Indução no assistente de provas Coq

Nesta seção, veremos como construir provas utilizando a indução matemática no assistente de provas Coq (`https://coq.inria.fr`). Um conjunto indutivamente definido pode ser construído com a palavra reservada `Inductive`. Por exemplo, o conjunto finito *Sem* dos dias da semana apresentado na seção anterior pode ser definido como a seguir:

```
Inductive Sem :=
| domingo: Sem
| segunda_feira: Sem
| terca_feira: Sem
| quarta_feira: Sem
| quinta_feira: Sem
| sexta_feira: Sem
| sabado: Sem.
```

O conjunto dos números naturais é uma construção nativa do Coq, *i.e.* ela está disponível uma vez que o sistema é iniciado, e pode ser vista a partir do comando `Print nat.`

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

Ou seja, o conjunto `nat` dos números naturais é definido indutivamente e possui 2 construtores: o `O` (zero), e `S` (sucessor). Claramente, esta definição corresponde à gramática (2.8). Toda definição indutiva possui um princípio indutivo associado, e que é automaticamente gerado pelo Coq. Por padrão, o nome do princípio indutivo associado a uma definição indutiva, digamos `mdef`, é `mdef_ind`. No caso de `nat` podemos acessar este princípio pelo comando `Print nat_ind`:

```
nat_ind =
fun (P : nat -> Prop) (f : P O) (f0 : forall n : nat, P n -> P (S n)) =>
fix F (n : nat) : P n := match n as n0 return (P n0) with
                         | O => f
                         | S n0 => f0 n0 (F n0)
                         end
:forall P: nat -> Prop, P O ->
                        (forall n: nat, P n -> P (S n)) ->
                        forall n: nat, P n
```

A parte que nos interessa desta saída está em azul. Como em *(PIM)*, a base de indução diz que `P O`, e o passo indutivo corresponde ao trecho `(forall n :  nat, P n -> P (S n))`. A conclusão como esperado, diz que `forall n :  nat, P n`.

A seguir faremos a prova de que a soma dos n primeiros números ímpares é igual a n^2. Inicialmente precisamos expressar a "soma dos n primeiros números ímpares" em Coq. Para isto, definiremos um somatório. Antes disto carregamos a biblioteca `Lia`, que vai nos ajudar com a simplificação de expressões aritméticas nos inteiros.

```
Require Import Lia.

Fixpoint msum (n:nat) :=
 match n with
 | 0 => 0
 |S k => (msum k) + (2*k+1)
 end.
```

A palavra reservada `Fixpoint` é utilizada para definir funções recursivas. Note que $\sum_{i=1}^{n}(2.i-1)$ corresponde a `msum n`. Podemos fazer alguns testes com esta definição:

```
Eval compute in (msum 1).
```

```
     = 1
   : nat
```

O valor retornado é 1 porque que é igual ao primeiro número ímpar.

```
Eval compute in (msum 2).
```

```
     = 4
   : nat
```

Aqui a resposta é igual a 4 porque corresponde à soma dos dois primeiros números ímpares, ou seja, 1+3.

```
Eval compute in (msum 3).
```

```
     = 9
   : nat
```

O valor retornado corresponde à soma dos 3 primeiros números ímpares: 1+3+5=9.

```
Eval compute in (msum 4).
```

```
     = 16
   : nat
```

Por fim, temos que a soma dos 4 primeiros números ímpares é 1+3+5+7=16.

De acordo com estes testes, nossa definição de somatório está funcionando corretamente, e portanto podemos escrever o lema que queremos provar, a saber, que a soma dos `n` primeiros números naturais é igual a `n*n`:

```
Lemma msum_square: forall n, msum n = n*n.
```

Faremos a mesma prova apresentada na árvore construída na seção anterior. Iniciamos a prova por indução em `n` com a tática (ou comando) `induction n`.

```
Lemma msum_square: forall n, msum n = n*n.
Proof.
 induction n.
```

```
2 goals (ID 11)

 ============================
 msum 0 = 0 * 0

goal 2 (ID 14) is:
msum (S n) = S n * S n
```

Temos 2 casos para analisar (`2 goals`): o primeiro corresponde a base da indução, e o segundo é o passo indutivo.

