

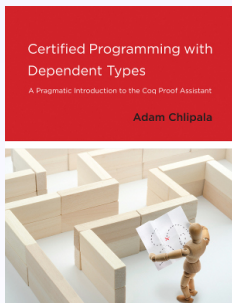
Dependent pattern-matching in Coq

Daria Walukiewicz-Chrząszcz

5 April 2022

Dependent types in programming

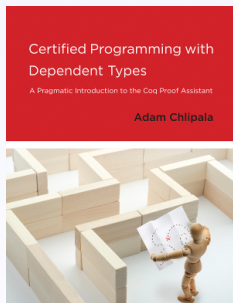
Compilation correctness — example from “Certified Programming with Dependent Types” by Adam Chlipala (MIT)



- available: <http://adam.chlipala.net/cpdt/>

Dependent types in programming

Compilation correctness — example from “Certified Programming with Dependent Types” by Adam Chlipala (MIT)



- available: <http://adam.chlipala.net/cpdt/>

Single sort: Source language

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

Inductive *exp* : Set :=

| *Const* : $\mathbb{Z} \rightarrow \text{exp}$

| *Binop* : *binop* $\rightarrow \text{exp} \rightarrow \text{exp} \rightarrow \text{exp}$.

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 $binop : Set := Plus \mid Times.$

Inductive $exp : Set :=$

| $Const : Z \rightarrow exp$

| $Binop : binop \rightarrow exp \rightarrow exp \rightarrow exp.$

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 $binop : Set := Plus \mid Times.$

Inductive $exp : Set :=$

| $Const : Z \rightarrow exp$

| $Binop : binop \rightarrow exp \rightarrow exp \rightarrow exp.$

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 $binop : Set := Plus \mid Times$.

Inductive $exp : Set :=$

| $Const : Z \rightarrow exp$

| $Binop : binop \rightarrow exp \rightarrow exp \rightarrow exp$.

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 $binop : Set := Plus \mid Times$.

Inductive $exp : Set :=$

| $Const : Z \rightarrow exp$

| $Binop : binop \rightarrow exp \rightarrow exp \rightarrow exp$.

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 $binop : Set := Plus \mid Times.$

Inductive $exp : Set :=$

| $Const : Z \rightarrow exp$

| $Binop : binop \rightarrow exp \rightarrow exp \rightarrow exp.$

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 $binop : Set := Plus \mid Times$.

Inductive $exp : Set :=$

| $Const : Z \rightarrow exp$

| $Binop : binop \rightarrow exp \rightarrow exp \rightarrow exp$.

Definition $binopDenote (b : binop) : Z \rightarrow Z \rightarrow Z :=$

match b with

| $Plus \Rightarrow \text{fun } x \ y \Rightarrow x + y$

| $Times \Rightarrow \text{fun } x \ y \Rightarrow x \times y$

end.

Fixpoint $expDenote (e : exp) : Z :=$

match e with

| $Const \ n \Rightarrow n$

| $Binop \ b \ e1 \ e2 \Rightarrow (binopDenote \ b) (expDenote \ e1) (expDenote \ e2)$

end.

Single sort: Source language denotation

Inductive $binop : Set := Plus \mid Times.$

Inductive $exp : Set :=$

| $Const : Z \rightarrow exp$

| $Binop : binop \rightarrow exp \rightarrow exp \rightarrow exp.$

Definition $binopDenote (b : binop) : Z \rightarrow Z \rightarrow Z :=$

match b with

| $Plus \Rightarrow \text{fun } x \ y \Rightarrow x + y$

| $Times \Rightarrow \text{fun } x \ y \Rightarrow x \times y$

end.

Fixpoint $expDenote (e : exp) : Z :=$

match e with

| $Const \ n \Rightarrow n$

| $Binop \ b \ e1 \ e2 \Rightarrow (binopDenote \ b) (expDenote \ e1) (expDenote \ e2)$

end.

Single sort: Source language denotation

Inductive $binop : Set := Plus \mid Times$.

Inductive $exp : Set :=$

| $Const : Z \rightarrow exp$

| $Binop : binop \rightarrow exp \rightarrow exp \rightarrow exp$.

Definition $binopDenote (b : binop) : Z \rightarrow Z \rightarrow Z :=$

match b with

| $Plus \Rightarrow \text{fun } x \ y \Rightarrow x + y$

| $Times \Rightarrow \text{fun } x \ y \Rightarrow x \times y$

end.

