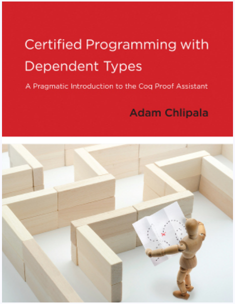
# Dependent pattern-matching in Coq

Daria Walukiewicz-Chrząszcz

20 April 2021

# Dependent types in programming

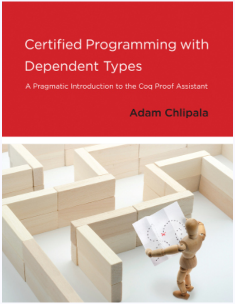
Compilation correctness — example from "Certified Programming with Dependent Types" by Adam Chlipala (MIT)




- 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* : *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, tt))) : 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, tt))) : 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, tt))) : 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, tt))) : 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, tt))) : 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
  $\mid MZConst\ n \Rightarrow MCons\ (MiZConst\ \_\ n)\ (MNil\ \_)$
  $\mid MBConst\ b \Rightarrow MCons\ (MiBConst\ \_\ b)\ (MNil\ \_)$
  $\mid MBinop\ b\ e1\ e2 \Rightarrow mconcat\ (mcompile\ e2\ \_)$
  $(mconcat\ (mcompile\ e1\ \_)\ (MCons\ (MiBinop\ \_\ b)\ (MNil\ \_)))$
  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 ⇒ 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) tt = (mexpDenote e, tt).
```

```
Lemma mcompile_correct' : ∀ 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 ⇒ 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) tt = (mexpDenote e, tt).
```

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

# Function pred

`Print` *pred*.

$$\text{pred} = \texttt{fun}\ n : \textbf{nat} \Rightarrow \texttt{match}\ n\ \texttt{with}$$
$$\mid 0 \Rightarrow 0$$
$$\mid \mathsf{S}u \Rightarrow u$$
$$\text{end}$$
$$: \textbf{nat} \to \textbf{nat}$$

# 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 → 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'

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"
```

# Function 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"
```

# Function pred_strong1 with explicit return

```
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.
```

# Lists with length

```
Section ilist.
  Variable A : Set.

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

# Dependent pattern-matching for ilist (1)

```
match l:ilist m

with
  | INil ⇒ ...
  | ICons n x l' ⇒ ...
end.   : P m
```

# Dependent pattern-matching for ilist (2)

```
match l:ilist m
  in (ilist k)
  return (P k)
with
  | lNil ⇒ ... : P 0
  | lCons n x l' ⇒ ... : P (S n)
end.   : P m
```

# Dependent pattern-matching for `ilist` (3)

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

# Destruction - 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 :

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

# 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.

$$\{c : (I_i\ p_1 \ldots p_r\ t_1 \ldots t_p)\}^P \equiv (P\ t_1 \ldots\ t_p\ c)$$

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

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

# 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 |B]`.

`[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 $s1 \in \texttt{Set}, \texttt{Type}(j)$, $s2 \in S$
- `[I:Prop|I -> s]` for any $s \in \texttt{SProp}, \texttt{Prop}$
- `[I:Prop|I -> s]` for I empty or singleton definition (one constructor and all its argumnets are in `Prop`), $s \in S$

# 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 |B]`.

`[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 $s1 \in \texttt{Set}, \texttt{Type}(j)$, $s2 \in S$
- `[I:Prop|I -> s]` for any $s \in \texttt{SProp}, \texttt{Prop}$
- `[I:Prop|I -> s]` for I empty or singleton definition (one constructor and all its argumnets are in Prop), $s \in S$

# 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 |B].

[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 -> s] for any $s \in$ SProp, Prop
- [I:Prop|I -> s] for I empty or singleton definition (one constructor and all its argumnets are in Prop), $s \in S$

# 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 |B]`.

`[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 -> s]` for any $s \in$ SProp, Prop
- `[I:Prop|I -> s]` for `I` empty or singleton definition (one constructor and all its argumnets are in `Prop`), $s \in S$

# Destruction - match (3)

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

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

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

# Iota reduction

A $\iota$-reduction has the following form

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

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

# Typing: 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.
```

```
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     : 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
    | O ⇒ 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
    | O ⇒ 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 ⇒ ls2
      | ICons _ x ls1' ⇒ 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 ⇒ ls2    : ilist (0+n2)
      | ICons n' x ls1' ⇒ 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
      | lNil ⇒ ls2
      | lCons _ x ls1' ⇒ lCons 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
      | lNil ⇒ ls2    : ilist (0+n2)
      | lCons n' x ls1' ⇒ lCons 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
```

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

- 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
```

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

- 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
```

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

- 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
```

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

- 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
```

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

- 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.
```