇒ Z.eqb
    | MEq Mbool ⇒ Bool.eqb
    | MLt ⇒ Z.ltb
  end.

Fixpoint mexpDenote s (e : mexp s) : sortDenote s :=
  match e with
    | MZConst n ⇒ n
    | MBConst b ⇒ b
    | MBinop _ _ _ b e1 e2 ⇒
        (mbinopDenote b) (mexpDenote e1) (mexpDenote e2)
  end.
```

Many sorts: Source language denotation

```
Definition sortDenote (s : sort) : Set :=
  match s with Mint ⇒ Z | Mbool ⇒ bool end.

Definition mbinopDenote arg1 arg2 res (b : mbinop arg1 arg2 res)
  : sortDenote arg1 → sortDenote arg2 → sortDenote res :=
  match b with
    | MPlus ⇒ Z.add
    | MTimes ⇒ Z.mul
    | MEq Mint ⇒ Z.eqb
    | MEq Mbool ⇒ Bool.eqb
    | MLt ⇒ Z.ltb
  end.

Fixpoint mexpDenote s (e : mexp s) : sortDenote s :=
  match e with
    | MZConst n ⇒ n
    | MBConst b ⇒ b
    | MBinop _ _ _ b e1 e2 ⇒
        (mbinopDenote b) (mexpDenote e1) (mexpDenote e2)
  end.
```

Many sorts: Source language denotation

```
Definition sortDenote (s : sort) : Set :=
  match s with Mint ⇒ Z | Mbool ⇒ bool end.

Definition mbinopDenote arg1 arg2 res (b : mbinop arg1 arg2 res)
  : sortDenote arg1 → sortDenote arg2 → sortDenote res :=
  match b with
    | MPlus ⇒ Z.add
    | MTimes ⇒ Z.mul
    | MEq Mint ⇒ Z.eqb
    | MEq Mbool ⇒ Bool.eqb
    | MLt ⇒ Z.ltb
  end.

Fixpoint mexpDenote s (e : mexp s) : sortDenote s :=
  match e with
    | MZConst n ⇒ n
    | MBConst b ⇒ b
    | MBinop _ _ _ b e1 e2 ⇒
        (mbinopDenote b) (mexpDenote e1) (mexpDenote e2)
  end.
```

Many sorts: Target language

```
Definition sstack := list sort.

Inductive minstr : sstack → sstack → Set :=
| MiZConst : ∀ ss, Z → minstr ss (Mint :: ss)
| MiBConst : ∀ ss, bool → minstr ss (Mbool :: ss)
| MiBinop : ∀ arg1 arg2 res ss,
    mbinop arg1 arg2 res → minstr (arg1 :: arg2 :: ss) (res :: ss).

Inductive mprog : sstack → sstack → Set :=
| MNil : ∀ ss, mprog ss ss
| MCons : ∀ ss1 ss2 ss3, minstr ss1 ss2 → mprog ss2 ss3 → mprog ss1 ss3.
```

Many sorts: Target language

```
Definition sstack := list sort.

Inductive minstr : sstack → sstack → Set :=
| MiZConst : ∀ ss, Z → minstr ss (Mint :: ss)
| MiBConst : ∀ ss, bool → minstr ss (Mbool :: ss)
| MiBinop : ∀ arg1 arg2 res ss,
   mbinop arg1 arg2 res → minstr (arg1 :: arg2 :: ss) (res :: ss).

Inductive mprog : sstack → sstack → Set :=
| MNil : ∀ ss, mprog ss ss
| MCons : ∀ ss1 ss2 ss3, minstr ss1 ss2 → mprog ss2 ss3 → mprog ss1 ss3.
```

Many sorts: Target language

```
Definition sstack := list sort.

Inductive minstr : sstack → sstack → Set :=
| MiZConst : ∀ ss, Z → minstr ss (Mint :: ss)
| MiBConst : ∀ ss, bool → minstr ss (Mbool :: ss)
| MiBinop : ∀ arg1 arg2 res ss,
   mbinop arg1 arg2 res → minstr (arg1 :: arg2 :: ss) (res :: ss).

Inductive mprog : sstack → sstack → Set :=
| MNil : ∀ ss, mprog ss ss
| MCons : ∀ ss1 ss2 ss3, minstr ss1 ss2 → mprog ss2 ss3 → mprog ss1 ss3.
```

Many sorts: Target language denotation

```
Fixpoint vstack (ss : sstack) : Set :=
  match ss with
    | nil ⇒ unit
    | s :: ss' ⇒ sortDenote s × vstack ss'
  end.