Fixpoint $expDenote (e : exp) : Z :=$

match e with

| $Const \ n \Rightarrow n$

| $Binop \ b \ e1 \ e2 \Rightarrow (binopDenote \ b) (expDenote \ e1) (expDenote \ e2)$

end.

Single sort: Target language

Inductive *instr* : Set :=
| *iConst* : $Z \rightarrow instr$
| *iBinop* : *binop* $\rightarrow instr$.

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* : $Z \rightarrow instr$
| *iBinop* : *binop* $\rightarrow instr$.

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* : $Z \rightarrow instr$
| *iBinop* : *binop* $\rightarrow instr$.

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* : $Z \rightarrow instr$
| *iBinop* : *binop* $\rightarrow instr$.

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 $instr : Set :=$
| $iConst : Z \rightarrow instr$
| $iBinop : binop \rightarrow instr$.

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 :  $Z \rightarrow instr$   
| iBinop : binop  $\rightarrow instr$ .
```

Definition *prog* := *list instr*.

Definition *stack* := *list Z*.

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

Single sort: Target language denotation

Inductive $instr : Set :=$
 | $iConst : Z \rightarrow instr$
 | $iBinop : binop \rightarrow instr$.

Definition $prog := list\ instr$.

Definition $stack := list\ Z$.

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

Single sort: Target language denotation

Inductive $instr : Set :=$
| $iConst : Z \rightarrow instr$
| $iBinop : binop \rightarrow instr$.

Definition $prog := list instr$.

Definition $stack := list Z$.

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

Single sort: Target language denotation

Inductive $instr : Set :=$
 $| iConst : Z \rightarrow instr$
 $| iBinop : binop \rightarrow instr.$

Definition $prog := list\ instr.$

Definition $stack := list\ Z.$

Definition $instrDenote (i : instr) (s : stack) : option\ stack := \dots$

Fixpoint $progDenote (p : prog) (s : stack) : option\ stack :=$
 $match\ p\ with$
 $| nil \Rightarrow Some\ s$
 $| i :: p' \Rightarrow match\ instrDenote\ i\ s\ with$
 $| None \Rightarrow None$
 $| Some\ s' \Rightarrow progDenote\ p'\ s'$
 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* ($e : \text{exp}$) : *prog* :=
 match e with
 | *Const* $n \Rightarrow i\text{Const } n :: \text{nil}$
 | *Binop* $b \ e1 \ e2 \Rightarrow \text{compile } e2 \ ++ \ \text{compile } e1 \ ++ \ i\text{Binop } b :: \text{nil}$
 end.

Theorem *compile_correct* :
 $\forall e, \text{progDenote } (\text{compile } e) \ \text{nil} = \text{Some } (\text{expDenote } e :: \text{nil}).$

Lemma *compile_correct'* : $\forall e \ p \ s,$
 $\text{progDenote } (\text{compile } e \ ++ \ p) \ s = \text{progDenote } p \ (\text{expDenote } e :: s).$

Single sort: Compilation correctness

Fixpoint *compile* ($e : \text{exp}$) : *prog* :=
 match e with
 | *Const* $n \Rightarrow i\text{Const } n :: \text{nil}$
 | *Binop* $b e1 e2 \Rightarrow \text{compile } e2 ++ \text{compile } e1 ++ i\text{Binop } b :: \text{nil}$
 end.

Theorem *compile_correct* :
 $\forall e, \text{progDenote } (\text{compile } e) \text{ nil} = \text{Some } (\text{expDenote } e :: \text{nil}).$

Lemma *compile_correct'* : $\forall e p s,$
 $\text{progDenote } (\text{compile } e ++ p) s = \text{progDenote } p (\text{expDenote } e :: s).$

Single sort: Compilation correctness

Fixpoint *compile* ($e : \text{exp}$) : *prog* :=
 match e with
 | *Const* $n \Rightarrow i\text{Const } n :: \text{nil}$
 | *Binop* $b \ e1 \ e2 \Rightarrow \text{compile } e2 \ ++ \ \text{compile } e1 \ ++ \ i\text{Binop } b :: \text{nil}$
 end.

Theorem *compile_correct* :
 $\forall e, \text{progDenote } (\text{compile } e) \ \text{nil} = \text{Some } (\text{expDenote } e :: \text{nil}).$

Lemma *compile_correct'* : $\forall e \ p \ s,$
 $\text{progDenote } (\text{compile } e \ ++ \ p) \ s = \text{progDenote } p \ (\text{expDenote } e :: s).$

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*
 | *MBCConst* : *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* ($s : \text{sort}$) : Set :=

