依存型の話

田中　哲

産業技術総合研究所
2018-09-02

Proof Summit 2018

産総研 Research Assistant 募集中

- 研究内容
  - IoT の安全性のための検証ツールの研究開発
  - ① 定理証明支援系 Coq によるプログラムの形式化・形式仕様作成・形式検証
    (参考文献 : https://staff.aist.go.jp/reynald.affeldt/documents/mona_equa.pdf)
  - ② 定理証明支援系 Coq による C 言語のコード生成の実験
    (参考文献 : https://staff.aist.go.jp/tanaka-akira/succinct/)
- https://unit.aist.go.jp/hrd/keiyaku_koubo/30-itri_0018.html

注意

- Coq の話です
- 型の玄人ではないです
- 変なことを言っていたら教えてください

依存型の噂

- すごいらしい
- 証明できるらしい
- リストの長さを型で扱えるらしい
- なんか面倒くさいらしい

型システム入門 p.365

- 他の式でインデックス付けされた式の族
  - λx:T. t　項でインデックス付けされた項の族
    (通常のラムダ抽象、関数)
  - λX::K. t　型でインデックス付けされた項の族
    (型抽象、多相型関数)
  - λX::K. T 型でインデックス付けされた型の族
    (型演算子、パラメタ付き型)
- 「上記のリストには明らかにまだ考えていない可能性がある」
  - λx:T. T　値でインデックス付けされた型の族
    （依存型)

他の式でインデックス付けされた式の族

- λx:T. t　　項でインデックス付けされた項の族
  (通常のラムダ抽象、関数)
- λX::K. t　　型でインデックス付けされた項の族
  (型抽象、多相型関数)
- λX::K. T　　型でインデックス付けされた型の族
  (型演算子、パラメタ付き型)
- λx:T. T　　値でインデックス付けされた型の族
  （依存型)

λx:T. t 項でインデックス付けされた項の族 (通常のラムダ抽象、関数)

- 項でインデックス付けされた項の族
  - 項 a ↦ 項 A
  - 項 b ↦ 項 B
  - ...
- 後者関数 S : nat → nat
  - 0 ↦ S 0 = 1
  - 1 ↦ S 1 = 2
  - ...

λX::K. t　型でインデックス付けされた項の族 (型抽象、多相型関数)

- 型でインデックス付けされた項の族
  - 型 a ↦ 項 A
  - 型 b ↦ 項 B
  - ...
- Some : forall A : Type, A → option A
  - bool ↦ Some bool =
    { true ↦ Some true, false ↦ Some false }
  - nat ↦ Some nat =
    { 0 ↦ Some 0, 1 ↦ Some 1, 2 ↦ Some 2, ... }
  - ...

λX::K. T 型でインデックス付けされた型の族 (型演算子、パラメタ付き型)

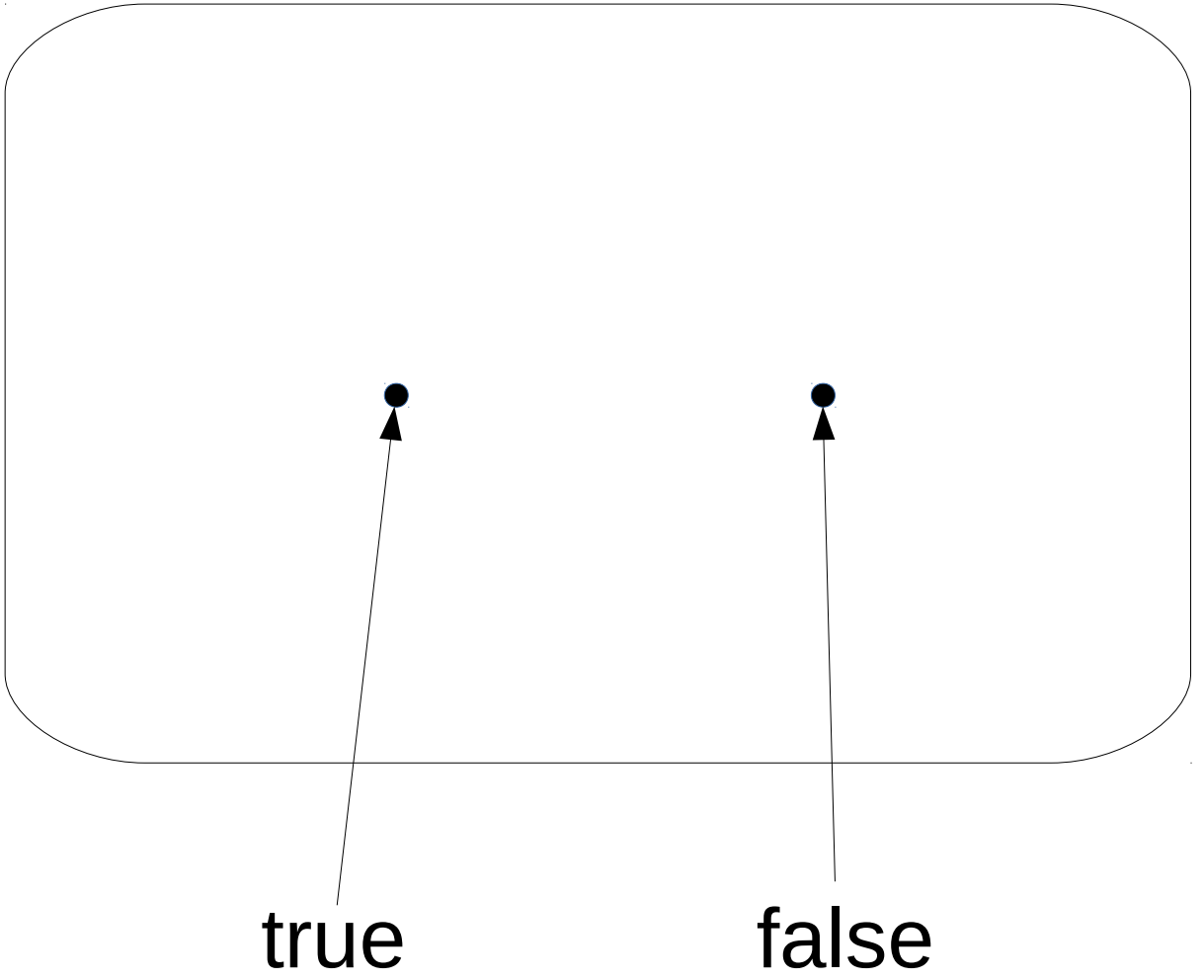
- 型でインデックス付けされた型の族
  - 型 a ↦ 型 A
  - 型 b ↦ 型 B
  - ...
- option : Type → Type
  - bool ↦ option bool =
    { None, Some true, Some false }
  - nat ↦ option nat =
    { None, Some 0, Some 1, Some 2, ... }
  - ...

λx:T. T 項でインデックス付けされた型の族 (依存型)

- 項でインデックス付けされた型の族
  - 項 a ↦ 型 A
  - 項 b ↦ 型 B
  - ...
- even : nat → Prop
  - 0 ↦ even 0 = { even_O }
  - 1 ↦ even 1 = {}
  - 2 ↦ even 2 = { even_S (odd_S even_O) }
  - 3 ↦ even 3 = {}
  - ...

復習：依存型じゃない型

bool 型



Coq の bool の定義 (theories/Init/Datatypes.v)

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

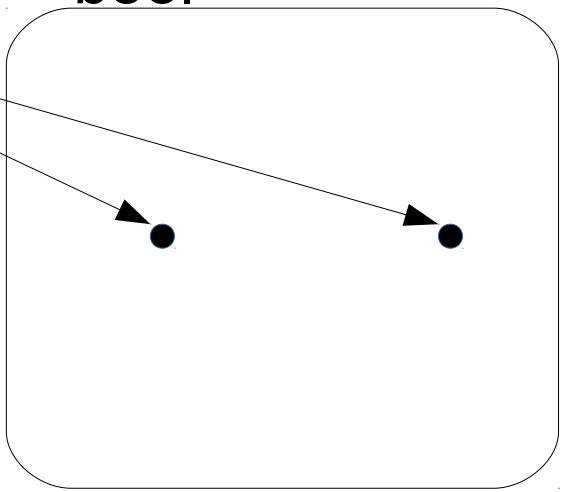
Coq の bool と集合

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

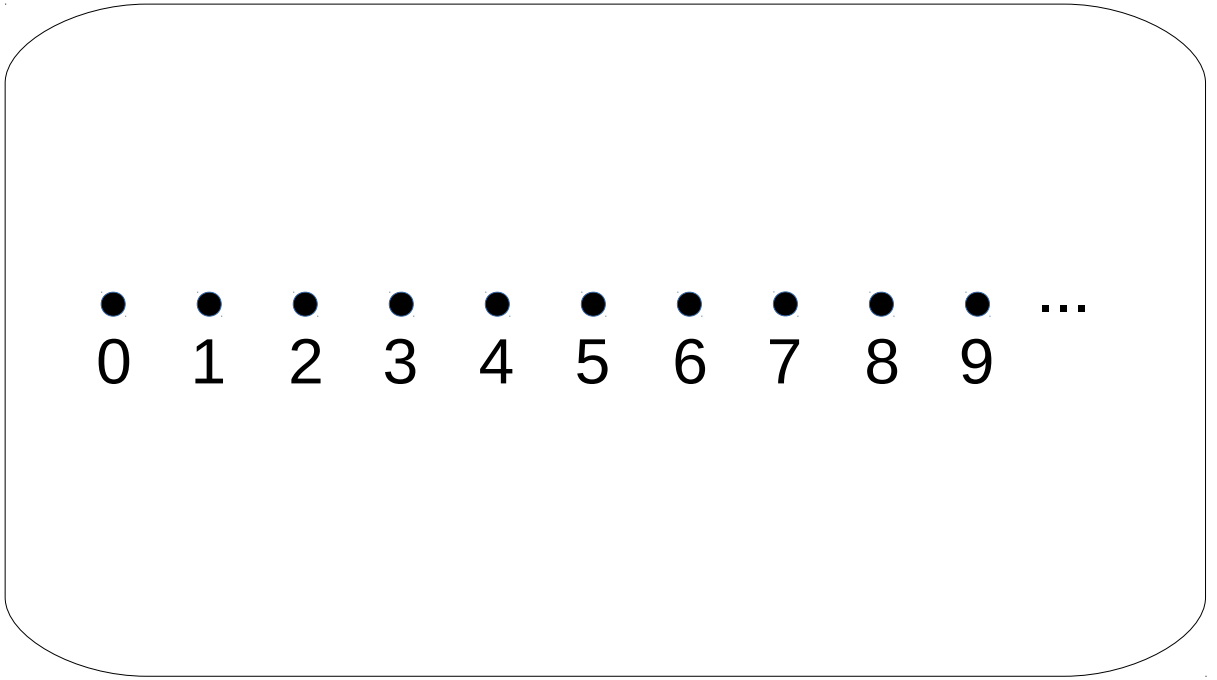
bool という集合を作る

bool

bool 型の値 true と
bool 型の値 false を作る



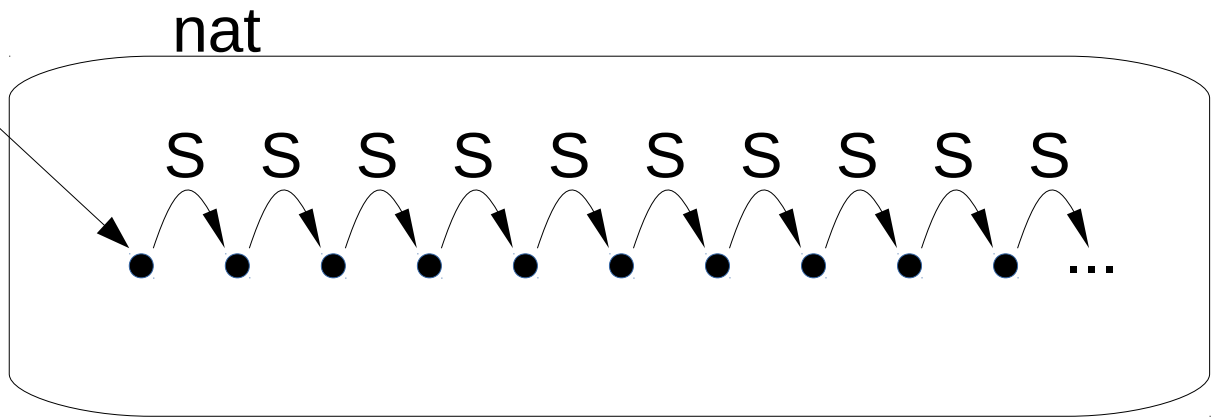
自然数 nat



Coq の nat の定義 (theories/Init/Datatypes.v)

