Introdução ao Assistente de Provas Coq

Marcus Vinícius Midena Ramos

PUC - Encontro de Ciências Exatas e Tecnologia

21/10/2020

marcus.ramos@univasf.edu.br
(19 de outubro de 2020, 15:54)

Roteiro

Apresentação

Marcus Vinícius Midena Ramos

- Engenheiro Eletricista (EPUSP 1982);
- Mestre em Sistemas Digitais (EPUSP 1991);
- Doutor em Ciência da Computação (UFPE 2016);
- Professor do curso de Engenharia de Computação da UNIVASF (desde 04/2018); antes EPUSP, Sumaré, FASP, Senac, PUC e Maurício de Nassau;
- Coautor do livro Linguagens Formais (com I.S. Vega e J.J. Neto, Bookman 2009);
- Professor das disciplinas: Teoria da Computação, Linguagens Formais e Autômatos e Compiladores;
- Coordenador do grupo de estudos Provadores de Teoremas e suas Aplicações (de 07/2018 até 01/2020);
- Trabalha com a formalização matemática de linguagens livres de contexto usando o Coq (desde 2013).

Slides

Estes slides estão disponíveis em:
Prof. Marcus Ramos
`http://www.marcusramos.com.br/univasf/`
(Palestras e Mini-Cursos)

Introdução

Abordagem conceitual

Palestra realizada na PUC em 16/04/2019 (GEMS 348):
Provadores de Teoremas e suas Aplicações

- Motivação;
- Assistentes de prova;
- Aplicações;
- Coq;
- Objetivos;
- Exemplo completo;
- Teoria.

Abordagem prática

Palestra de hoje, 21/10/2020:
Introdução ao Assistente de Provas Coq

- Hands-on;
- Coq na prática;
- Construção de enunciados;
- Provas simples;
- Táticas básicas;
- Programação funcional.

Exercícios

Observações gerais

- Booleanos (`bool`), números naturais (`nat`) e listas de naturais (`nat_list`);
- Definições, exemplos e exercícios;
- Funções (linguagem funcional) e provas;
- Funções simples (não-recursivas e recursivas);
- Provas simples (diretas e por indução);
- Definições do próprio Coq;
- Utilizaremos o ProofWeb;
- Não será necessário instalar o Coq localmente;
- Assistência para dúvidas;
- Conferência dos resultados.