```

match s with
| Mint  $\Rightarrow Z$ 
| Mbool  $\Rightarrow \text{bool}$ 
end.

```

Definition *mbinopDenote* $\text{arg1 arg2 res } (b : \text{mbinop arg1 arg2 res})$
 : *sortDenote* $\text{arg1} \rightarrow \text{sortDenote } \text{arg2} \rightarrow \text{sortDenote } \text{res} :=$

```

match b with
| MPlus  $\Rightarrow Z.\text{add}$ 
| MTimes  $\Rightarrow Z.\text{mul}$ 
| MEq Mint  $\Rightarrow Z.\text{eqb}$ 
| MEq Mbool  $\Rightarrow \text{Bool.eqb}$ 
| MLt  $\Rightarrow Z.\text{ltb}$ 
end.

```

Fixpoint *mexpDenote* $s (e : \text{mexp } s) : \text{sortDenote } s :=$

```

match e with
| MZConst  $n \Rightarrow n$ 
| MBConst  $b \Rightarrow b$ 
| MRinop  $b e1 e2 \Rightarrow$ 

```


Many sorts: Source language denotation

Definition *sortDenote* ($s : \text{sort}$) : Set :=
 match s with $Mint \Rightarrow Z$ | $Mbool \Rightarrow \text{bool}$ end.

Definition *mbinopDenote* $arg1\ arg2\ res$ ($b : \text{mbinop}\ arg1\ arg2\ res$)
 : $\text{sortDenote}\ arg1 \rightarrow \text{sortDenote}\ arg2 \rightarrow \text{sortDenote}\ res :=$
 match b with
 | $MPlus \Rightarrow Z.add$
 | $MTimes \Rightarrow Z.mul$
 | $MEq\ Mint \Rightarrow Z.eqb$
 | $MEq\ Mbool \Rightarrow \text{Bool.eqb}$
 | $MLt \Rightarrow Z.ltb$
 end.

Fixpoint *mexpDenote* s ($e : \text{mexp}\ s$) : $\text{sortDenote}\ s :=$
 match e with
 | $MZConst\ n \Rightarrow n$
 | $MBCConst\ b \Rightarrow b$
 | $MBinop\ b\ e1\ e2 \Rightarrow$
 $(\text{mbinopDenote}\ b)\ (\text{mexpDenote}\ e1)\ (\text{mexpDenote}\ e2)$
 end.

Many sorts: Source language denotation

Definition *sortDenote* ($s : \text{sort}$) : Set :=
 match s with $Mint \Rightarrow Z \mid Mbool \Rightarrow \text{bool}$ end.

Definition *mbinopDenote* $arg1\ arg2\ res$ ($b : \text{mbinop}\ arg1\ arg2\ res$)
 : $\text{sortDenote}\ arg1 \rightarrow \text{sortDenote}\ arg2 \rightarrow \text{sortDenote}\ res :=$
 match b with
 | $MPlus \Rightarrow Z.add$
 | $MTimes \Rightarrow Z.mul$
 | $MEq\ Mint \Rightarrow Z.eqb$
 | $MEq\ Mbool \Rightarrow Bool.eqb$
 | $MLt \Rightarrow Z.ltb$
 end.

Fixpoint *mexpDenote* s ($e : \text{mexp}\ s$) : $\text{sortDenote}\ s :=$
 match e with
 | $MZConst\ n \Rightarrow n$