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

S は nat の要素から nat の新しい要素を作る関数



0 = O
1 = S O
2 = S (S O)
3 = S (S (S O))
... という対応を想定すれば自然数と考えられる

add (theories/Init/Nat.v)

```
Fixpoint add n m :=
  match n with
  | 0 => m
  | S p => S (p + m)
  end
where "n + m" := (add n m) : nat_scope.
```

- 0 = O, 1 = S O, 2 = S (S O), ... と想定すれば add は自然数の加算と考えられる
- 1 = O, 2 = S O, 3 = S (S O), ... と想定すれば add は自然数の加算ではない
- これに add という名前がついているということは 0 = O ということであろう

convertible

- Coq では計算を進めて同じになる項は同じものとみなされる
(型推論や tactic の動作については変わることもある)
- この「同じ」を convertible と呼ぶ
- a と b が convertible なら a=b は reflexivity で証明できるし、a と b は unification も可能
- 例
  - add 1 2 は 3 と convertible である
  - add 2 n は S (S n) と convertible である
add 2 n = S (S n) は reflexivity で証明できる
  - add n 2 は S (S n) と convertible ではない
add n 2 = S (S n) は reflexivity では証明できない
(induction が必要)

```
Fixpoint
  add n m :=
match n with
| 0 => m
| S p =>
   S (add p m)
end.
```

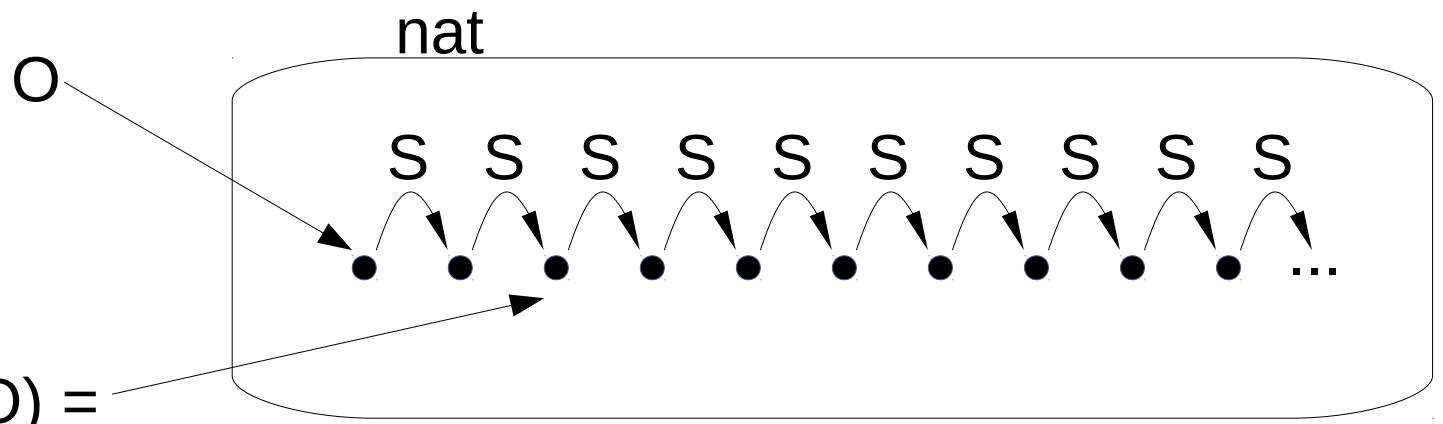
convertible の定義

ふたつの項が以下の reduction で同じ項にたどり着けるなら convertible

- β-reduction (beta)　関数呼び出しの簡約 (引数を置換した関数本体に置換)
- ι-reduction (iota)
  - match　分岐を選ぶ (制約 : match 対象がコンストラクタになっている)
  - fix　再帰関数呼び出しを beta redex に変える
(制約 : decreasing argument がコンストラクタになっている)
- δ-reduction (delta)
  - Delta-Local　let で束縛された変数をその定義を置換する
  - Delta-Global　定数参照をその定義に置換する
- ζ-reduction (zeta)　let で束縛された定義を展開して let を除去する
- η-expansion (eta)　eta 展開する (注 : eta 簡約ではない)

cf. Coq Reference Manual →
Calculus of Inductive Constructions →
Conversion rules および Well-formed inductive definitions

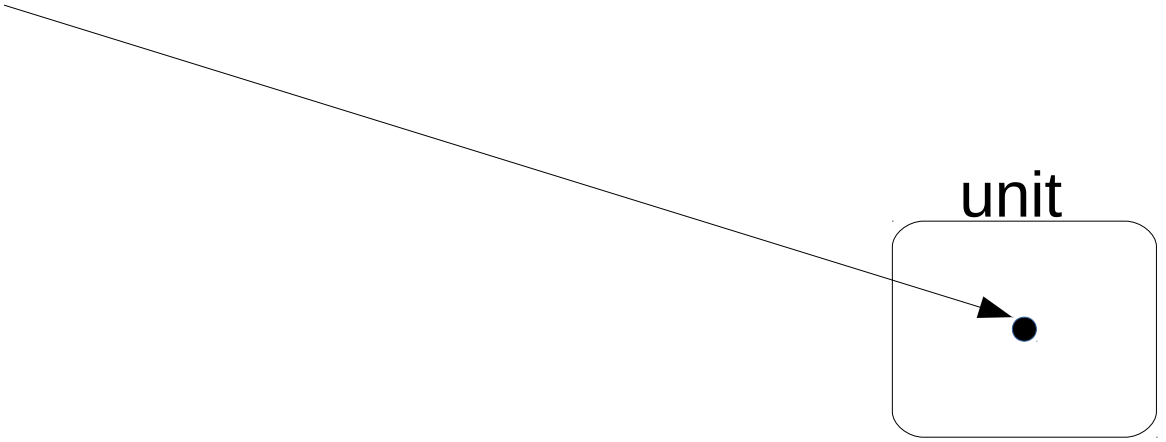
nat と convertible



S (S O) =
2 =
0 + 2 =
1 + 1 =
2 + 1 =
2 – 0 =
3 – 1 = ...

値がひとつしかない型 unit (theories/Init/Datatypes.v)

```
Inductive unit : Set :=
    tt : unit.
```

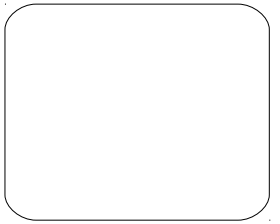


値がひとつもない型 Empty_set (theories/Init/Datatypes.v)

Inductive Empty_set : Set :=.

コンストラクタがないので要素がない

Empty_set 型の値を返すプログラムは記述できない



問題 : C 言語の void 型とは

C 言語の void 型に相当する型を以下から選べ

- unit
- Empty_set

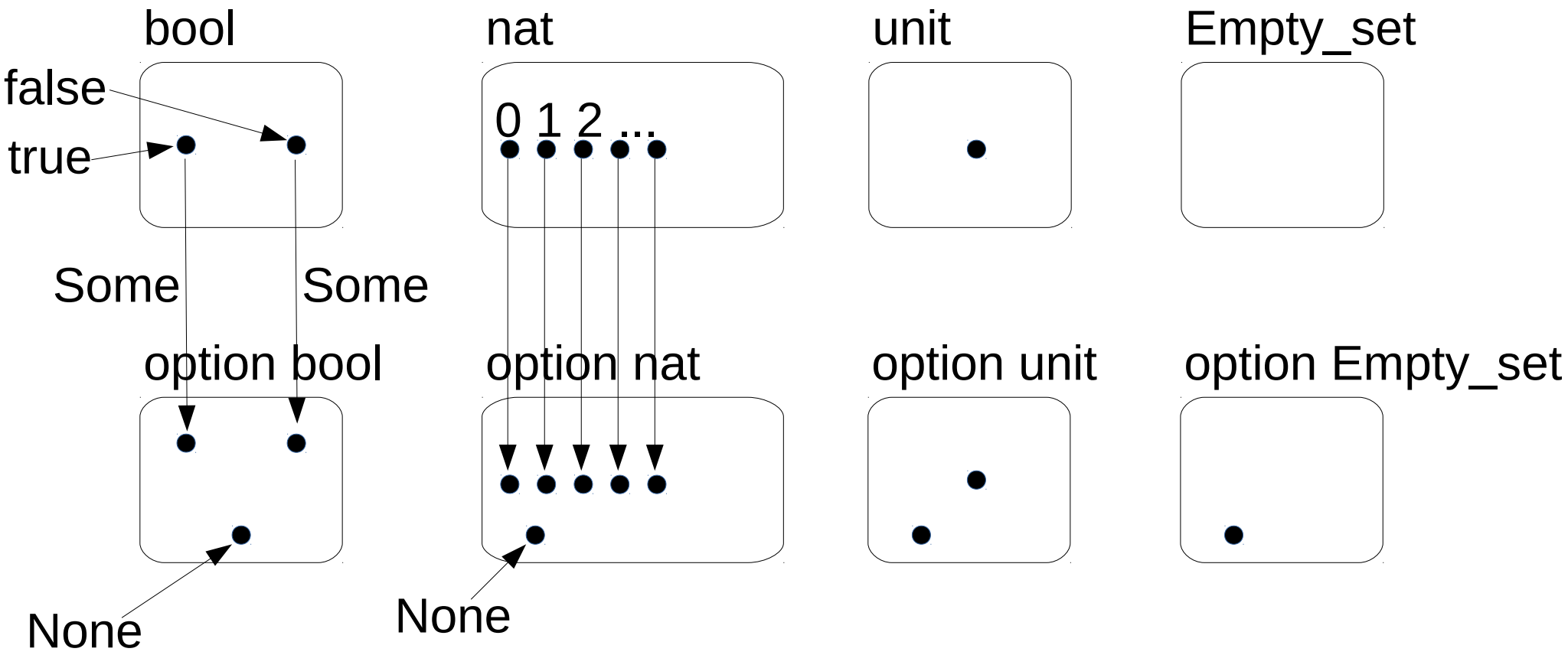
option (theories/Init/Datatypes.v)

- Inductive option (A:Type) : Type :=
  | Some : A → option A
  | None : option A.
- 型の名前の直後に書いた引数はすべてのコンストラクタが受け取るパラメータとなる
  上記の例では (A : Type) の部分
- つまり以下とだいたい等しい
  Inductive option : Type → Type :=
  | Some : forall (A : Type), A → option A
  | None : forall (A : Type), option A.