O primeiro caso é trivial, e a tática `reflexivity` é capaz de concluir que os lados esquerdo e direito da igualdade são iguais a 0, uma vez que `msum 0` é igual a 0.

No segundo caso, temos como hipótese de indução que a soma dos `k` primeiros números ímpares é igual a `k*k`, e precisamos provar que a soma dos `(S k)` primeiros números ímpares é igual a `(S k)*(S K)`:

```
1 goal (ID 14)

k : nat
IHk : msum k = k * k
============================
msum (S k) = S k * S k
```

Podemos, por exemplo, aplicar a tática `simpl` para simplificar a expressão `msum (S k)`, ou seja, para aplicarmos a definição de `msum`. Agora podemos substituir o lado esquerdo da hipótese de indução pelo lado direito via o comando `rewrite IHn`. A expressão resultante é uma igualdade envolvendo somas e multiplicações de números naturais:

```
1 goal (ID 24)

k : nat
IHk : msum k = k * k
============================
k * k + (k + (k + 0) + 1) = S (k + k * S k)
```

As simplificações algébricas necessárias para que possamos concluir que os lados esquerdo e direito da igualdade coincidem são feitas pela tática `lia`, e a prova completa, disponível no arquivo `pim.v`[4] , tem a seguinte forma:

[4] `http://flaviomoura.info/files/lca/pim.v`. Os exercícios propostos na seção anterior também estão disponíveis neste arquivo.

```
Lemma msum_square: forall n, msum n = n*n.
Proof.
 induction n.
 - reflexivity.
 - simpl. rewrite IHn. lia.
Qed.
```

Agora vamos estabelecer a equivalência entre PIM e o PIG:

Exercício 87. *Complete a prova a seguir:*

```
Lemma PIG: forall (P : nat -> Prop) (k : nat), P k ->
              (forall n, n >= k -> P n -> P (S n)) ->
                       forall n : nat, n >= k -> P n.
Proof.
 intros P k H1 IH n H2.
 assert (H := nat_ind (fun n => n >= k -> P n)).
 Admitted.
```

Observe que a prova do exercício anterior utiliza o PIM via o comando `nat_ind`, e portanto temos uma prova de PIG via PIM. No outro sentido, vamos enunciar PIM como um lema:

Exercício 88. *Complete a prova a seguir:*

```
Lemma PIM : forall P: nat -> Prop,
  (P 0) ->
  (forall n, P n -> P (S n)) ->
  forall n, P n.
Proof.
 intros P H IH n.
 apply PIG with 0.
 Admitted.
```

Indução Estrutural

Nesta seção veremos que o princípio de indução matemática (PIM) visto anteriormente é um caso particular de um princípio geral que está associado a qualquer conjunto definido indutivamente. Vimos dois tipos de regras utilizadas na construção de um conjunto definido indutivamente:

1. As regras não recursivas, ou seja, aquelas que definem diretamente um elemento do conjunto definido indutivamente;

2. As regras recursivas, ou seja, aquelas que constroem novos elementos a partir de elementos já construídos.

Como veremos no próximo exemplo, estas regras podem fazer uso de elementos de outros conjuntos previamente definidos. Formalmente, se $A_1, A_2, \ldots$ são conjuntos então a estrutura geral das regras de

um conjunto definido indutivamente B é como a seguir:

1. Inicialmente temos as regras não recursivas que definem diretamente os elementos $b_1, \ldots, b_m$ de B:

$$\frac{a_1 \in A_1 \qquad a_2 \in A_2 \ldots \; a_{j_1} \in A_{j_1}}{b_1[a_1, \ldots, a_{j_1}] \in B} \qquad \ldots \qquad \frac{a_1 \in A_1 \qquad a_2 \in A_2 \ldots \; a_{j_m} \in A_{j_m}}{b_m[x_1, \ldots, x_{j_m}] \in B}$$