 | $MBCConst\ b \Rightarrow b$
 | $MBinop\ b\ e1\ e2 \Rightarrow$
 $(\text{mbinopDenote}\ b)\ (\text{mexpDenote}\ e1)\ (\text{mexpDenote}\ e2)$
 end.

Many sorts: Source language denotation

Definition *sortDenote* ($s : \text{sort}$) : Set :=
 match s with $Mint \Rightarrow Z \mid Mbool \Rightarrow \text{bool}$ end.

Definition *mbinopDenote* $arg1\ arg2\ res$ ($b : \text{mbinop}\ arg1\ arg2\ res$)
 : $\text{sortDenote}\ arg1 \rightarrow \text{sortDenote}\ arg2 \rightarrow \text{sortDenote}\ res :=$
 match b with
 | $MPlus \Rightarrow Z.add$
 | $MTimes \Rightarrow Z.mul$
 | $MEq\ Mint \Rightarrow Z.eqb$
 | $MEq\ Mbool \Rightarrow Bool.eqb$
 | $MLt \Rightarrow Z.ltb$
 end.

Fixpoint *mexpDenote* s ($e : \text{mexp}\ s$) : $\text{sortDenote}\ s :=$
 match e with
 | $MZConst\ n \Rightarrow n$
 | $MBCConst\ b \Rightarrow b$
 | $MBinop\ b\ e1\ e2 \Rightarrow$
 $(\text{mbinopDenote}\ b)\ (\text{mexpDenote}\ e1)\ (\text{mexpDenote}\ e2)$
 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 $minstr : sstack \rightarrow sstack \rightarrow Set :=$
| $MiZConst : \forall ss, Z \rightarrow minstr\ ss\ (Mint :: ss)$
| $MiBConst : \forall ss, bool \rightarrow minstr\ ss\ (Mbool :: ss)$
| $MiBinop : \forall arg1\ arg2\ res\ ss,$
 $mbinop\ arg1\ arg2\ res \rightarrow minstr\ (arg1 :: arg2 :: ss)\ (res :: ss)$.

Inductive $mprog : sstack \rightarrow sstack \rightarrow Set :=$
| $MNil : \forall ss, mprog\ ss\ ss$
| $MCons : \forall ss1\ ss2\ ss3, minstr\ ss1\ ss2 \rightarrow mprog\ ss2\ ss3 \rightarrow mprog\ ss1\ ss3$.

Many sorts: Target language

Definition $sstack := list\ sort$.

Inductive $minstr : sstack \rightarrow sstack \rightarrow Set :=$
 | $MiZConst : \forall ss, Z \rightarrow minstr\ ss\ (Mint :: ss)$
 | $MiBConst : \forall ss, bool \rightarrow minstr\ ss\ (Mbool :: ss)$
 | $MiBinop : \forall arg1\ arg2\ res\ ss,$
 $mbinop\ arg1\ arg2\ res \rightarrow minstr\ (arg1 :: arg2 :: ss)\ (res :: ss)$.

Inductive $mprog : sstack \rightarrow sstack \rightarrow Set :=$
 | $MNil : \forall ss, mprog\ ss\ ss$
 | $MCons : \forall ss1\ ss2\ ss3, minstr\ ss1\ ss2 \rightarrow mprog\ ss2\ ss3 \rightarrow mprog\ ss1\ ss3$.

Many sorts: Target language denotation