  等しくないところ：パラメータの形で書くと、コンストラクタはそのパラメータをそのまま渡した型を返さなければならない
  パラメータは match で値を取り出せない
  定義時に生成される帰納法の原理 (prod_ind とか) が変化する
- option は型を受け取って型を返す型演算子

option は型から型を作る

- 型演算子なので、型でインデックス付けされた型



ふたつの型の組 prod (theories/Init/Datatypes.v)

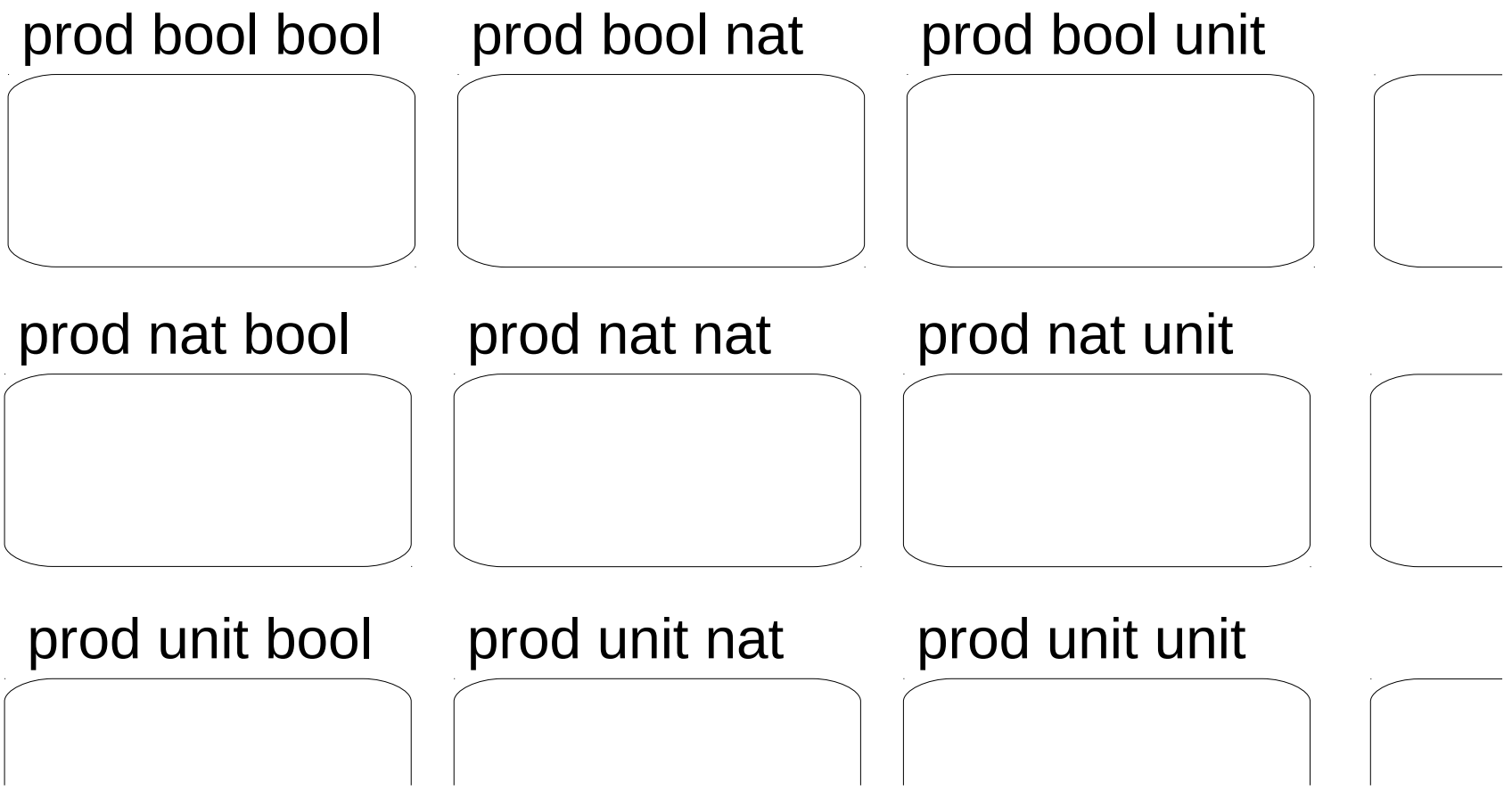
- Inductive prod (A B:Type) : Type :=
  pair : A → B → prod A B.
- パラメータは A と B

C++ の template なら

```
template <typename A, typename B>
struct prod {
  A a;
  B b;
};
```

prod は集合をたくさん作る

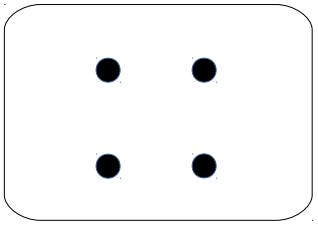
```
Inductive prod (A B:Type) : Type :=
  pair : A → B → prod A B.
```



pair はそれぞれの集合の要素を作る

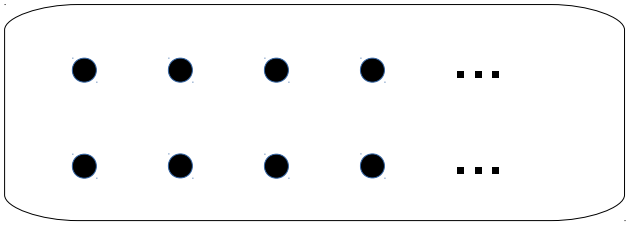
```
Inductive prod (A B:Type) : Type :=
  pair : A → B → prod A B.
```

prod bool bool



pair bool bool true true
pair bool bool true false
pair bool bool false true
pair bool bool false false

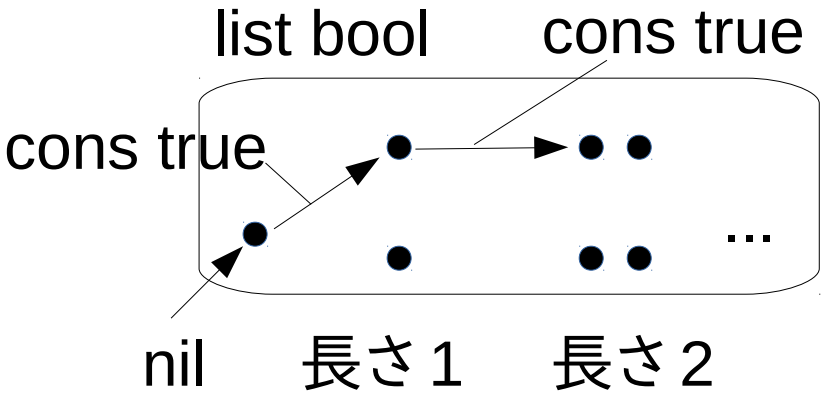
prod bool nat

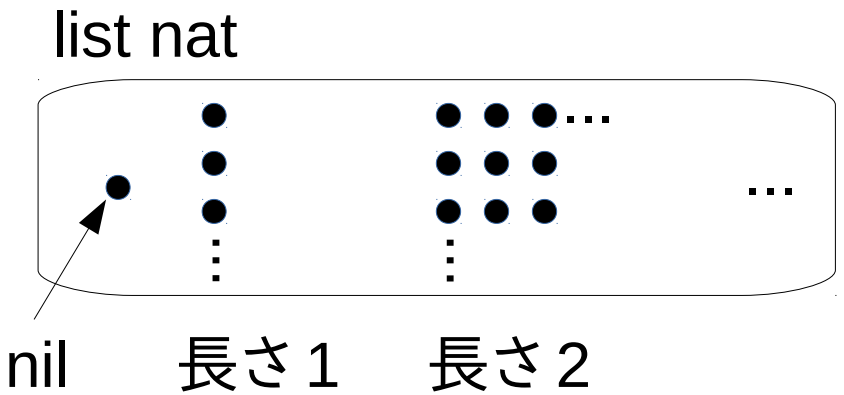


pair bool nat true O
pair bool nat true (S O)
pair bool nat true (S (S O))
...
pair bool nat false O
pair bool nat false (S O)
...

list (theories/Init/Datatypes.v)

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





依存型の話に入ろう

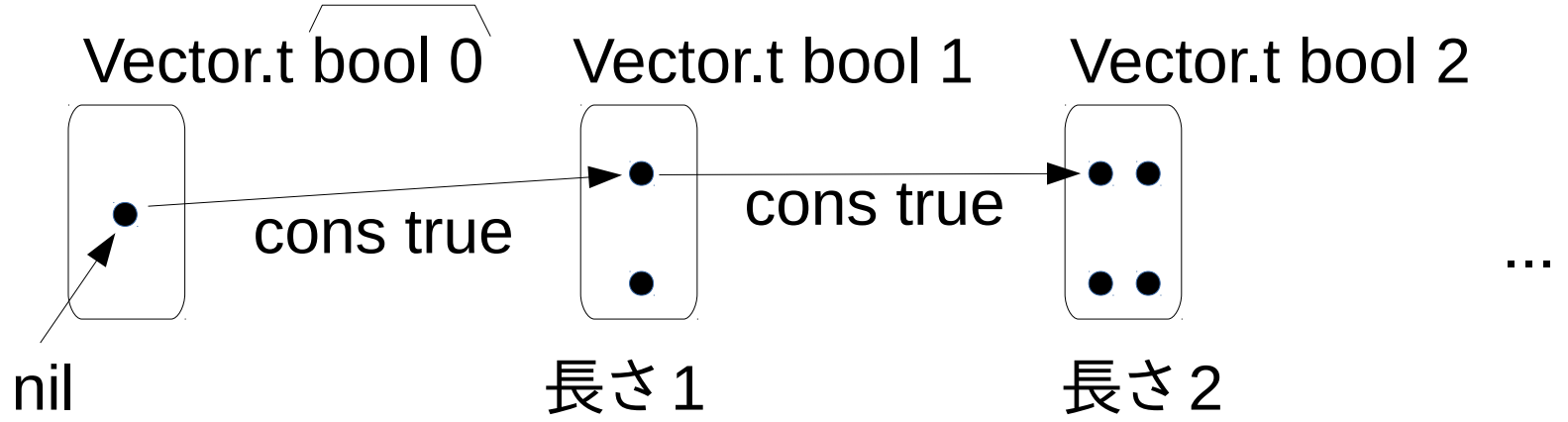
Vector (theories/Vectors/VectorDef.v)

- すごく有名な依存型
  型に長さが入ったリスト
- Vector.t の定義
  Inductive t A : nat → Type :=
  |nil : t A 0
  |cons : forall (h:A) (n:nat), t A n -> t A (S n).
- Vector ライブラリ内なので Vector.t を t と表記

Vector

- 依存型は項でインデックス付けされた型の族
- Vector.t は長さでインデックスされた型の族
- Inductive t A : nat → Type :=
  |nil : t A 0
  |cons : forall (h:A) (n:nat), t A n -> t A (S n).