ProofWeb

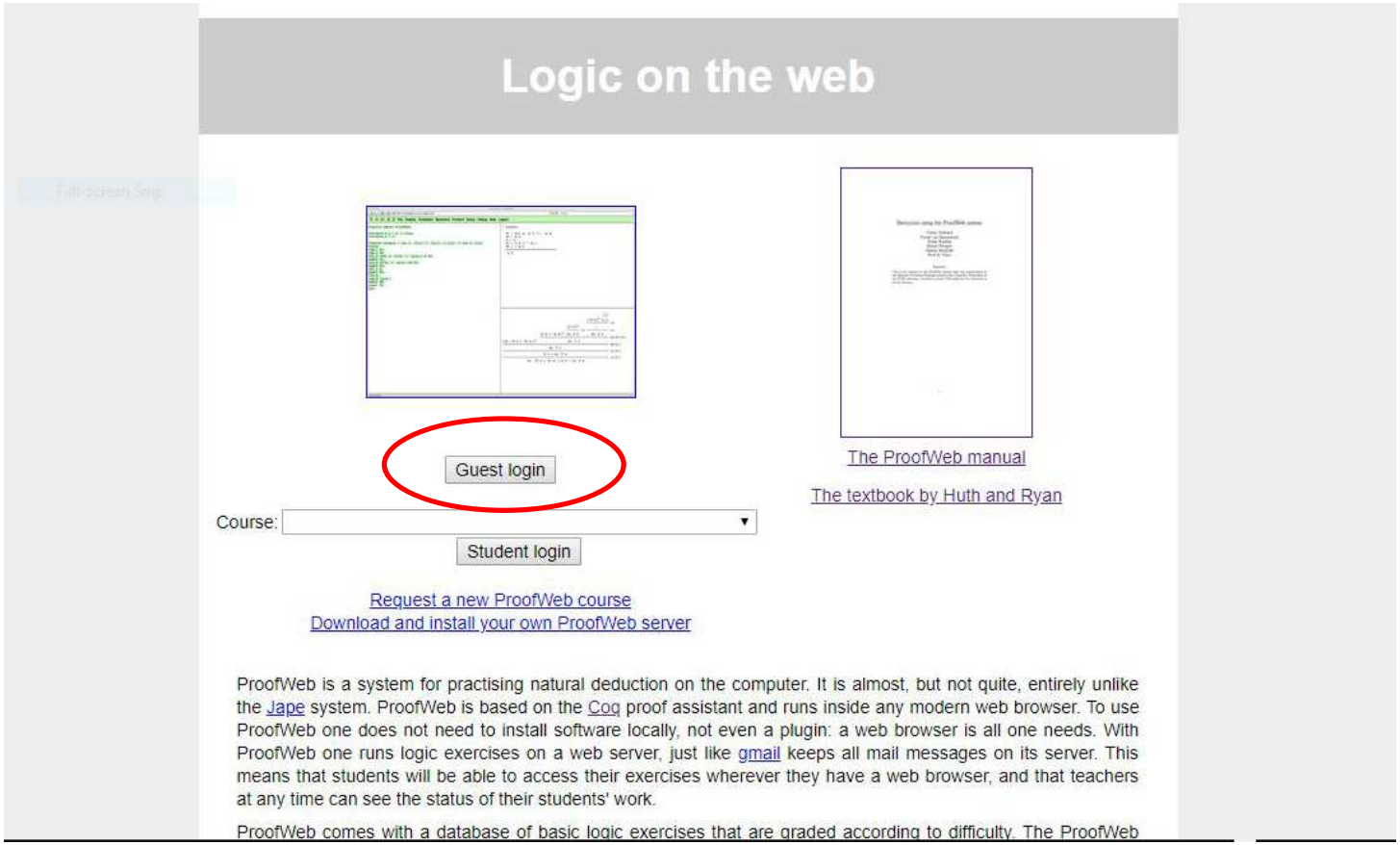
Versão web do Coq:

- Pode ser usada via navegador;
- Não precisa baixar nem instalar;
- Disponível em `http://proofweb.cs.ru.nl`;
- Clicar em "Guest login";
- Clicar em "Access the interface";
- Alternativamente, é possível se identificar e salvar os arquivos;
- Oferece também cursos na área;
- Suporta diversos assistentes de prova.

