Pattern covering by set approximations

Nicolas Oury

Laboratoire de Recherche en Informatique
Université Paris–Sud, France

TYPES, 2006

Outline

Outline

The Calculus of Inductive Constructions

- Proof theory used in the Coq proof assistant
- Proving is typing a proof term
- Dependent inductive data types: *list n*...

The Calculus of Inductive Constructions

- Proof theory used in the Coq proof assistant
- Proving is typing a proof term
- Dependent inductive data types: *list n* . . .

Inductive data types

- Types defined by different constructors :

```
nat =
  O : nat
  S : nat → nat
```

- Values are constructed inductively: `O`, `S O`, `S (S O)`, `...`
- Elements are finite: `x = S x` is forbidden
- Dependent types:

```
list _ =
  nil  : list O
  cons : A → list n → list (S n)
```

Inductive data types

- Types defined by different constructors :

```
nat =
  O : nat
  S : nat → nat
```

- Values are constructed inductively: `O`, `S O`, `S (S O), ...`
- Elements are finite: `x = S x` is forbidden
- Dependent types:

```
list _ =
  nil  : list O
  cons : A → list n → list (S n)
```

Inductive data types

- Types defined by different constructors :

```
nat =
  O : nat
  S : nat → nat
```

- Values are constructed inductively: `O`, `S O`, `S (S O), ...`
- Elements are finite: `x = S x` is forbidden
- Dependent types:

```
list _  =
  nil   : list O
  cons  : A → list n → list (S n)
```

Pattern matching

- Functions can be defined by pattern matching

```
plus O     n = n
plus (S m) n = S (plus m n)
```

- With dependent types

```
append :: list n → list m → list (n +m)
append nil         l = l
append (cons a l') l = cons a (append l' l)
```

Pattern matching

- Functions can be defined by pattern matching

```
plus O     n = n
plus (S m) n = S (plus m n)
```

- With dependent types

```
append :: list n → list m → list (n +m)
append nil         l = l
append (cons a l′) l = cons a (append l′ l)
```

Useless cases

- Another example :

```
head :: list (S n) → A
head (cons a _) = a
head nil = ???
```

- What do we want to write here?
  - A default case?
  - A proof that the case is impossible?
- We want to automaticaly eliminate these cases

Useless cases

- Another example :

```
head :: list (S n) → A
head (cons a _) = a
head nil = ???
```

- What do we want to write here?
  - A default case?
  - A proof that the case is impossible?
- We want to automaticaly eliminate these cases

Useless cases

- Another example :

  ```
  head :: list (S n) → A
  head (cons a _) = a
  head nil = ???
  ```

- What do we want to write here?
  - A default case?
  - A proof that the case is impossible?
- We want to automaticaly eliminate these cases

Useless cases

- Another example :

```
head :: list (S n) → A
head (cons a _) = a
head nil = ???
```

- What do we want to write here?
  - A default case?
  - A proof that the case is impossible?
- We want to automaticaly eliminate these cases

Useless cases

- Another example :

```
head :: list (S n) → A
head (cons a _) = a
head nil = ???
```

- What do we want to write here?
  - A default case?
  - A proof that the case is impossible?
- We want to automaticaly eliminate these cases

Useless cases

- Another example :

```
head :: list (S n) → A
head (cons a _) = a
head nil = ???
```

- What do we want to write here?
  - A default case?
  - A proof that the case is impossible?
- We want to automaticaly eliminate these cases

Outline

Undecidability

- Post problem
  - $(u_1, v_1) \ldots (u_n, v_n)$ words on $\{a; b\}$
  - $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$ for some non empty $(i_j)_{1 \leq j \leq k}$?
  - This problem is undecidable
- Encoding words :

```
Word =
  ϵ : Word
  A   : Word → Word
  B   : Word → Word
```

- To each word we asociate a context:

$$\overline{abb}[] = \texttt{A(B(B[]))}$$

Undecidability

- Post problem
  - $(u_1, v_1) \ldots (u_n, v_n)$ words on $\{a; b\}$
  - $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$ for some non empty $(i_j)_{1 \leq j \leq k}$?
  - This problem is undecidable
- Encoding words :

```
Word =
  ϵ : Word
  A   : Word → Word
  B   : Word → Word
```