ここに値（項）が出てくるのが依存型

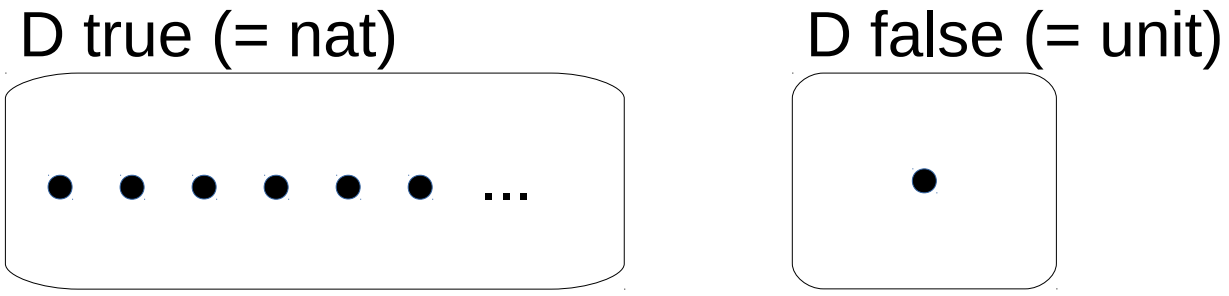


Vector に対する filter の型は？
Vector.t A n → (A → bool) → Vector.t A ？

- 引数の長さから結果の長さが判明するなら型づけできる
  map : forall A B : Type,
    (A -> B) -> forall n : nat, t A n -> t B n
- 引数の長さだけでは結果の長さが判明しないなら難しい
  filter : forall A : Type, forall n:nat,
    t A n → (A → bool) → t A？
- filter の型の中で filter を使うことはできない
  filter : forall A : Type, forall n:nat,
    forall (v : t A n) (pred : A → bool) →
    t A (length (filter A n v pred))
- 別の関数を最初に定義すればなんとか？
  filter : forall A : Type, forall n:nat,
    forall (v : t A n) (pred : A → bool) →
    t A (count A n v pred)
- Vector を使うのはアプリケーション全体で、長さが長さから判明するとわかっているときだけにするのがいいかも？

型を計算して求めることもできる

- Definition D (b : bool) : Set :=
  if b then nat else unit.
- D は bool を受け取って型を返す関数
- これも依存型
  項でインデックス付けされている型だから



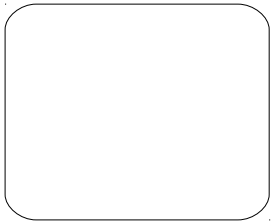
プログラムと証明

カリーハワード同型対応

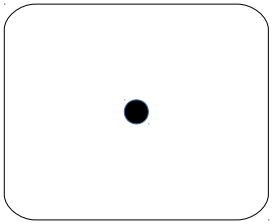
- 型と命題は同じ構造をしている
- プログラムと証明は同じ構造をしている
- 命題を型として書くと、
  その型の値を返すプログラムを証明とみなせる
- ただし例外や無限ループはないものとする

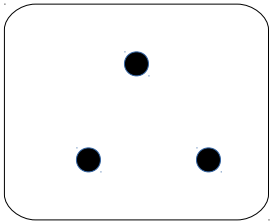
型と命題の対応

- 値がない型は証明できない命題
- 値がある型は証明できる（かもしれない）命題



証明できない





証明できるかもしれない

- 命題を表す型に複数の値がある場合、いずれかひとつの値を記述できれば証明したものとみなせる
- Coq は通常のプログラムの中で証明（に対応するプログラム）を使った場合、同じ命題を証明する複数の値を区別して動作を変えることを禁止している（証明を除去可能にするため）

Set と Prop (と Type) と extraction

- Coq では通常のプログラムの中で証明（に対応するプログラム）を使える
- 証明は存在することがわかっていれば充分なので、実行時に計算するのは無駄
- extraction により、証明以外の部分を抽出したプログラムを OCaml, Haskell, Scheme に出力できる
- extraction のために、通常のプログラムと証明を区別する必要がある
- Set は通常のプログラムの型の型（カインド）
- Prop は証明の型の型
- Type は通常のプログラムと証明のどちらでも使える値の型の型（extraction では出力される）

Wikipedia: カリー＝ハワード同型対応

論理

- 全称量化 $\forall x \in A.\ B(x)$
- 存在量化 $\exists x \in A.\ B(x)$
- 含意 $A \supset B$
- 論理積 $A \wedge B$
- 論理和 $A \vee B$
- 真 $\top$
- 偽 $\bot$

プログラミング

- 依存直積 $\prod_{x:a} B(x)$
- 依存直和 $\Sigma_{x:A} B(x)$
- 関数型 $A \to B$
- 直積型 $A \times B$
- 直和型 $A + B$
- トップ型 1
- ボトム型 0

Coq のカリー＝ハワード同型対応

論理

- 全称量化 $\forall x \in A.\ B(x)$
- 存在量化 $\exists x \in A.\ B(x)$
- 含意 $A \supset B$
- 論理積 $A \wedge B$
- 論理和 $A \vee B$
- 真 $\top$
- 偽 $\bot$

Coq

- forall (x:A), B x
- exists x:A, B x
- A → B
- A /\ B
- A \/ B
- True
- False

Coq のカリー＝ハワード同型対応 (2)

論理

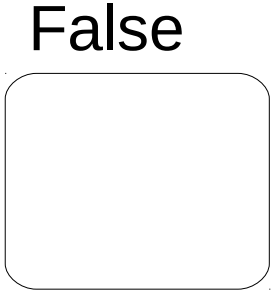
- 全称量化 $\forall x \in A.\ B(x)$
- 存在量化 $\exists x \in A.\ B(x)$
- 含意 $A \supset B$
- 論理積 $A \wedge B$
- 論理和 $A \vee B$
- 真 $\top$
- 偽 $\bot$

Coq (notation なし)

- forall (x:A), B x
- ex (fun x:A => B x)
- forall (_:A), B
- and A B
- or A B
- True
- False

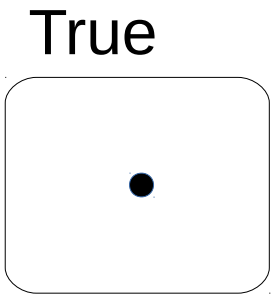
False (theories/Init/Logic.v)

- Inductive False : Prop :=.
- Prop 版の Empty_set
- 依存型ではない



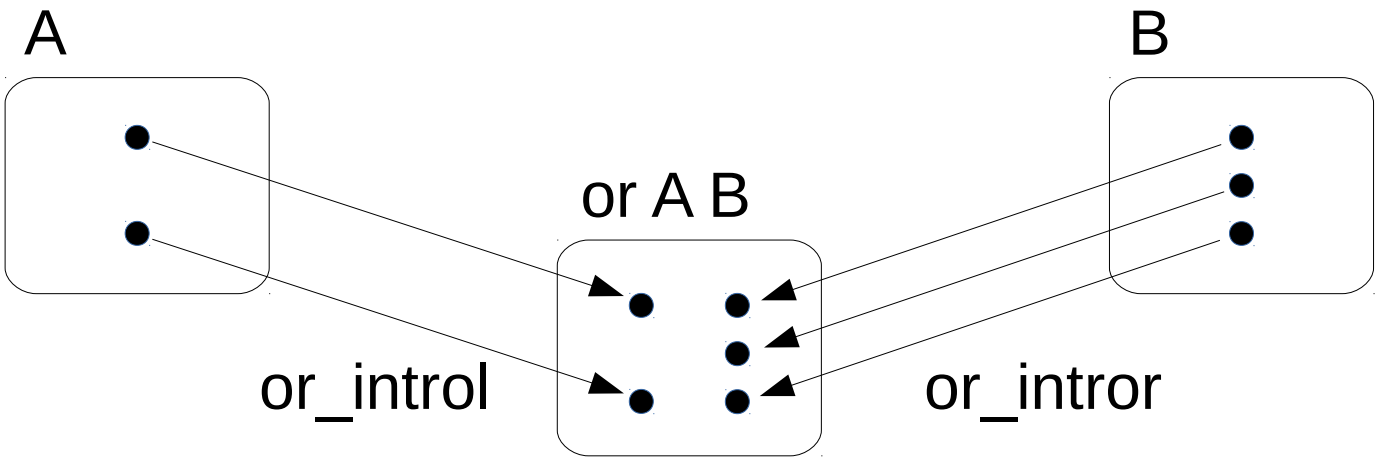
True (theories/Init/Logic.v)

- Inductive True : Prop := I : True.
- Prop 版の unit
- 依存型ではない



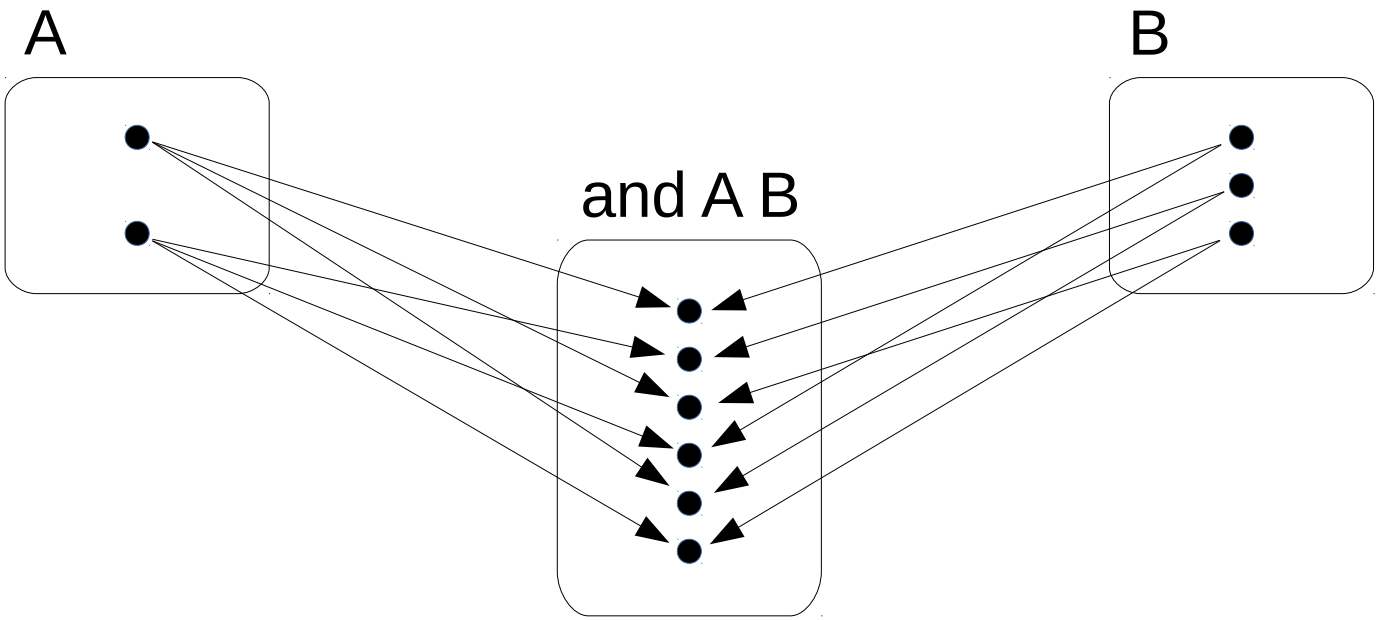
A ∨ B (theories/Init/Logic.v)