2. Em seguida, temos as regras recursivas que constroem novos elementos a partir de elementos já construídos:

$$\frac{a_1 \in A_1 \ldots \; a_{j'_1} \in A_{j'_1} \; d_1, \ldots, d_{k_1} \in B}{c_1[x_1, \ldots, x_{j'_1}, d_1, \ldots, d_{k_1}] \in B} \qquad \ldots \qquad \frac{a_1 \in A_1 \ldots \; a_{j_n} \in A_{j_n} \; d_1, \ldots, d_{k_n} \in B}{c_n[a_1, \ldots, a_{j_n}, d_1, \ldots, d_{k_n}] \in B}$$

Qualquer elemento de um conjunto definido indutivamente pode ser construído após um número finito de aplicações das regras que o definem (e somente com estas regras). Os elementos $d_1, d_2, \ldots, d_{k_i}$ são ditos *estruturalmente menores* do que o elemento $c_i[a_1, \ldots, a_{j_i}, d_1, \ldots, d_{k_i}]$. Isto significa que os elementos $d_1, d_2, \ldots, d_{k_i}$ são subtermos próprios de $c_i[a_1, \ldots, a_{j_i}, d_1, \ldots, d_{k_i}]$.

Podemos associar um princípio de indução a qualquer conjunto definido indutivamente. No contexto genérico acima, teremos um caso base (base da indução) para cada regra não recursiva, e um passo indutivo para cada regra recursiva. O esquema simplificado (omitindo os parâmetros por falta de espaço) tem a seguinte forma:

$$\frac{\overbrace{P(b_1) \ldots P(b_m)}^{\text{casos base}} \quad \overbrace{(\forall d_1 \ldots d_{k_1}, P(d_1), \ldots, P(d_{k_1}) \Rightarrow P(c_1)) \ldots (\forall d_1 \ldots d_{k_1}, P(d_1), \ldots, P(d_{k_n}) \Rightarrow P(c_n))}^{\text{casos indutivos}}}{\forall x \in B, P\ x}$$

Retornando ao caso do conjunto dos números naturais, temos um princípio indutivo com apenas um caso base e um caso indutivo:

$$\frac{P\ 0 \qquad \forall k, P\ k \Longrightarrow P\ (S\ k)}{\forall n, P\ n}$$

O conjunto dos booleanos possui um princípio indutivo com dois casos base, e nenhum caso indutivo:

$$\frac{P \text{ true} \qquad P \text{ false}}{\forall b, P\ b}$$

A gramática (2.2) nos diz como as fórmulas da LP podem ser construídas. Observe, em particular, seus os construtores recursivos: por exemplo, a negação de uma fórmula é construída a partir de outra fórmula já construída; a conjunção, a disjunção e a implicação são construídas a partir de duas fórmulas previamente construídas. Podemos derivar o princípio de indução para o conjunto das fórmulas da LP de maneira análoga aos casos anteriores, ou seja, como um caso particular da generalização apresentada acima. Como temos dois construtores não recursivos (variáveis proposicionais e a constante $\bot$) e quatro construtores recursivos (negação, conjunção, disjunção e implicação), o princípio indutivo correspondente terá a seguinte forma, considerando uma propriedade Q qualquer das fórmulas da LP:

$$\frac{(Q\ p) \qquad (Q\ \bot) \qquad (\forall \varphi, Q\ \varphi \Longrightarrow Q\ (\neg\varphi)) \qquad (\forall \varphi_1, Q\ \varphi_1 \wedge \forall \varphi_2, Q\ \varphi_2 \Longrightarrow Q\ (\varphi_1 \star \varphi_2))}{\forall \varphi, Q\ \varphi}$$

onde $\star \in \{\wedge, \vee, \rightarrow\}$. Chamamos o princípio de indução construído a partir de uma gramática recursiva de *indução estrutural*. Note que, para cada um dos conectivos binários (conjunção, disjunção e implicação) temos duas hipóteses de indução.