```

Fixpoint vstack (ss : sstack) : Set :=
  match ss with
  | nil  $\Rightarrow$  unit
  | s :: ss'  $\Rightarrow$  sortDenote s  $\times$  vstack ss'
  end.

```

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

```

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

```

```

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss  $\rightarrow$  vstack ss' :=
  match p with
  | MNil _  $\Rightarrow$  fun vs  $\Rightarrow$  vs
  | MCons i p'  $\Rightarrow$  fun vs  $\Rightarrow$  mprogDenote p' (minstrDenote i vs)
  end.

```

Many sorts: Target language denotation

```

Fixpoint vstack (ss : sstack) : Set :=
  match ss with
  | nil  $\Rightarrow$  unit
  | s :: ss'  $\Rightarrow$  sortDenote s  $\times$  vstack ss'
  end.

```

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

```

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

```

```

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss  $\rightarrow$  vstack ss' :=
  match p with
  | MNil _  $\Rightarrow$  fun vs  $\Rightarrow$  vs
  | MCons i p'  $\Rightarrow$  fun vs  $\Rightarrow$  mprogDenote p' (minstrDenote i vs)
  end.

```


Many sorts: Target language denotation

```
Fixpoint vstack (ss : sstack) : Set :=
  match ss with nil  $\Rightarrow$  unit | s :: ss'  $\Rightarrow$  sortDenote s  $\times$  vstack ss' end.
```

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

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

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

Many sorts: Target language denotation

Fixpoint $vstack (ss : sstack) : Set :=$
 $match\ ss\ with\ nil \Rightarrow unit \mid s :: ss' \Rightarrow sortDenote\ s \times vstack\ ss' end.$

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

Definition $minstrDenote\ ss\ ss' (i : minstr\ ss\ ss') : vstack\ ss \rightarrow vstack\ ss' :=$
 $match\ i\ with$
 $\mid\ MiZConst\ _\ n \Rightarrow fun\ vs \Rightarrow (n, vs)$
 $\mid\ MiBConst\ _\ b \Rightarrow fun\ vs \Rightarrow (b, vs)$
 $\mid\ MiBinop\ _\ b \Rightarrow fun\ vs \Rightarrow$
 $\quad let\ '(v1, (v2, vs')) := vs\ in\ ((mbinopDenote\ b)\ v1\ v2, vs')$
 $end.$

Fixpoint $mprogDenote\ ss\ ss' (p : mprog\ ss\ ss') : vstack\ ss \rightarrow vstack\ ss' :=$
 $match\ p\ with$
 $\mid\ MNil\ _ \Rightarrow fun\ vs \Rightarrow vs$
 $\mid\ MCons\ i\ p' \Rightarrow fun\ vs \Rightarrow mprogDenote\ p' (minstrDenote\ i\ vs)$
 $end.$

Many sorts: Target language denotation

Fixpoint $vstack (ss : sstack) : Set :=$
 match ss with $nil \Rightarrow unit \mid s :: ss' \Rightarrow sortDenote s \times vstack ss'$ end.

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

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

Fixpoint $mprogDenote ss ss' (p : mprog ss ss') : vstack ss \rightarrow vstack ss' :=$
 match p with
 | $MNil _ \Rightarrow fun vs \Rightarrow vs$
 | $MCons i p' \Rightarrow fun vs \Rightarrow mprogDenote p' (minstrDenote i vs)$
 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* : $\forall s (e : mexp s)$,
 $mprogDenote (mcompile e nil) tt = (mexpDenote e, tt)$.

Lemma *mcompile_correct'* : $\forall s (e : mexp s) ss (v : vstack ss)$,
 $mprogDenote (mcompile e ss) v = (mexpDenote e, v)$.

Many sorts: Compilation correctness

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

Theorem $mcompile_correct : \forall\ s\ (e : mexp\ s),$
 $mprogDenote\ (mcompile\ e\ nil)\ tt = (mexpDenote\ e,\ tt).$

Lemma $mcompile_correct' : \forall\ s\ (e : mexp\ s)\ ss\ (v : vstack\ ss),$
 $mprogDenote\ (mcompile\ e\ ss)\ v = (mexpDenote\ e,\ v).$

Many sorts: Compilation correctness

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

Theorem $mcompile_correct : \forall\ s\ (e : mexp\ s),$
 $mprogDenote\ (mcompile\ e\ nil)\ tt = (mexpDenote\ e,\ tt).$

Lemma $mcompile_correct' : \forall\ s\ (e : mexp\ s)\ ss\ (v : vstack\ ss),$
 $mprogDenote\ (mcompile\ e\ ss)\ v = (mexpDenote\ e,\ v).$

Function pred

Print *pred*.

```
pred = fun n : nat => match n with
  | 0 => 0
  | Su => u
end
: nat -> nat
```


Function `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

Function `pred_strong1`

Lemma `zgtz` : $0 > 0 \rightarrow$ **False**.

intro H.

inversion H.

Qed.

Definition `pred_strong1` ($n : \mathbf{nat}$) : $n > 0 \rightarrow \mathbf{nat} :=$

match n with

 | `O` \Rightarrow *fun pf : $0 > 0 \Rightarrow$ match `zgtz` *pf* with end*

 | `S n'` \Rightarrow *fun _ \Rightarrow n'*

end.

Theorem `two_gt0` : $2 > 0$.

auto.

Qed.

Eval `compute in pred_strong1 two_gt0`.

 = 1

 : **nat**

Function `pred_strong1'`

```
Definition pred_strong1' (n : nat) (pf : n > 0) : nat :=  
  match n with  
  | 0 => 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`"