- Inductive or (A B:Prop) : Prop :=
  | or_introl : A -> A ∨ B
  | or_intror : B -> A ∨ B
  where "A ∨ B" := (or A B) : type_scope.
- 依存型ではない



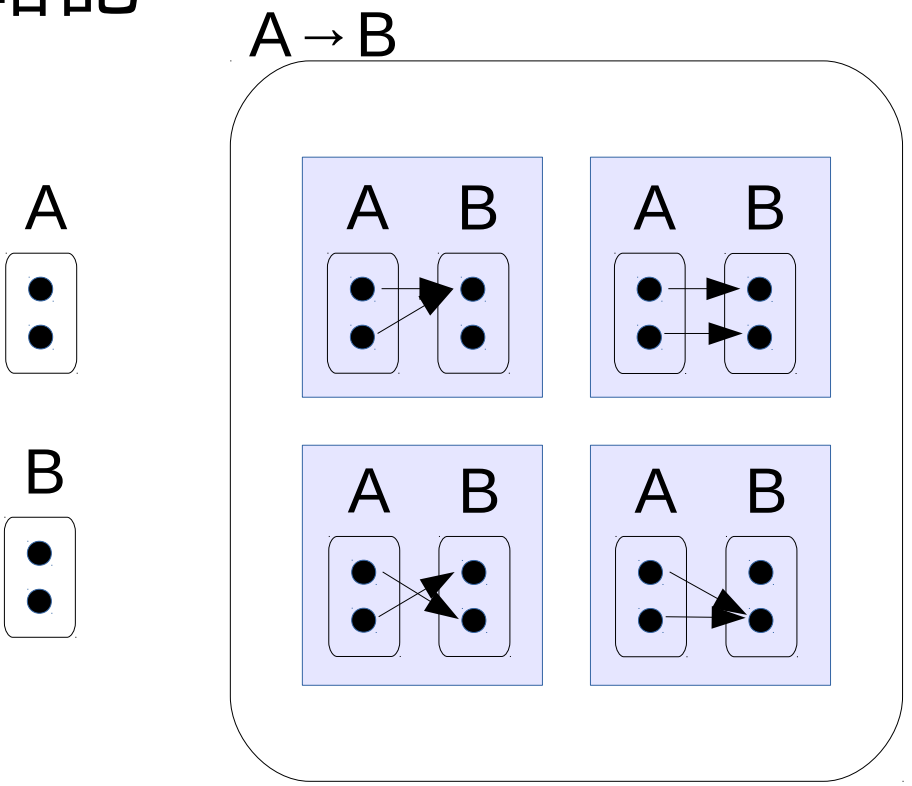
A /\ B (theories/Init/Logic.v)

- Inductive and (A B:Prop) : Prop :=
  conj : A -> B -> A /\ B
  where "A /\ B" := (and A B) : type_scope.
- 依存型ではない



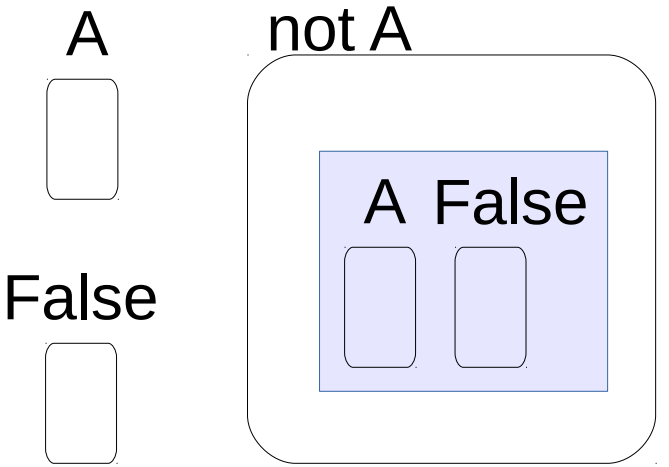
A→B

- A→B は関数型
- Inductive で定義されるものではない
- forall (_:A), B の略記
- 依存型ではない



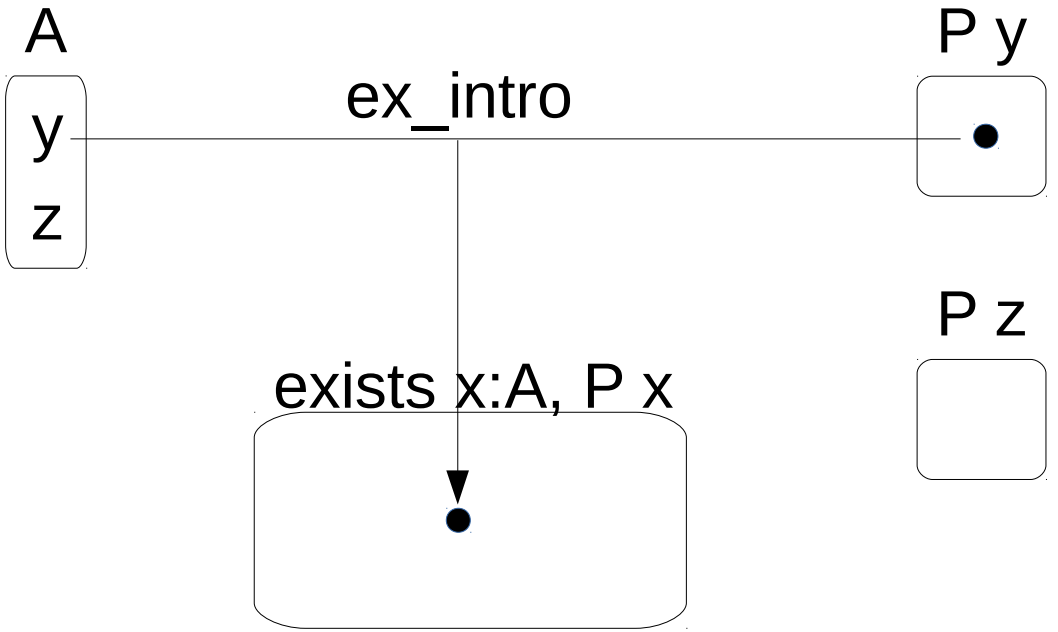
not (theories/Init/Logic.v)

- not は返り値が False 型の関数型
- Definition not (A:Prop) := A → False.
  Notation "~ x" := (not x) : type_scope.
- もし A に値 a が存在すると、A から False への関数は a を受け取って False の要素を返さなければならないが、False には値がないので無理
- したがって関数を作れるのは
  A が空のときだけ
- つまり not A が空でないなら
  A は空



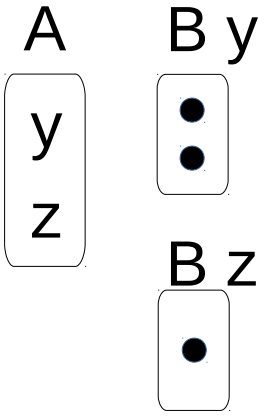
exists x:A, B x (theories/Init/Logic.v)

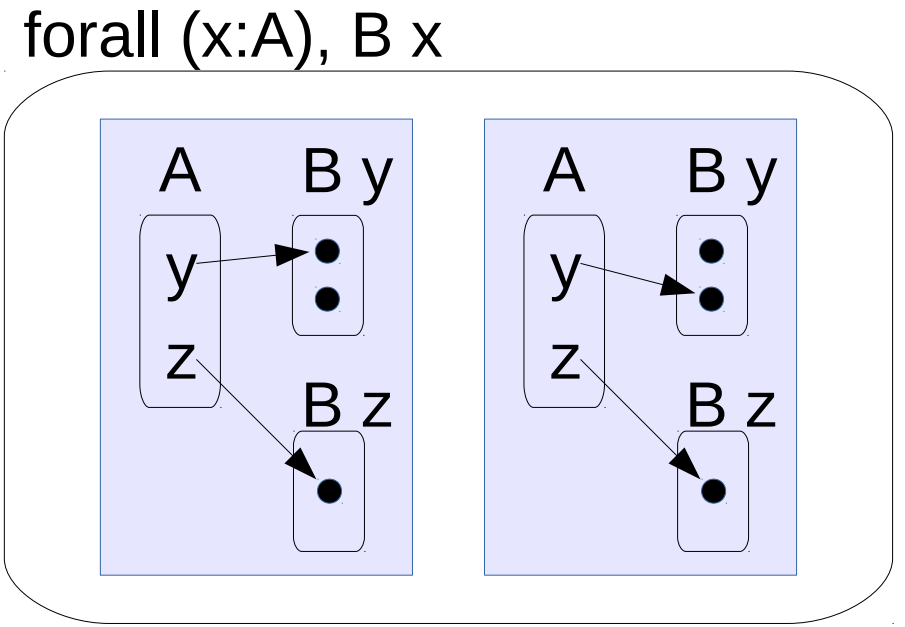
- ex (fun x:A => B x) の略記
- Inductive ex (A:Type) (P:A -> Prop) : Prop :=
  ex_intro : forall x:A, P x -> ex (A:=A) P.
- 依存型 (P は型ではなく値)



forall (x:A), B x

- 依存型の関数
- Inductive で定義されるものではない
- B x が空になる x があると、この関数は作れない
  だから、この関数が作れたとすれると、B x はすべて要素を持つ