No exemplo a seguir, vamos mostrar que a gramática acima possui redundâncias, isto é, que existem conectivos que podem ser escritos a partir de outros:

Exemplo 89. *Prove, sem utilizar tabela de verdade, que para qualquer fórmula φ, existe uma fórmula φ' equivalente a φ construída apenas com os conectivos $\vee$ e $\neg$, e com os símbolos proposicionais que ocorrem em φ.*

Dizemos que duas fórmulas φ e ψ da LP são equivalentes se $\varphi \leftrightarrow \psi$ é uma tautologia. Provaremos este exercício por indução estrutural, isto é, indução na estrutura de φ:

- *Se φ é uma variável proposicional ou a constante $\bot$ então tome $\varphi' = \varphi$.*
- *Se $\varphi = \neg\psi$ então, por hipótese de indução, existe uma fórmula ψ' equivalente a ψ construída apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em ψ. Neste caso, basta tomar $\varphi' = \neg\psi'$, e estamos prontos.*
- *Se $\varphi = \psi_1 \vee \psi_2$ então, por hipótese de indução, existem fórmulas $\psi_i'(i = 1, 2)$, equivalentes respectivamente a $\psi_i(i = 1, 2)$, e construídas apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em $\psi_i(i = 1, 2)$. Neste caso, basta tomar $\varphi' = \psi_1' \vee \psi_2'$ e estamos prontos.*
- *Se $\varphi = \psi_1 \wedge \psi_2$ então, por hipótese de indução, existem fórmulas $\psi_i'(i = 1, 2)$, equivalentes respectivamente a $\psi_i(i = 1, 2)$, e construídas apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em $\psi_i(i = 1, 2)$. Pelo exercício 53 sabemos que $\psi_1 \wedge \psi_2 \dashv\vdash \neg(\neg\psi_1 \vee \neg\psi_2)$. Então basta tomar $\varphi' = \neg(\neg\psi_1' \vee \neg\psi_2')$, e estamos prontos.*
- *Por fim, se $\varphi = \psi_1 \rightarrow \psi_2$ então, por hipótese de indução, existem fórmulas $\psi_i'(i = 1, 2)$, equivalentes respectivamente a $\psi_i(i = 1, 2)$, e construídas apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em $\psi_i(i = 1, 2)$. Pelo exercício 54 da lista sabemos que $\psi_1 \rightarrow \psi_2 \dashv\vdash (\neg\psi_1) \vee \psi_2$. Então basta tomar $\varphi' = (\neg\psi_1') \vee \psi_2'$ e estamos prontos.*

Agora é a sua vez! Resolva o exercícios a seguir:

Exercício 90. *Prove, sem utilizar tabela de verdade, que para qualquer fórmula φ, existe uma fórmula φ' equivalente a φ construída apenas com os conectivos $\rightarrow$ e $\neg$, e com os símbolos proposicionais que ocorrem em φ.*

Exercício 91. *Prove, sem utilizar tabela de verdade, que para qualquer fórmula φ, existe uma fórmula φ' equivalente a φ construída apenas com os conectivos $\wedge$ e $\neg$, e com os símbolos proposicionais que ocorrem em φ.*

Considere a gramática da LPO:

$$\varphi ::= p(t, \ldots, t) \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \to \varphi) \mid \exists_x \varphi \mid \forall_x \varphi$$

o princípio de indução correspondente é dado como a seguir:

$$\frac{(\forall t_1, \ldots, t_n, Q\ p(t_1, \ldots, t_n)) \qquad (Q\ \bot) \qquad (\forall \varphi, Q\ \varphi \Longrightarrow Q\ (\neg\varphi)) \qquad (*) \qquad (**)}{\forall \varphi, Q\ \varphi}$$

onde

$(*)$ é igual a $(\forall \varphi_1, Q\ \varphi_1 \wedge \forall \varphi_2, Q\ \varphi_2 \Longrightarrow Q\ (\varphi_1 \star \varphi_2))$, $\star \in \{\wedge, \vee, \to\}$;

$(**)$ é igual a $(\forall x, \varphi, Q\ \varphi(x) \Longrightarrow Q\ (R_x \varphi(x)))$, $R \in \{\exists, \forall\}$.