Check (5, (true, (false, ()))) : vstack (Mint::Mbool::Mbool::nil).

Definition minstrDenote ss ss' (i : minstr ss ss') : vstack ss → vstack ss' :=
  match i with
    | MiZConst _ n ⇒ fun vs ⇒ (n, vs)
    | MiBConst _ b ⇒ fun vs ⇒ (b, vs)
    | MiBinop _ _ _ _ b ⇒ fun vs ⇒
      let '(v1, (v2, vs')) := vs in ((mbinopDenote b) v1 v2, vs')
  end.

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss → vstack ss' :=
  match p with
    | MNil _ ⇒ fun vs ⇒ vs
    | MCons _ _ _ i p' ⇒ fun vs ⇒ mprogDenote p' (minstrDenote i vs)
  end.
```

Many sorts: Target language denotation

```
Fixpoint vstack (ss : sstack) : Set :=
  match ss with
    | nil ⇒ unit
    | s :: ss' ⇒ sortDenote s × vstack ss'
  end.

Check (5, (true, (false, ()))) : vstack (Mint::Mbool::Mbool::nil).

Definition minstrDenote ss ss' (i : minstr ss ss') : vstack ss → vstack ss' :=
  match i with
    | MiZConst _ n ⇒ fun vs ⇒ (n, vs)
    | MiBConst _ b ⇒ fun vs ⇒ (b, vs)
    | MiBinop _ _ _ _ b ⇒ fun vs ⇒
      let '(v1, (v2, vs')) := vs in ((mbinopDenote b) v1 v2, vs')
  end.

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss → vstack ss' :=
  match p with
    | MNil _ ⇒ fun vs ⇒ vs
    | MCons _ _ _ i p' ⇒ fun vs ⇒ mprogDenote p' (minstrDenote i vs)
  end.
```

Many sorts: Target language denotation

```
Fixpoint vstack (ss : sstack) : Set :=
  match ss with nil ⇒ unit | s :: ss' ⇒ sortDenote s × vstack ss' end.

Check (5, (true, (false, ()))) : vstack (Mint::Mbool::Mbool::nil).

Definition minstrDenote ss ss' (i : minstr ss ss') : vstack ss → vstack ss' :=
  match i with
    | MiZConst _ n ⇒ fun vs ⇒ (n, vs)
    | MiBConst _ b ⇒ fun vs ⇒ (b, vs)
    | MiBinop _ _ _ _ b ⇒ fun vs ⇒
      let '(v1, (v2, vs')) := vs in ((mbinopDenote b) v1 v2, vs')
  end.

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss → vstack ss' :=
  match p with
    | MNil _ ⇒ fun vs ⇒ vs
    | MCons _ _ _ i p' ⇒ fun vs ⇒ mprogDenote p' (minstrDenote i vs)
  end.
```

Many sorts: Target language denotation

```
Fixpoint vstack (ss : sstack) : Set :=
  match ss with nil ⇒ unit | s :: ss' ⇒ sortDenote s × vstack ss' end.

Check (5, (true, (false, ()))) : vstack (Mint::Mbool::Mbool::nil).

Definition minstrDenote ss ss' (i : minstr ss ss') : vstack ss → vstack ss' :=
  match i with
    | MiZConst _ n ⇒ fun vs ⇒ (n, vs)
    | MiBConst _ b ⇒ fun vs ⇒ (b, vs)
    | MiBinop _ _ _ _ b ⇒ fun vs ⇒
      let '(v1, (v2, vs')) := vs in ((mbinopDenote b) v1 v2, vs')
  end.

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss → vstack ss' :=
  match p with
    | MNil _ ⇒ fun vs ⇒ vs
    | MCons _ _ _ i p' ⇒ fun vs ⇒ mprogDenote p' (minstrDenote i vs)
  end.
```

Many sorts: Target language denotation

```
Fixpoint vstack (ss : sstack) : Set :=
  match ss with nil ⇒ unit | s :: ss' ⇒ sortDenote s × vstack ss' end.

Check (5, (true, (false, ()))) : vstack (Mint::Mbool::Mbool::nil).