述語論理と命題論理と依存型

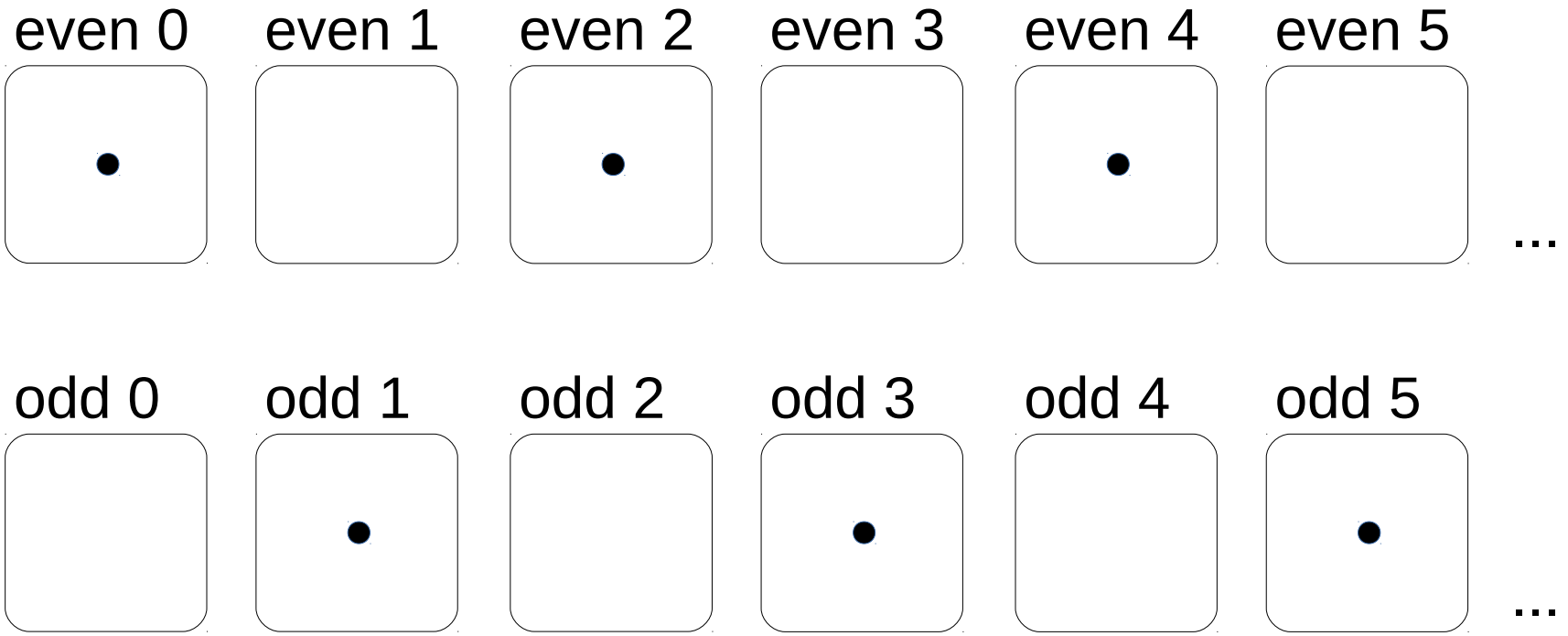
論理	Coq	
• 全称量化 $\forall x \in A.\ B(x)$	• forall (x:A), B x	述語論理 依存型
• 存在量化 $\exists x \in A.\ B(x)$	• exists x:A, B x	
• 含意 $A \supset B$	• A → B	命題論理 依存型でない
• 論理積 $A \wedge B$	• A /\ B	
• 論理和 $A \vee B$	• A \/ B	
• 真 $\top$	• True	
• 偽 $\bot$	• False	

述語論理における値についての命題

- 「nat の値 n について、n が偶数ならば、n+1 は奇数」
- forall (n:nat), 「n は偶数」→「n+1 は奇数」
- 「n は偶数」「n+1 は奇数」はどう書く？
- 前ページの書き方では書けない

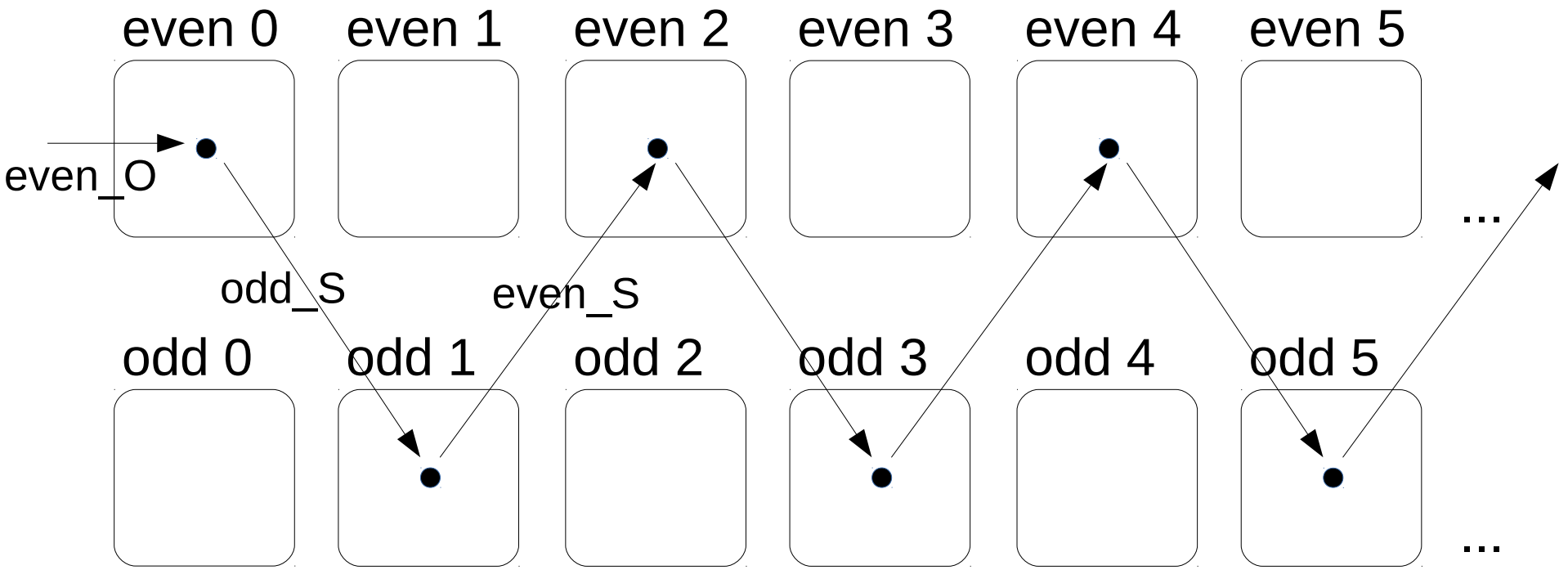
依存型による「n は偶数」「n は奇数」

- 「n は偶数」を even n とする
- 「n は奇数」を odd n とする
- 以下のような依存型を作ればよい



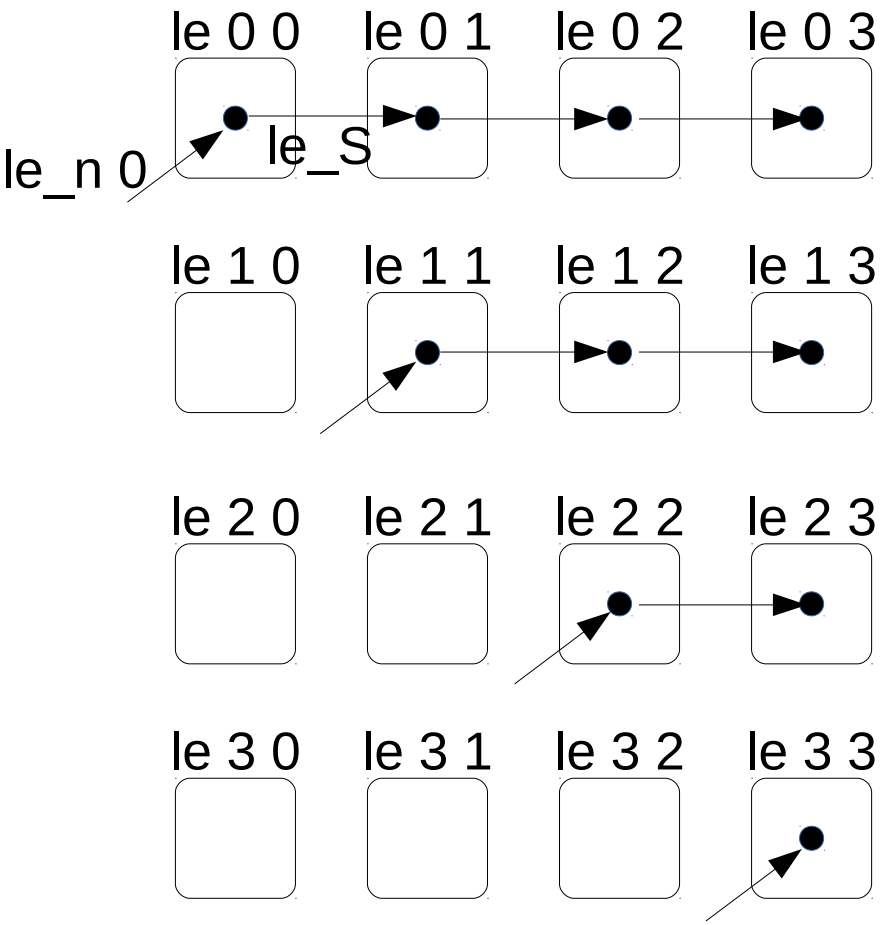
even (theories/Arith/Even.v)

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



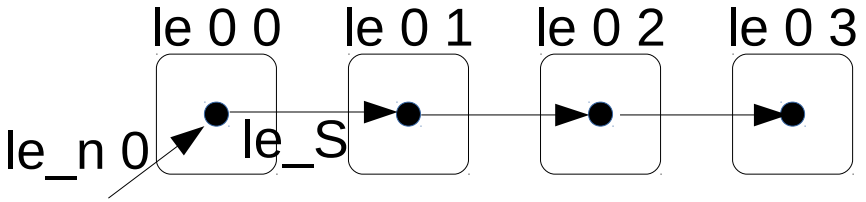
le (theories/Init/Peano.v)

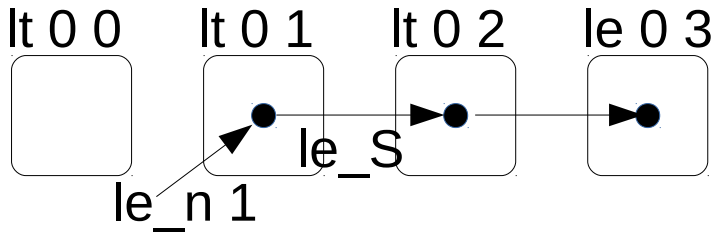
```
Inductive le (n:nat) : nat → Prop :=
  | le_n : n <= n
  | le_S : forall m:nat, n <= m → n <= S m
where "n <= m" := (le n m) : nat_scope.
```

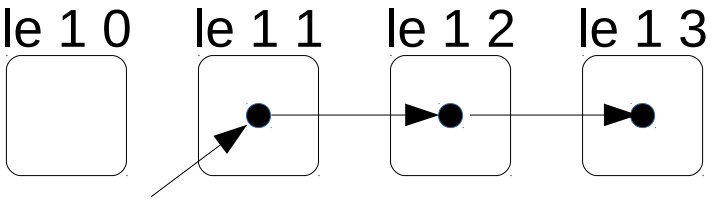


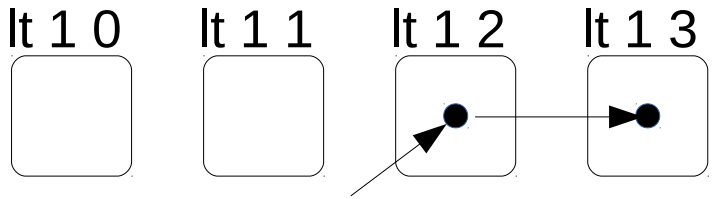
lt (theories/Init/Peano.v)

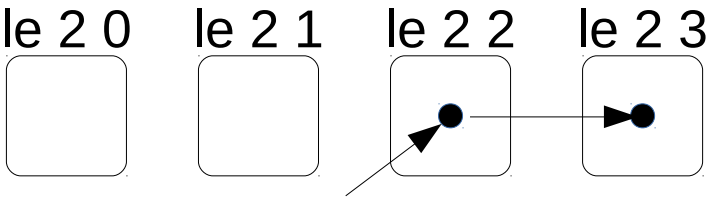
- Definition lt (n m:nat) := S n <= m.
  Infix "<" := lt : nat_scope.
- lt は le を利用して定義されている