Exercício 92. *Seja φ uma fórmula da lógica de predicados. Definimos a tradução negativa de Gödel-Gentzen de φ, denotada por φ^N, indutivamente por:*

$$\varphi^N = \begin{cases} \neg\neg\varphi & \text{se } \varphi \text{ é uma fórmula atômica, ou a constante } \bot \\ \neg(\psi^N) & \text{se } \varphi = \neg\psi \\ \varphi_1^N \wedge \varphi_2^N & \text{se } \varphi = \varphi_1 \wedge \varphi_2 \\ \neg(\neg(\varphi_1^N) \wedge \neg(\varphi_2^N)) & \text{se } \varphi = \varphi_1 \vee \varphi_2 \\ \varphi_1^N \to \varphi_2^N & \text{se } \varphi = \varphi_1 \to \varphi_2 \\ \forall_x(\psi^N) & \text{se } \varphi = \forall_x \psi \\ \neg(\forall_x \neg(\psi^N)) & \text{se } \varphi = \exists_x \psi \end{cases}$$

Construa uma prova intuicionista para o sequente a seguir: $\neg\neg(\varphi^N) \vdash_i \varphi^N$

Exercício 93. *Uma fórmula da lógica de predicados ϕ pertence ao* fragmento negativo *se ϕ pode ser construída a partir da seguinte gramática, onde $t_1, t_2, \ldots, t_n$ $(n > 0)$ são termos:*

$$\phi ::= \neg p(t_1, t_2, \ldots, t_n) \mid \bot \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \to \phi) \mid (\forall_x \phi)$$

Prove na lógica minimal que $\neg\neg\theta \vdash_m \theta$ para qualquer fórmula θ pertencente ao fragmento negativo. Use indução na estrutura de θ.

Nos próximos capítulos estudaremos diversos algoritmos que utilizam a estrutura de lista encadeada, definida pela seguinte gramática $l ::= nil \mid a :: l$, onde nil representa a lista vazia, e $a :: l$ representa a lista com primeiro elemento a e cauda l. Como esta gramática possui um construtor não recursivo, e um

construtor recursivo, teremos um princípio de indução com um caso base, e um passo indutivo:

$$\frac{P\ nil \qquad \forall l\ h, P\ l \Longrightarrow P\ (h :: l)}{\forall l, P\ l}$$

O comprimento de uma lista, isto é, o número de elementos que a lista possui, é definido recursivamente por:

$$|l| = \begin{cases} 0, & \text{se } l = nil \\ 1 + |l'|, & \text{se } l = a :: l' \end{cases}$$

Uma operação importante que nos permite construir uma nova lista a partir de duas listas já construídas é a concatenação. Podemos definir a concatenação de duas listas por meio da seguinte função recursiva:

$$l_1 \circ l_2 = \begin{cases} l_2, & \text{se } l_1 = nil \\ a :: (l' \circ l_2), & \text{se } l_1 = a :: l' \end{cases}$$

Por fim, o reverso de uma lista é definido recursivamente por:

$$rev(l) = \begin{cases} l, & \text{se } l = nil \\ (rev(l')) \circ (a :: nil), & \text{se } l = a :: l' \end{cases}$$

Os exercícios a seguir expressam diversas propriedades envolvendo estas operações. Resolva cada um deles utilizando indução.

Exercício 94. *Prove que $|l_1 \circ l_2| = |l_1| + |l_2|$, quaisquer que sejam as listas l_1, l_2.*

Exercício 95. *Prove que $l \circ nil = l$, qualquer que seja a lista l.*

Exercício 96. *Prove que a concatenação de listas é associativa, isto é, $(l_1 \circ l_2) \circ l_3) = l_1 \circ (l_2 \circ l_3)$ quaisquer que sejam as listas l_1, l_2 e l_3.*

Exercício 97. *Prove que $|rev(l)| = |l|$, qualquer que seja a lista l.*

Exercício 98. *Prove que $rev(l_1 \circ l_2) = (rev(l_2)) \circ (rev(l_1))$, quaisquer que sejam as listas l_1, l_2.*

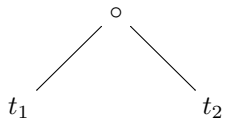
Exercício 99. *Prove que $rev(rev(l)) = l$, qualquer que seja a lista l.*

Outra estrutura de dados importante em Computação é a estrutura de árvores. O caso particular do conjunto *btree* das árvores binárias, isto é, as árvores cujos nós têm dois filhos, ou são folhas (não têm filhos) pode ser definido indutivamente pelas seguintes regras:

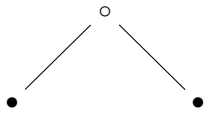
$$\frac{}{\bullet \in btree} \qquad \frac{t_1 \in btree \qquad t_2 \in btree}{\circ(t_1, t_2)}$$

A gramática correspondente é dada por $t ::= \bullet \mid \circ(t, t)$.