- To each word we asociate a context:

$$\overline{abb}[] = \mathtt{A}(\mathtt{B}(\mathtt{B}[]))$$

Undecidability

- Post problem
  - $(u_1, v_1) \ldots (u_n, v_n)$ words on $\{a; b\}$
  - $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$ for some non empty $(i_j)_{1 \leq j \leq k}$?
- Encoding Post problem in pattern matching covering :

```
I _ _ =
 init : I ϵ  ϵ
 u1v1 : I u v → I u1[u] v1[v]
...
 unvn : I u v → I un[u] vn[v]
```

- Is this function total?

```
f :: I w w → nat
f init = O
```

Undecidability

- Post problem
  - $(u_1, v_1) \ldots (u_n, v_n)$ words on $\{a; b\}$
  - $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$ for some non empty $(i_j)_{1 \leq j \leq k}$?
- Encoding Post problem in pattern matching covering :

```
I _ _ =
 init : I ϵ   ϵ
 u1v1 : I u v → I u1[u] v1[v]
...
 unvn : I u v → I un[u] vn[v]
```

- Is this function total?

```
f :: I w w → nat
f init = O
```

Splitting

- Split inductive types along their constructors.
- Unification to eliminate cases.

```
head :: list (S n) → A
head (cons a _) = a
head nil = ???
```

- `list (S n)` splits into :
  - `cons` ⇒ n : nat, a : A, l : `list n` ⊢ `cons` a l : `list (S n)`
  - `nil` ⇒ ⊢ nil : `list O`
- First case generate a new goal : `list n`
- The second case is impossible : `S n = O`
- Epigram, Alf, Twelf . . .

Splitting

- Split inductive types along their constructors.
- Unification to eliminate cases.

```
head :: list (S n) → A
head (cons a _) = a
head nil = ???
```

- `list (S n)` splits into :
  - `cons` ⇒ n : nat, a : A, l : `list n` ⊢ `cons` a l : `list (S n)`
  - `nil` ⇒ ⊢ nil : `list O`
- First case generate a new goal : `list n`
- The second case is impossible : `S n = O`
- Epigram, Alf, Twelf . . .

Splitting

- Split inductive types along their constructors.
- Unification to eliminate cases.

```
head :: list (S n) → A
head (cons a _) = a
head nil = ???
```

- `list (S n)` splits into :
  - `cons` ⇒ n : nat, a : A, l : `list n` ⊢ `cons` a l : `list (S n)`
  - `nil` ⇒ ⊢ nil : `list O`
- First case generate a new goal : `list n`
- The second case is impossible : `S n = O`
- Epigram, Alf, Twelf . . .

Splitting

- Split inductive types along their constructors.
- Unification to eliminate cases.

```
head :: list (S n) → A
head (cons a _) = a
head nil = ???
```

- `list (S n)` splits into :
  - `cons` ⇒ n : nat, a : A, l : `list n` ⊢ `cons` a l : `list (S n)`
  - `nil` ⇒ ⊢ nil : `list O`
- First case generate a new goal : `list n`
- The second case is impossible : `S n = O`
- Epigram, Alf, Twelf . . .

Splitting

- Split inductive types along their constructors.
- Unification to eliminate cases.

```
head :: list (S n) → A
head (cons a _) = a
head nil = ???
```

- `list (S n)` splits into :
  - `cons` ⇒ n : nat, a : A, l : `list n` ⊢ `cons` a l : `list (S n)`
  - `nil` ⇒ ⊢ nil : `list O`
- First case generate a new goal : `list n`
- The second case is impossible : `S n = O`
- Epigram, Alf, Twelf . . .