Definition minstrDenote ss ss' (i : minstr ss ss') : vstack ss → vstack ss' :=
  match i with
    | MiZConst _ n ⇒ fun vs ⇒ (n, vs)
    | MiBConst _ b ⇒ fun vs ⇒ (b, vs)
    | MiBinop _ _ _ _ b ⇒ fun vs ⇒
      let '(v1, (v2, vs')) := vs in ((mbinopDenote b) v1 v2, vs')
  end.

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss → vstack ss' :=
  match p with
    | MNil _ ⇒ fun vs ⇒ vs
    | MCons _ _ _ i p' ⇒ fun vs ⇒ mprogDenote p' (minstrDenote i vs)
  end.
```

Many sorts: Compilation

```
Fixpoint mcompile s (e : mexp s) (ss : sstack) : mprog ss (s :: ss) :=
  match e with
    | MZConst n ⇒ MCons (MiZConst _ n) (MNil _)
    | MBConst b ⇒ MCons (MiBConst _ b) (MNil _)
    | MBinop _ _ _ b e1 e2 ⇒ mconcat (mcompile e2 _)
      (mconcat (mcompile e1 _) (MCons (MiBinop _ b) (MNil _)))
  end.
```

Many sorts: Compilation correctness

```
Fixpoint mcompile s (e : mexp s) (ss : sstack) : mprog ss (s :: ss) :=
  match e with
    | MZConst n ⇒ MCons (MiZConst _ n) (MNil _)
    | MBConst b ⇒ MCons (MiBConst _ b) (MNil _)
    | MBinop _ _ _ b e1 e2 ⇒ mconcat (mcompile e2 _)
      (mconcat (mcompile e1 _) (MCons (MiBinop _ b) (MNil _)))
  end.

Theorem mcompile_correct : ∀ s (e : mexp s),
  mprogDenote (mcompile e nil) () = (mexpDenote e, ()).

Lemma mcompile_correct' : ∀ s (e : mexp s) ss (s : vstack ss),
  mprogDenote (mcompile e ss) s = (mexpDenote e, s).
```

Many sorts: Compilation correctness

```
Fixpoint mcompile s (e : mexp s) (ss : sstack) : mprog ss (s :: ss) :=
  match e with
    | MZConst n ⇒ MCons (MiZConst _ n) (MNil _)
    | MBConst b ⇒ MCons (MiBConst _ b) (MNil _)
    | MBinop _ _ _ b e1 e2 ⇒ mconcat (mcompile e2 _)
      (mconcat (mcompile e1 _) (MCons (MiBinop _ b) (MNil _)))
  end.

Theorem mcompile_correct : ∀ s (e : mexp s),
  mprogDenote (mcompile e nil) () = (mexpDenote e, ()).

Lemma mcompile_correct' : ∀ s (e : mexp s) ss (s : vstack ss),
  mprogDenote (mcompile e ss) s = (mexpDenote e, s).
```

Many sorts: Compilation correctness

```
Fixpoint mcompile s (e : mexp s) (ss : sstack) : mprog ss (s :: ss) :=
  match e with
    | MZConst n ⇒ MCons (MiZConst _ n) (MNil _)
    | MBConst b ⇒ MCons (MiBConst _ b) (MNil _)
    | MBinop _ _ _ b e1 e2 ⇒ mconcat (mcompile e2 _)
      (mconcat (mcompile e1 _) (MCons (MiBinop _ b) (MNil _)))
  end.
```

```
Theorem mcompile_correct : ∀ s (e : mexp s),
  mprogDenote (mcompile e nil) () = (mexpDenote e, ()).
```

```
Lemma mcompile_correct' : ∀ s (e : mexp s) ss (s : vstack ss),
  mprogDenote (mcompile e ss) s = (mexpDenote e, s).
```

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.
  intro H.
  inversion H.
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.
  auto.
Qed.

Eval compute in pred_strong1 two_gt0.

     = 1
     : nat
```

Funkcja pred_strong1

```
Lemma zgtz : 0 > 0 → False.
  intro H.
  inversion H.
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.
  auto.
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.
```

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
```

Od jakiegoś czasu, 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
```

Od jakiegoś czasu, 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:

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

Full form:

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

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

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

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 $s1 \in \mathtt{Set}, \mathtt{Type}(j)$, $s2 \in S$
- `[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 będa:

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

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

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`