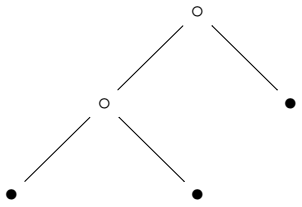
Assim, um nó sem filhos representa uma árvore (caso não recursivo), e t_1 e t_2 são duas árvores binárias então podemos construir uma nova árvore como na figura abaixo:



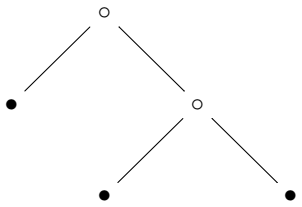
Observe que a árvore



é escrita como $\circ(\bullet, \bullet)$ na sintaxe da regra recursiva acima. Enquanto que



corresponde a $\circ(\circ(\bullet, \bullet), \bullet)$. Ou ainda,



corresponde a $\circ(\bullet, \circ(\bullet, \bullet))$.

O princípio de indução sobre *btree* terá um caso base, e um caso indutivo com duas hipóteses de indução. Nesta representação a raiz da árvore está no topo, e as folhas ficam para baixo (como ser a árvore estivesse de cabeça para baixo), mas veremos outras situações em que a raiz fica na parte inferior, e as folhas ficam no topo da árvore (como ocorre na natureza). As duas representações são utililizadas em Computação, como veremos.

Podemos definir a altura de uma árvore binária da seguinte forma:

$$h(t) = \begin{cases} 0, & \text{se } t = \bullet \\ 1 + \max(h(t_1), h(t_2)), & \text{se } t = \circ(t_1, t_2) \end{cases}$$

O número de nós de uma árvore binária é dado por:

$$n(t) = \begin{cases} 1, & \text{se } t = \bullet \\ 1 + n(t_1) + n(t_2)), & \text{se } t = \circ(t_1, t_2) \end{cases}$$

Exercício 100. *Mostre que $n(t) \leq 2^{h(t)+1} - 1$, para qualquer árvore binária t.*

Podemos flexibilizar um pouco a definição de árvore binária e permitir que um nó tenha, no máximo, dois filhos. Neste caso, acrescentaremos mais uma regra à nossa definição:

$$\frac{}{\bullet \in btree} \qquad \frac{t \in btree}{\circ(t)} \qquad \frac{t_1 \in btree \qquad t_2 \in btree}{\circ(t_1, t_2)}$$

A gramática correspondente é dada por $t ::= \bullet \mid \circ(t) \mid \circ(t, t)$. Resolva novamente o exercício anterior considerando esta nova gramática. Podemos estender esta definição para árvores cujos nóes possuem até $k \geq 0$ filhos, mas na prática ficaremos restritos a $k = 3$ por conta da estrutura das regras do sistema de dedução natural tanto na lógica proposicional quanto na lógica de predicados. Como exemplo, provaremos o Teorema de Glivenko:

Exemplo 101. *O teorema de Glivenko diz que se $\Gamma \vdash_c \varphi$ então $\Gamma \vdash_i \neg\neg\varphi$ na lógica proposicional, ou seja, se φ tem uma prova clássica a partir de Γ, então $\neg\neg\varphi$ tem uma prova intuicionista a partir de Γ na lógica proposicional. Faremos a prova deste teorema por indução na derivação $\Gamma \vdash_c \varphi$, isto é, indução na estrutura da árvore correspondente à derivação clássica de φ a partir de Γ. Queremos provar $\Gamma \vdash_i \neg\neg\varphi$, quais quer que sejam Γ e φ.*