Splitting does not use finiteness

```
empty =
 useless : empty → empty
```

- `empty` splits into `useless`
  ⇒ we have to show `empty` is empty

```
R _ _ =
 R1    : R 0 1
 R2    : R 0 2
 Trans : R n p → R p m → R n m
```

- We want to show `Trans` is not accessible.
- First goal : { `R n p; R p m`}
- Splits into : { `R n p'; R p' p; R p m`}

Splitting does not use finiteness

```
empty =
 useless : empty → empty
```

- `empty` splits into `useless`
  ⇒ we have to show `empty` is empty

```
R _ _ =
 R1    : R 0 1
 R2    : R 0 2
 Trans : R n p → R p m → R n m
```

- We want to show `Trans` is not accessible.
- First goal : `{ R n p; R p m}`
- Splits into : `{ R n p'; R p' p; R p m}`

Splitting does not use finiteness

```
empty =
 useless : empty → empty
```

- `empty` splits into `useless`
  ⇒ we have to show `empty` is empty

```
R _ _ =
 R1    : R 0 1
 R2    : R 0 2
 Trans : R n p → R p m → R n m
```

- We want to show `Trans` is not accessible.
- First goal : { `R n p`; `R p m`}
- Splits into : { `R n p'`; `R p' p`; `R p m`}

Splitting does not use finiteness

```
empty =
 useless : empty → empty
```

- `empty` splits into `useless`
  ⇒ we have to show `empty` is empty

```
R _ _ =
 R1    : R 0 1
 R2    : R 0 2
 Trans : R n p → R p m → R n m
```

- We want to show `Trans` is not accessible.
- First goal : { `R n p; R p m`}
- Splits into : { `R n p'; R p' p; R p m`}

Splitting does not use finiteness

```
empty =
 useless : empty → empty
```

- `empty` splits into `useless`
  ⇒ we have to show `empty` is empty

```
R _ _ =
 R1    : R 0 1
 R2    : R 0 2
 Trans : R n p → R p m → R n m
```

- We want to show `Trans` is not accessible.
- First goal : { `R n p; R p m`}
- Splits into : { `R n p'; R p' p; R p m`}

Outline

Computing the set of inhabitants

- Inductive types are least fixpoints so we iterate

```
empty =
 useless : empty → empty
```

- $\mathtt{empty}_0 = \emptyset$
- Applying `useless` to each elements of $\mathtt{empty}_0$ gives : $\mathtt{empty}_1 = \emptyset$

```
nat =
 O : nat
 S : nat → nat
```

$\mathtt{nat}_0 = \emptyset$, $\mathtt{nat}_1 = \{0\}$, $\mathtt{nat}_2 = \{0; 1\}$
$\mathtt{nat}_3 = \{0; 1; 2\}$, $\mathtt{nat}_4 = \{0; 1; 2; 3\}$, $\mathtt{nat}_5 = \{0; 1; 2; 3; 4\}$,
...

Computing the set of inhabitants

- Inductive types are least fixpoints so we iterate

```
empty =
 useless : empty → empty
```

- $\texttt{empty}_0 = \emptyset$
- Applying `useless` to each elements of $\texttt{empty}_0$ gives : $\texttt{empty}_1 = \emptyset$

```
nat =
 O : nat
 S : nat → nat
```

$\texttt{nat}_0 = \emptyset$, $\texttt{nat}_1 = \{0\}$, $\texttt{nat}_2 = \{0; 1\}$
$\texttt{nat}_3 = \{0; 1; 2\}$, $\texttt{nat}_4 = \{0; 1; 2; 3\}$, $\texttt{nat}_5 = \{0; 1; 2; 3; 4\}$,
...

Computing the set of inhabitants

- Inductive types are least fixpoints so we iterate

```
empty =
 useless : empty → empty
```

- $\texttt{empty}_0 = \emptyset$
- Applying `useless` to each elements of $\texttt{empty}_0$ gives : $\texttt{empty}_1 = \emptyset$

```
nat =
 O : nat
 S : nat → nat
```

$\texttt{nat}_0 = \emptyset$, $\texttt{nat}_1 = \{0\}$, $\texttt{nat}_2 = \{0; 1\}$

$\texttt{nat}_3 = \{0; 1; 2\}$, $\texttt{nat}_4 = \{0; 1; 2; 3\}$, $\texttt{nat}_5 = \{0; 1; 2; 3; 4\}$,

...

Computing the set of inhabitants

- Inductive types are least fixpoints so we iterate

```
empty =
 useless : empty → empty
```

- $\mathtt{empty}_0 = \emptyset$
- Applying `useless` to each elements of $\mathtt{empty}_0$ gives :
  $\mathtt{empty}_1 = \emptyset$

```
nat =
 O : nat
 S : nat → nat
```

$\mathtt{nat}_0 = \emptyset$, $\mathtt{nat}_1 = \{0\}$, $\mathtt{nat}_2 = \{0; 1\}$
$\mathtt{nat}_3 = \{0; 1; 2\}$, $\mathtt{nat}_4 = \{0; 1; 2; 3\}$, $\mathtt{nat}_5 = \{0; 1; 2; 3; 4\}$,
...