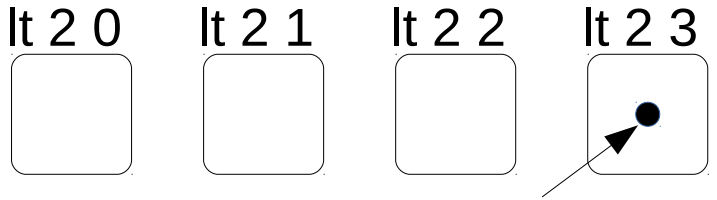


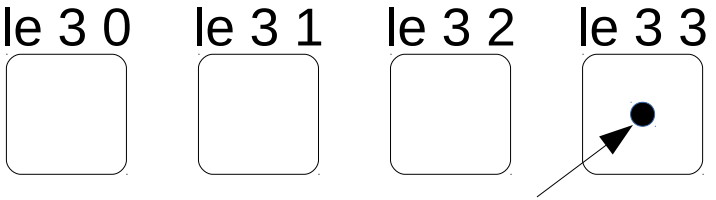






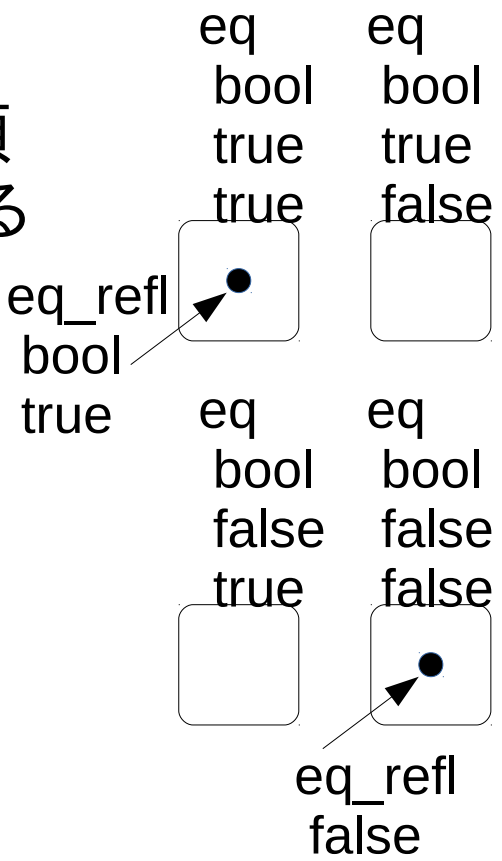


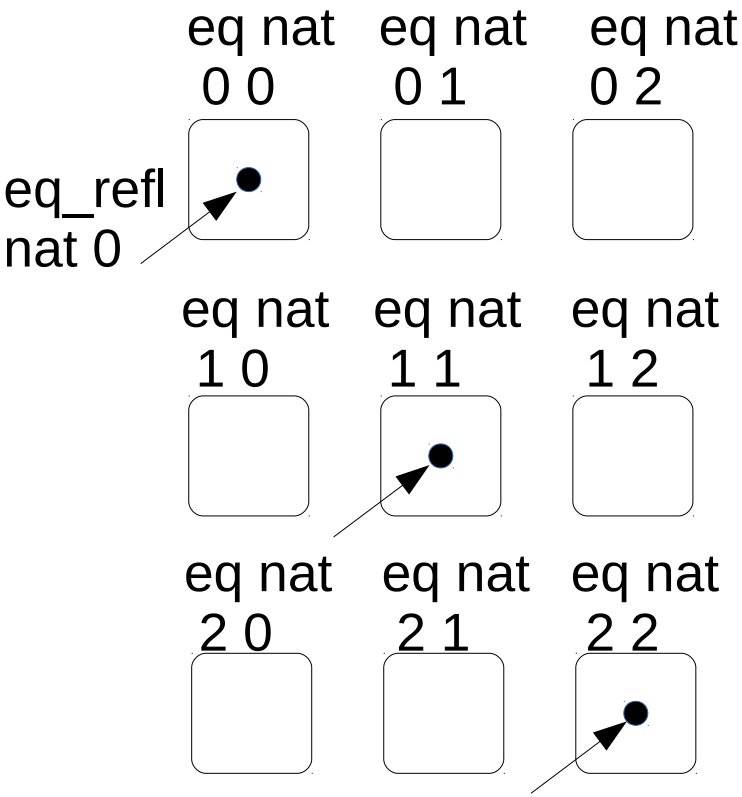




eq (theories/Init/Logic.v)

- Inductive eq (A:Type) (x:A) : A -> Prop :=
  eq_refl : x = x :>A
  where "x = y :> A" := (@eq A x y) : type_scope.
- eq は任意の型の等価性
- reflexivity は証明項としてeq_refl を作る
- eq_refl T x : eq T x y の型検査が通るのは x と y が convertible なとき



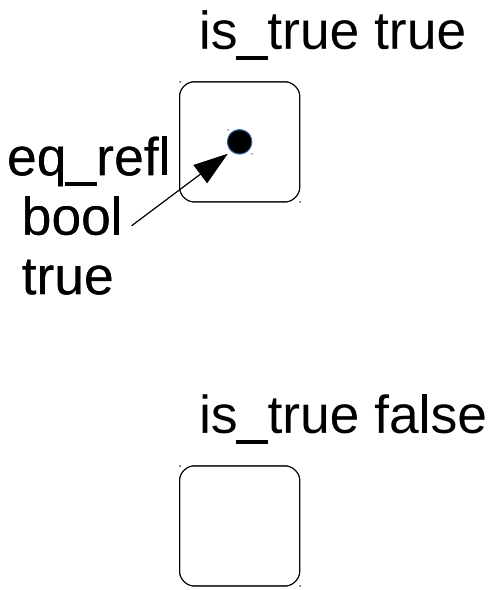


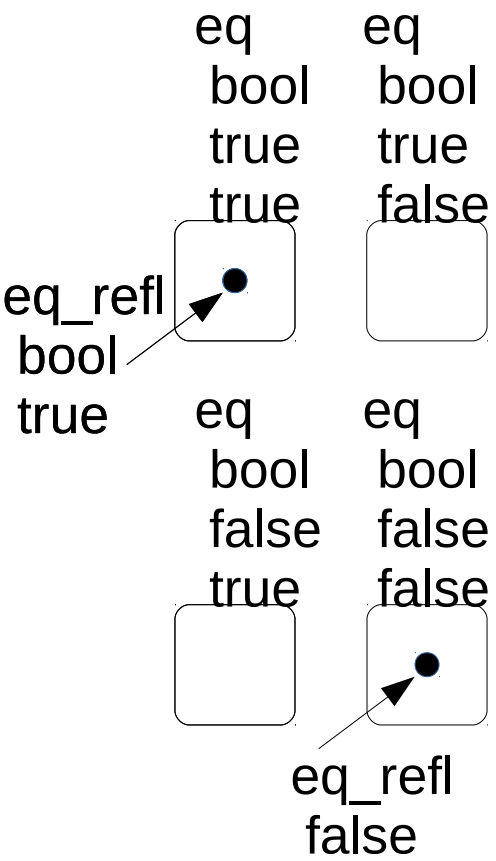
bool と計算を活用する (SSReflect)

- SSReflect では Prop よりもなるべく bool と計算を使う
- 計算は (convertible な範囲で) 自動的に行われるのでいろいろ楽
- Prop が必要なときには is_true : bool → Prop で暗黙に変換する

is_true (theories/Init/Datatypes.v)

- Definition is_true b := b = true.
- eq を使って定義されている





leq (ssrnat.v)

- Definition leq m n := m - n == 0.
  Notation "m <= n" := (leq m n) : nat_scope.
  Notation "m < n" := (m.+1 <= n) : nat_scope.
  Notation "m >= n" := (n <= m) (only parsing) : nat_scope.
  Notation "m > n" := (n < m) (only parsing) : nat_scope.
- is_true (leq m n) は le m n 相当
- ssrbool.v で is_true は Coercion として設定されているので、通常 is_true は書かなくてよい

odd (ssrnat.v)

- Fixpoint odd n := if n is n'.+1 then ~~ odd n' else false.
- is_true (odd n) は Even.odd n 相当

プログラムと証明

ゼロ除算が起きない除算

- Definition divc (x y : nat) (H : y <> 0) := x / y.
- y <> 0 は not (y = 0) の略記
- divc を呼び出すには y <> 0 の値 (証明) が必要
  y が 0 なら y <> 0 の値は存在しない
  y が 0 のときに y <> 0 型の式は書けない
  divc は y が 0 だと呼び出せない
- プログラム内で除算に divc を使っておけば、ゼロ除算は起きない
- (strict evaluation を仮定)

プログラムと証明をどう組み合わせるか

- 以下のプログラムで証明をどのように行うか
  - A→B 型のプログラム f
  - 引数 x:A についての条件 P x　　（事前条件）
  - 返値 y:B についての条件 Q x y　(事後条件）
- 案 1: 証明を別に書く
  - f : A→B のプログラムを書く
  - forall (x:A), P x → Q x (f x) を証明する
  - f を extraction して使う
- 案 2: 証明を混ぜて書く (divc みたいな書き方)
  - forall (x:A), P x → exists y:B, Q x y の証明 p を書く
  - p から extraction で証明を取り除き、f 相当のプログラムを得て使う

forall (x:A), P x → exists y:B, Q x y の証明からプログラムを抽出する

- exists y:B, Q x y は ex (fun y:B => Q x y) の略記
- Inductive ex (A:Type) (P:A -> Prop) : Prop :=
  ex_intro : forall x:A, P x -> ex (A:=A) P.
- exists y:B, Q x y の証明は
  ex_intro y (Q x y の証明) として構成する
- y は f の返値なので、証明を構成するときに返値を定義しておく必要がある
- extraction 結果の関数では証明を除去し、返値の部分だけを生成するようにする

プログラムに証明を混ぜるかどうか

- 案 1: 証明を別に書く
  - 証明しなくてもプログラムを動かせる
  - Coq の中でプログラムを実行できる
  - 複数の性質の証明を別々に書ける
  - 構成的証明にこだわる必要はない
- 案 2: 証明を混ぜて書く
  - 証明できない限りプログラムを動かせない
  - Coq の中では実行できないことになりがち
    (Qed で定義した定理は delta reduction できない)
  - 証明したいことはプログラムに最初からぜんぶ混ぜておくはめになりがち

プログラムに証明を混ぜるかどうか 個人的見解

- 非構造的再帰のために Acc を使う
  現在のところ、これはしょうがない
  （でもどうにかしてなくせないかな）
- 引数を制限する（事前条件）
  やらないほうがいいんじゃないかな
- 返値の性質を表現する（事後条件）
  やらないほうがいいんじゃないかな

停止性と再帰関数

- Coq において式はその型の値に対応していなければならない
- そのために関数は停止しなければならない
  停止しない関数があると、値を返さない式を記述可能になり、それは任意の型をつけても型エラーにならない
  型は命題なので、任意の命題を証明できるようになってしまう
- 停止性のため、無限再帰は禁止される
- そのため、再帰呼び出しでは特定の引数 (decreasing argument) がもとの引数の部分項になっていることが検査される
- 部分項をとっていく再帰を構造的再帰という

add は構造的再帰で停止する

- Fixpoint add n m :=
  match n with
  | 0 => m
  | S p => S (p + m)
  end
  where "n + m" := (add n m) : nat_scope.
- 第１引数 n が S p の場合、再帰呼び出しでは第１引数に p を渡している
- p は n の部分項なので再帰のたびに第１引数は小さくなる
- 帰納型の値は有限なのでそのうち停止する