Function `pred_strong1'`

```

Definition pred_strong1' (n : nat) (pf : n > 0) : nat :=
  match n with
  | 0 => 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`"

Function `pred_strong1` with explicit return

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

Lists with length

Section `ilist`.

Variable `A : Set`.

Inductive `ilist : nat → Set :=`

| `INil : ilist 0`

| `ICons : ∀ n, A → ilist n → ilist (S n)`.

Dependent pattern-matching for `ilist` (1)

```
match /:ilist m
```

```
with
```

```
| INil  $\Rightarrow$  ...
```

```
| ICons  $n \times l' \Rightarrow$  ...
```

```
end. : P m
```

Dependent pattern-matching for `ilist` (2)

```
match /:ilist m
  in (ilist k)
  return (P k)
with
  | INil  $\Rightarrow$  ... : P 0
  | ICons  $n \times l' \Rightarrow$  ... : P (S n)
end. : P m
```


Dependent pattern-matching for `ilist` (3)

```
match /:ilist m
  as v:ilist k
  in (ilist k)
  return (P k v)
with
  | INil  $\Rightarrow$  ... : P 0 INil
  | ICons n x l'  $\Rightarrow$  ... : P (S n) (ICons n x l')
end. : P m l
```

Destruction - match

Simple form:

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

Full form:

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

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

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

Destruction - match (2)

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.

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

Example for $\text{ilist} : s \in \mathcal{S}$ and $P : \forall n : \text{nat}, \text{ilist } n \rightarrow s$
 $\text{ICons} : \forall n : \text{nat}, a : A, l : \text{ilist } n, \text{ilist } (S n)$
 $\{(\text{ICons})\}^P \equiv \forall n : \text{nat}, a : A, l : \text{ilist } n, P (S n) (\text{ICons } n a l).$

Allowed elimination sorts

Assume that $m : I$, $I : A$ and $\lambda a x.P : B$.

To eliminate m using P on must have $[I:A \mid B]$.

$[I:A \mid B]$ is the smallest relation satisfying the following rules:

- if $[(I \ x):A' \mid B']$ then $[I:\text{forall } x:A, A' \mid \text{forall } x:A, B']$
- $[I:s1 \mid I \rightarrow s2]$ for any $s1 \in \text{Set}, \text{Type}(j)$, $s2 \in \mathcal{S}$
- $[I:\text{Prop} \mid I \rightarrow s]$ for any $s \in \text{SProp}, \text{Prop}$
- $[I:\text{Prop} \mid I \rightarrow s]$ for I empty or singleton definition (one constructor and all its arguments are in Prop), $s \in \mathcal{S}$

Allowed elimination sorts

Assume that $m : I, I : A$ and $\lambda a x.P : B$.

To eliminate m using P on must have $[I:A \mid B]$.

$[I:A \mid B]$ is the smallest relation satisfying the following rules:

- if $[(I \ x):A' \mid B']$ then $[I:\text{forall } x:A, A' \mid \text{forall } x:A, B']$
- $[I:s1 \mid I \rightarrow s2]$ for any $s1 \in \text{Set}, \text{Type}(j), s2 \in \mathcal{S}$
- $[I:\text{Prop} \mid I \rightarrow s]$ for any $s \in \text{SProp}, \text{Prop}$
- $[I:\text{Prop} \mid I \rightarrow s]$ for I empty or singleton definition (one constructor and all its arguments are in Prop), $s \in \mathcal{S}$

Allowed elimination sorts

Assume that $m : I, I : A$ and $\lambda a x.P : B$.

To eliminate m using P on must have $[I:A \mid B]$.

$[I:A \mid B]$ is the smallest relation satisfying the following rules:

- if $[(I \ x):A' \mid B']$ then $[I:\text{forall } x:A, A' \mid \text{forall } x:A, B']$
- $[I:s1 \mid I \rightarrow s2]$ for any $s1 \in \text{Set}, \text{Type}(j), s2 \in \mathcal{S}$
- $[I:\text{Prop} \mid I \rightarrow s]$ for any $s \in \text{SProp}, \text{Prop}$
- $[I:\text{Prop} \mid I \rightarrow s]$ for I empty or singleton definition (one constructor and all its arguments are in Prop), $s \in \mathcal{S}$

Allowed elimination sorts

Assume that $m : I$, $I : A$ and $\lambda a x.P : B$.

To eliminate m using P on must have $[I:A \mid B]$.

$[I:A \mid B]$ is the smallest relation satisfying the following rules:

- if $[(I \ x):A' \mid B']$ then $[I:\text{forall } x:A, A' \mid \text{forall } x:A, B']$
- $[I:s1 \mid I \rightarrow s2]$ for any $s1 \in \text{Set}, \text{Type}(j)$, $s2 \in \mathcal{S}$
- $[I:\text{Prop} \mid I \rightarrow s]$ for any $s \in \text{SProp}, \text{Prop}$
- $[I:\text{Prop} \mid I \rightarrow s]$ for I empty or singleton definition (one constructor and all its arguments are in Prop), $s \in \mathcal{S}$

Destruction - match (3)

Typing rule.

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

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

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

lota reduction

A ι -reduction has the following form

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

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

Typing: pred_strong1