Approximations of sets

- We work on over–approximations in domains where fixpoints converge.
- For example, $\mathtt{nat}_\infty = \{\perp\}$
- We test if the over–approximation is empty.
- Each construction must be reflected on the approximated sets.
- We only consider to monomorph first order inductives.
- We approximate dependent inductives by the set of terms with dependencies.

$$\mathrm{R}_\infty = \{(0, 1, \mathrm{R1}); (0, 2, \mathrm{R2})\}$$

Approximations of sets

- We work on over–approximations in domains where fixpoints converge.
- For example, $\mathtt{nat}_\infty = \{\bot\}$
- We test if the over–approximation is empty.
- Each construction must be reflected on the approximated sets.
- We only consider to monomorph first order inductives.
- We approximate dependent inductives by the set of terms with dependencies.

$$\mathrm{R}_\infty = \{(0, 1, \mathrm{R1}); (0, 2, \mathrm{R2})\}$$

Approximations of sets

- We work on over–approximations in domains where fixpoints converge.
- For example, $\mathtt{nat}_\infty = \{\perp\}$
- We test if the over–approximation is empty.
- Each construction must be reflected on the approximated sets.
- We only consider to monomorph first order inductives.
- We approximate dependent inductives by the set of terms with dependencies.

$$\mathrm{R}_\infty = \{(0, 1, \mathrm{R1}); (0, 2, \mathrm{R2})\}$$

Approximations of sets

- We work on over–approximations in domains where fixpoints converge.
- For example, $\mathtt{nat}_\infty = \{\perp\}$
- We test if the over–approximation is empty.
- Each construction must be reflected on the approximated sets.
- We only consider to monomorph first order inductives.
- We approximate dependent inductives by the set of terms with dependencies.

$$\mathrm{R}_\infty = \{(0, 1, \mathrm{R1}); (0, 2, \mathrm{R2})\}$$

Approximations of sets

- We work on over–approximations in domains where fixpoints converge.
- For example, $\mathtt{nat}_\infty = \{\bot\}$
- We test if the over–approximation is empty.
- Each construction must be reflected on the approximated sets.
- We only consider to monomorph first order inductives.
- We approximate dependent inductives by the set of terms with dependencies.

$$\mathrm{R}_\infty = \{(0, 1, \mathrm{R1}); (0, 2, \mathrm{R2})\}$$

Example of approximation

```
R n m =
 R1     : R 0 1
 R2     : R 0 2
 Trans  : R n p → R p m → R n m
```

- $R_1 = \{(0, 1, R1); (0, 2, R2)\}$
- We approximate the context of `Trans`

```
n, m ,p : nat        n, m, p ∈ nat_∞
t1 : R n p   (t1,n,p)∈{(R1,0,1);  (R2,0,2)}
t2 : R p m   (t2,p,m)∈{(R1,0,1);  (R2,0,2)}
```

- p is in both $\{0\}$ and $\{1; 2\} \Rightarrow$ `Trans` can't be applyied.

$R_\infty = R_1$

Example of approximation

```
R n m =
 R1     : R 0 1
 R2     : R 0 2
 Trans  : R n p → R p m → R n m
```

- $\mathtt{R}_1 = \{(0, 1, R1); (0, 2, R2)\}$
- We approximate the context of `Trans`

```
n, m ,p : nat        n, m, p ∈ nat_∞
t1 : R n p    (t1,n,p)∈{(R1,0,1);  (R2,0,2)}
t2 : R p m    (t2,p,m)∈{(R1,0,1);  (R2,0,2)}
```

- p is in both $\{0\}$ and $\{1; 2\}$ $\Rightarrow$ `Trans` can't be applyed.