非構造的再帰

- 構造的再帰では済まないこともある
  - クイックソート： リストを 2 分割するが、要素の並び順が変わるので分割結果は部分項ではない
- 非構造的再帰を使う方法がいくつかある
  - Function
  - Program Fixpoint
  - Fix (theories/Init/Wf.v)
- 構造的再帰にするための引数を追加する
  追加した引数のカインドは Prop にするため、
  extraction で消える

Acc (theories/Init/Wf.v)

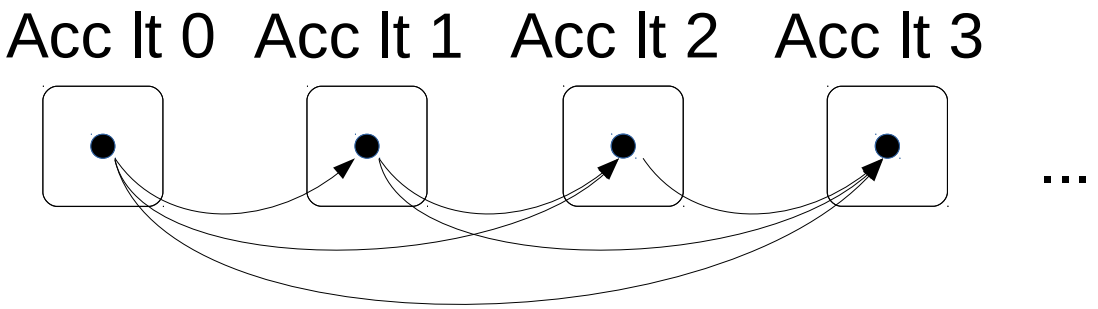
- 非構造的再帰を構造的再帰にするために追加する引数は通常 Acc 型を使う
- これで整礎帰納法を使えるようになる
- Variable A : Type.
  Variable R : A → A → Prop.
  Inductive Acc (x: A) : Prop :=
    Acc_intro : (forall y:A, R y x → Acc y) → Acc x.
- よく使うのは自然数の減少を利用する Acc lt
- コンストラクタ Acc_intro の引数は Acc y を返す関数
  - その関数は Acc x の部分項
  - その関数を呼び出して得られる Acc y 型の値も Acc x 型の値の部分項とみなされる
  - これで問題ないように、帰納的定義で定義しようとしている型をコンストラクタの引数に使う場合、書ける場所が制限されている (positivity condition)

Acc lt

- 自然数の完全帰納法を使うためのもの
  帰納段階 n で、n-1 だけでなく 0<m<n な任意の m の前提を使える
- Require Import Wf_nat.
  Check lt_wf. (* well_founded lt *)
  Check lt_wf 8. (* Acc lt 8 *)
  Definition lt_7_8 : 7 < 8 := le_n 8.
  Definition lt_5_8 : 5 < 8 := le_S 6 7 (le_S 6 6 (le_n 6)).
  Check match lt_wf 8 with Acc_intro _ H => H end _ lt_7_8.
  (* Acc lt 7 *)
  Check match lt_wf 8 with Acc_intro _ H => H end _ lt_5_8.
  (* Acc lt 5 *)
- Acc lt 8 の値から、Acc lt 7 の値と Acc lt 5 の値を取り出している
  この取り出しには 7<8 と 5<8 の証明が必要
  Acc lt 9 など大きいほうの値は取り出せない
- 普通は match を直接書くかわりに Acc_inv 関数を使う

Acc lt

- Acc lt 2 の値を作る場合には Acc lt 0 と Acc lt 1 の値を取り出せる関数をコンストラクタ Acc_intro に渡す
- 各型の値はひとつだけではない
- インデックスに値があるので依存型



非構造的再帰

- 構造的再帰では書けない関数
  Fail Fixpoint f v :=
    match v with
    | パターン 1 => 式
    | パターン 2 => ... (f u) ...
    end
- v, u の大きさを nat で測定する関数 g があり、
  g u < g u

Function による非構造的再帰

- Function f v {measure g v} :=
  match v with
  | パターン 1 => 式
  | パターン 2 => ... (f u) ...
  end.
  Proof. g u < g v の証明 . Defined.
- 証明では v = パターン 2 の証明を利用できる

Acc を直接使った非構造的再帰

- Fixpoint f v (a : Acc R v) : T :=
  match v as v' return v = v' → T with
  | パターン 1 => fun _ => 式
  | パターン 2 => fun (H : v = パターン 2) =>
  ... (f u (Acc_inv a (R (g u) (g v) の証明))) ...
  end eq_refl.
- 証明では v = パターン 2 の証明を利用できる

dependent match

dependent match (return 節を使う match) を使う用途はいくつかある

- 非構造的再帰では match でどの分岐を選んだかを示す証明が必要
- 絶対に使われない式の型を任意に決めたい
- 型を計算で求める類の依存型の値を返す
- 型を計算で求める類の依存型の値を使う
- もっとあるかも？

match でどの分岐を選んだかを示す証明が欲しい

- 非構造的再帰で Acc の証明を作るために必要になる
- match e as x
  return e = x -> T
  with ... | C ... => fun v' => ... | ... end
  (eq_refl e)
- match は e=e の証明を受け取る関数を返し、その関数を即座に呼び出す
- tactic なら case_eq 相当

絶対に使われない式の型を任意に決めたい

- 異なるコンストラクタC1,C2が等しいという型C1=C2の証明Hがある場合、そこには絶対に処理が到達しない
- でも、Coqでは必ず型に合った式を書かなければならない
- 型が型パラメタTの場合、その型の値は（呼び出し元から渡されていない限り）式として記述できない
- という場合にdependent matchを使うとT型の式を記述できる
- match H as x
  return match x with C1 => unit | C2 => T end
  with eq_refl => tt end
- Falseを経由することも多い
  match H as x
  return match x with C1 => True | C2 => False end
  with eq_refl => I end
- tacticならdiscriminate相当

型を計算して求める類の依存型の値を返したい

- Definition D (b : bool) : Set :=
  if b then nat else unit.
- D b 型とすれば正しいはずだが、Coq は見抜いてくれない
  Fail Definition f b :=
  match b with
  | true => 0
  | false => tt
  end.
- return 節に D b と書けば Coq は正しく確認してくれる
  Definition f b :=
  match b return D b with
  | true => 0
  | false => tt
  end.

型を計算で求める類の依存型の値を使う

- natとunitのコンストラクタを混ぜて使うのはもちろん無理

```
Fail Definition g (b : bool) (d : D b) :=
  match d with
  | O => 1
  | S n => n+1
  | tt => 0
  end.
```

- いったんbで分岐してもそれだけではだめ

```
Fail Definition g (b : bool) (d : D b) :=
  match b with
  | true => match d with
              | O => 1
              | S n => n+1
              end
  | false => match d with
               | tt => 0
               end
  end.
```

- bで分岐するときに、dを受け取る関数を返すようにして、その関数を即座に呼ぶとうまくいく

```
Definition g (b : bool) (d : D b) :=
  match b as b return D b -> nat
with
  | true => fun d' =>
              match d' with
              | O => 1
              | S n => n+1
              end
  | false => fun d' =>
              match d' with
              | tt => 0
              end
  end d.
```

dependent match の仕組み

- Inductive D : nat → Set :=
  C : forall (n:nat), D (f n).
- v : D m とする
- v は C n として作られているはず
  - v = C n でなければならない
  - C n : D (f n) なので、m = f n でなければならない
- match v as v' in D n'
  return (v' と n' を使った型 T) with
  C n => 式
  end
- match 全体の型は T[v':=v, n':=m]
- match 内部の分岐の「式」の型は T[v':=C n, n':=n]
- 一般には複数のコンストラクタがあり、それぞれの分岐では対応するコンストラクタの定義に従う型になる

dependent match の書き方がよくわからない場合

- proof editing mode で tactic に項を作らせる
  - Definition f. Proof. refine (... _ ...). ... Defined.
    refine で項を与えるが、書き方がわからにところは _ にする
    _ の部分は goal になるので tactic で作る
  - discriminate, case_eq, inversion など、dependent match を生成するいろいろな tactic がある
- Show Proof. で証明項を見る
- (とくに SSReflect の場合)
  Eval cbv beta zeta delta [f] in f.
  として証明項を簡約して表示させる
  (delta reduction で f の定義を展開する必要があるので、Qed じゃなくて Defined で proof editing mode を終わる必要がある)

heterogeneous list を作る

- 各要素の型が異なっていてもよいリスト
- 案 1: 帰納型で定義し、要素を追加するたびに型を与える
- 案 2: 型のリストを最初に与えて、実際の型を計算して求める
- ここでは後者を考える

型のリストから prod のネストに変換する

- From mathcomp Require Import all_ssreflect.
- 型のリスト
  Definition hltype := seq Set.
- Definition htnth (ht : hltype) (i : nat) := nth unit ht i.
- Definition ht_nil := @nil Set.
- Definition ht_cons := @cons Set.
- 型のリストをネストしたprodに変換する
  Fixpoint hlist (ht : hltype) :=
  match ht with
  | [::] => unit
  | T :: ht' => prod T (hlist ht')
  end.
- Check (1,(true,tt)) : hlist [:: nat; bool].

heterogeneous list の生成

- Definition h_nil : hlist [::] := tt.
- Definition h_cons (T : Set) (v : T)
  (ht : hltype) (hs : hlist ht) : hlist (T :: ht) :=
  (v, hs).
- 以下のように、h_nil, h_cons で heterogeneous list を生成できる
  Check h_nil : hlist ht_nil.
  Check h_cons bool true [::] h_nil.
  Check h_cons bool true _ (h_cons nat 0 _ h_nil).
- h_nil, h_cons で生成するのは面倒なので Notation を定義する
  Notation "['hlist:']" := h_nil.
  Notation "['hlist:' x1 ; .. ; xn]" :=
  (h_cons _ x1 _ (.. (h_cons _ xn _ h_nil) ..)).
- Notation により、簡単に生成できる
  Check [hlist: true; 0; (1,2)].

heterogeneous list の lookup

- i 番目の要素を取り出す
- Fixpoint hlookup (ht : hltype) (hs : hlist ht) (i : nat) : htnth ht i :=
  match i with
  | 0 =>
    match ht as ht return hlist ht -> htnth ht 0 with
    | [::] => fun hs => tt
    | T :: ht' => fun hs => hs.1
    end hs
  | i'.+1 =>
    match ht as ht return hlist ht -> htnth ht i'.+1 with
    | [::] => fun hs => tt
    | T :: ht' => fun hs => hlookup ht' hs.2 i'
      end hs
  end.
- Compute hlookup [:: bool; nat] [hlist: true; 0] 0. (* true *)
  Compute hlookup [:: bool; nat] [hlist: true; 0] 1. (* 0 *)

まとめ

- 依存型の噂
  - すごいらしい→アイデア自体は難しくない
  - 証明できるらしい→できる
  - リストの長さを型で扱えるらしい→個人的には勧めない
  - なんか面倒くさいらしい→ Yes