```

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

```

```

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

```

Typing: pred_strong1

```

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

```

```

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

```

Typing: append of `ilist`

```

Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2) :=
  match ls1 in (ilist k) return (ilist (k + n2)) with
  | INil  $\Rightarrow$  ls2
  | ICons _ x ls1'  $\Rightarrow$  ICons x (app' ls1' ls2)
end.

```

```

Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2) :=
  match ls1 in (ilist k) return (ilist (k + n2)) with
  | INil  $\Rightarrow$  ls2   : ilist (0+n2)
  | ICons n' x ls1'  $\Rightarrow$  ICons x (app' ls1' ls2)   : ilist (S n'+n2)
end.

```

Typing: append of `ilist`

```

Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2) :=
  match ls1 in (ilist k) return (ilist (k + n2)) with
  | INil  $\Rightarrow$  ls2
  | ICons _ x ls1'  $\Rightarrow$  ICons x (app' ls1' ls2)
  end.

```

```

Fixpoint app' n1 (ls1 : ilist n1) n2 (ls2 : ilist n2) : ilist (n1 + n2) :=
  match ls1 in (ilist k) return (ilist (k + n2)) with
  | INil  $\Rightarrow$  ls2   : ilist (0+n2)
  | ICons n' x ls1'  $\Rightarrow$  ICons x (app' ls1' ls2)   : ilist (S n'+n2)
  end.

```

Typing: strong elimination

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

Elimination from `nat` to `Type` is needed to show that $0 \neq 1$:

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

Typing: strong elimination

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

Elimination from nat to Type is needed to show that $0 \neq 1$:

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

Typing: strong elimination

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

Elimination from nat to Type is needed to show that $0 \neq 1$:

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


Typing: eq elimination

```

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

```

```

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

```

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

- That is an elimination from Prop to Set for a singleton type
- That is how tactic rewrite works

Typing: eq elimination

```

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

```

```

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

```

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

- That is an elimination from Prop to Set for a singleton type
- That is how tactic rewrite works

Typing: eq elimination

```

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

```

```

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

```

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

- That is an elimination from Prop to Set for a singleton type
- That is how tactic rewrite works

Typing: eq elimination

```

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

```

```

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

```

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

- That is an elimination from Prop to Set for a singleton type
- That is how tactic `rewrite` works

Typing: eq elimination

```

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

```

```

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

```

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

- That is an elimination from Prop to Set for a singleton type
- That is how tactic rewrite works

For nonsingleton types in Prop elimination to Prop only

```
Inductive or (A B:Prop) : Prop :=  
lintro : A -> or A B | rintro : B -> or A B.
```

Incorrect:

```
Definition choice (A B: Prop) (x:or A B): bool :=  
match x with lintro a => true | rintro b => false end.
```

Error: Incorrect elimination of "x" in the inductive type "or": the return type has sort "Set" while it should be "Prop".
Elimination of an inductive object of sort Prop is not allowed on a predicate in sort Set because proofs can be eliminated only to build proofs.

For nonsingleton types in Prop elimination to Prop only

```
Inductive or (A B:Prop) : Prop :=  
lintro : A -> or A B | rintro : B -> or A B.
```

Incorrect:

```
Definition choice (A B: Prop) (x:or A B): bool :=  
match x with lintro a => true | rintro b => false end.
```

Error: Incorrect elimination of "x" in the inductive type "or": the return type has sort "Set" while it should be "Prop". Elimination of an inductive object of sort Prop is not allowed on a predicate in sort Set because proofs can be eliminated only to build proofs.