$\mathtt{R}_\infty = \mathtt{R}_1$

Example of approximation

```
R n m =
 R1    : R 0 1
 R2    : R 0 2
 Trans : R n p → R p m → R n m
```

- $\mathtt{R}_1 = \{(0, 1, R1); (0, 2, R2)\}$
- We approximate the context of `Trans`

```
n, m ,p : nat        n, m, p ∈ nat_∞
t1 : R n p   (t1,n,p)∈{(R1,0,1); (R2,0,2)}
t2 : R p m   (t2,p,m)∈{(R1,0,1); (R2,0,2)}
```

- p is in both $\{0\}$ and $\{1; 2\}$ $\Rightarrow$ `Trans` can't be applyied.

  $\mathtt{R}_\infty = \mathtt{R}_1$

Example of approximation

```
R n m =
 R1     : R 0 1
 R2     : R 0 2
 Trans  : R n p → R p m → R n m
```

- $\mathtt{R}_1 = \{(0, 1, R1); (0, 2, R2)\}$
- We approximate the context of `Trans`

```
n, m ,p : nat       n, m, p ∈ nat_∞
t1 : R n p   (t1,n,p)∈{(R1,O,1);  (R2,O,2)}
t2 : R p m   (t2,p,m)∈{(R1,O,1);  (R2,O,2)}
```

- p is in both $\{0\}$ and $\{1; 2\}$ $\Rightarrow$ `Trans` can't be applyied.

$\mathtt{R}_\infty = \mathtt{R}_1$

Example of approximation

```
R n m =
 R1    : R 0 1
 R2    : R 0 2
 Trans : R n p → R p m → R n m
```

- $\texttt{R}_1 = \{(0, 1, R1); (0, 2, R2)\}$
- We approximate the context of `Trans`

```
n, m ,p : nat        n, m, p ∈ nat_∞
t1 : R n p    (t1,n,p)∈{(R1,0,1); (R2,0,2)}
t2 : R p m    (t2,p,m)∈{(R1,0,1); (R2,0,2)}
```

- p is in both $\{0\}$ and $\{1; 2\}$ $\Rightarrow$ `Trans` can't be applyied.

$\texttt{R}_\infty = \texttt{R}_1$

Example of approximation

```
R n m =
 R1    : R 0 1
 R2    : R 0 2
 Trans : R n p → R p m → R n m
```

- $\mathtt{R}_1 = \{(0, 1, R1); (0, 2, R2)\}$
- We approximate the context of `Trans`

  ```
  n, m ,p : nat        n, m, p ∈ nat_∞
  t1 : R n p   (t1,n,p)∈{(R1,0,1); (R2,0,2)}
  t2 : R p m   (t2,p,m)∈{(R1,0,1); (R2,0,2)}
  ```
- p is in both $\{0\}$ and $\{1; 2\}$ $\Rightarrow$ `Trans` can't be applyied.

  $\mathtt{R}_\infty = \mathtt{R}_1$

Another example of approximation

```
le n m =
 eq    : le n n
 trans : le n m -> le n (S m)
```

- Counting the number of occurences of constructors

  $\mathtt{le}_O = \emptyset, \mathtt{le}_1 = [|n|_O = 1; |m|_O = 1; |n|_S = |m|_S]$
- We approximate the context of `trans`

  ```
  t : le n m  [|n|_O=1;|m|_O=1;|n|_S=|m|_S]
  ```

  $\mathtt{le}_2 = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S; |n|_S + 1 \geq |m|_S] \ldots$

  $\mathtt{le}_k = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S; |n|_S + k \geq |m|_S]$
- And with *acceleration*.

  $\mathtt{le}_\infty = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S]$

Another example of approximation

```
le n m =
 eq    : le n n
 trans : le n m -> le n (S m)
```

- Counting the number of occurences of constructors
  $\texttt{le}_o = \emptyset, \texttt{le}_1 = [|n|_O = 1; |m|_O = 1; |n|_S = |m|_S]$
- We approximate the context of `trans`

  ```
  t : le n m   [|n|_O=1;|m|_O=1;|n|_S=|m|_S]
  ```

  $\texttt{le}_2 = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S; |n|_S + 1 \geq |m|_S] \ldots$
  $\texttt{le}_k = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S; |n|_S + k \geq |m|_S]$
- And with *acceleration*.
  $\texttt{le}_\infty = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S]$

Another example of approximation

```
le n m =
 eq    : le n n
 trans : le n m -> le n (S m)
```

- Counting the number of occurences of constructors
  $\mathtt{le}_o = \emptyset, \mathtt{le}_1 = [|n|_O = 1; |m|_O = 1; |n|_S = |m|_S]$
- We approximate the context of `trans`

  `t : le n m  [|n|_O=1;|m|_O=1;|n|_S=|m|_S]`

  $\mathtt{le}_2 = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S; |n|_S + 1 \geq |m|_S] \ldots$
  $\mathtt{le}_k = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S; |n|_S + k \geq |m|_S]$
- And with *acceleration*.
  $\mathtt{le}_\infty = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S]$

Another example of approximation

```
le n m =
 eq    : le n n
 trans : le n m -> le n (S m)
```

- Counting the number of occurences of constructors
  $\mathtt{le}_o = \emptyset, \mathtt{le}_1 = [|n|_O = 1; |m|_O = 1; |n|_S = |m|_S]$
- We approximate the context of `trans`

  `t : le n m  [|n|_O=1;|m|_O=1;|n|_S=|m|_S]`

  $\mathtt{le}_2 = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S; |n|_S + 1 \geq |m|_S] \dots$
  $\mathtt{le}_k = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S; |n|_S + k \geq |m|_S]$
- And with *acceleration*.
  $\mathtt{le}_\infty = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S]$

Another example of approximation

```
le n m =
 eq     : le n n
 trans  : le n m -> le n (S m)
```

- Counting the number of occurences of constructors
  $\mathtt{le}_o = \emptyset, \mathtt{le}_1 = [|n|_O = 1; |m|_O = 1; |n|_S = |m|_S]$
- We approximate the context of `trans`

  ```
  t : le n m   [|n|_O=1;|m|_O=1;|n|_S=|m|_S]
  ```

  $\mathtt{le}_2 = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S; |n|_S + 1 \geq |m|_S] \ldots$
  $\mathtt{le}_k = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S; |n|_S + k \geq |m|_S]$
- And with *acceleration*.
  $\mathtt{le}_\infty = [|n|_O = 1; |m|_O = 1; |n|_S \leq |m|_S]$

Prototype implantation

- An implantation parametric in the approximation used for inductive sets
- Two instances:
  - Trees with limited size
  - Counting the number of occurences of a constructors with a library of convex set `Polka`

Prototype implantation

- An implantation parametric in the approximation used for inductive sets
- Two instances:
  - Trees with limited size
  - Counting the number of occurences of a constructors with a library of convex set `Polka`

Prototype implantation

- An implantation parametric in the approximation used for inductive sets
- Two instances:
  - Trees with limited size
  - Counting the number of occurences of a constructors with a library of convex set `Polka`

Prototype implantation

- An implantation parametric in the approximation used for inductive sets
- Two instances:
  - Trees with limited size
  - Counting the number of occurences of a constructors with a library of convex set `Polka`

Refutations reconstruction

- Importance of reconstructing proof : safety of case elimination
- Two methods :
  - Prove every approximations is correct :
    Proof of the correction of the operation on approximated sets
  - Prove each approximation is correct :
    Use of automatic tactics in `Coq`, like `omega`

Refutations reconstruction

- Importance of reconstructing proof : safety of case elimination
- Two methods :
  - Prove every approximations is correct :
    Proof of the correction of the operation on approximated sets
  - Prove each approximation is correct :
    Use of automatic tactics in `Coq`, like `omega`

Refutations reconstruction

- Importance of reconstructing proof : safety of case elimination
- Two methods :
  - Prove every approximations is correct :
    Proof of the correction of the operation on approximated sets
  - Prove each approximation is correct :
    Use of automatic tactics in `Coq`, like `omega`

Outline

Conclusions

- This method allows to eliminate case with some simple inductive analysis.
- Need to extend the method with polymorphic and higher order types.
- Need of other data structures to approximate set of inductives.

Conclusions

- This method allows to eliminate case with some simple inductive analysis.
- Need to extend the method with polymorphic and higher order types.
- Need of other data structures to approximate set of inductives.

Conclusions

- This method allows to eliminate case with some simple inductive analysis.
- Need to extend the method with polymorphic and higher order types.
- Need of other data structures to approximate set of inductives.