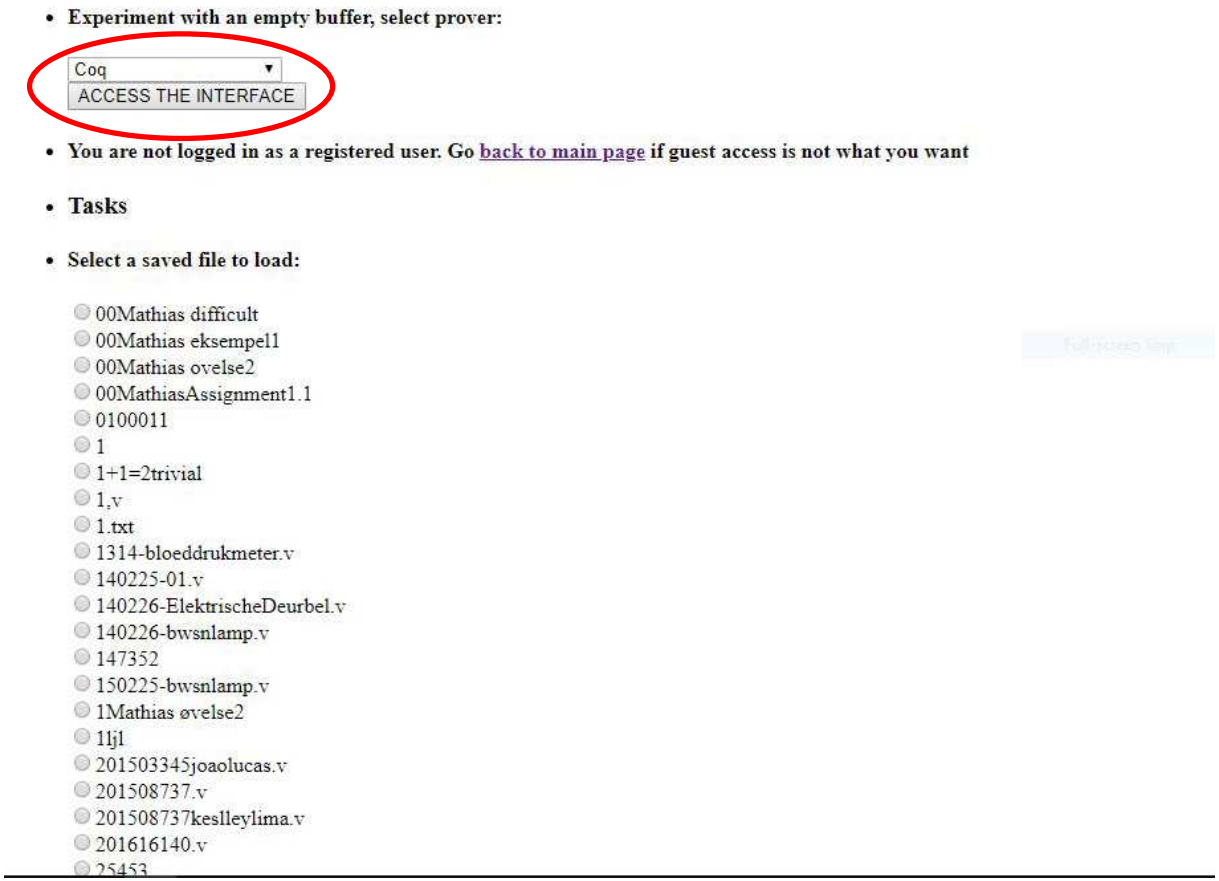
ProofWeb 1(4)


Courses
Provers
MathWiki
Calculator
ProofWeb
SURF FOUNDATION
What is ProofWeb?
ProofWeb is both a system for teaching logic and for using proof assistants through the web.
ProofWeb can be used in three ways. First, one can use the guest login, for which one does not even need to register. Secondly, a user can be a student in a logic or proof assistants course. We are hosting courses free of charge. If you are a teacher and would like to host your course on this server, send email to proofweb@cs.ru.nl. Thirdly, if teachers do not want to trust us with their students' files, they can freely download the ProofWeb system and run it on a server of their own.
ProofWeb works well with many web browsers, but it does not work with all versions of Internet Explorer. ProofWeb was developed using the Firefox browser, which can be downloaded for free.

ProofWeb 2(4)


Logic on the web
Guest login
Course:
Student login
Request a new ProofWeb course
Download and install your own ProofWeb server
The ProofWeb manual
The textbook by Huth and Ryan
ProofWeb is a system for practising natural deduction on the computer. It is almost, but not quite, entirely unlike the Jape system. ProofWeb is based on the Coq proof assistant and runs inside any modern web browser. To use ProofWeb one does not need to install software locally, not even a plugin: a web browser is all one needs. With ProofWeb one runs logic exercises on a web server, just like gmail keeps all mail messages on its server. This means that students will be able to access their exercises wherever they have a web browser, and that teachers at any time can see the status of their students' work.
ProofWeb comes with a database of basic logic exercises that are graded according to difficulty. The ProofWeb

ProofWeb 3(4)



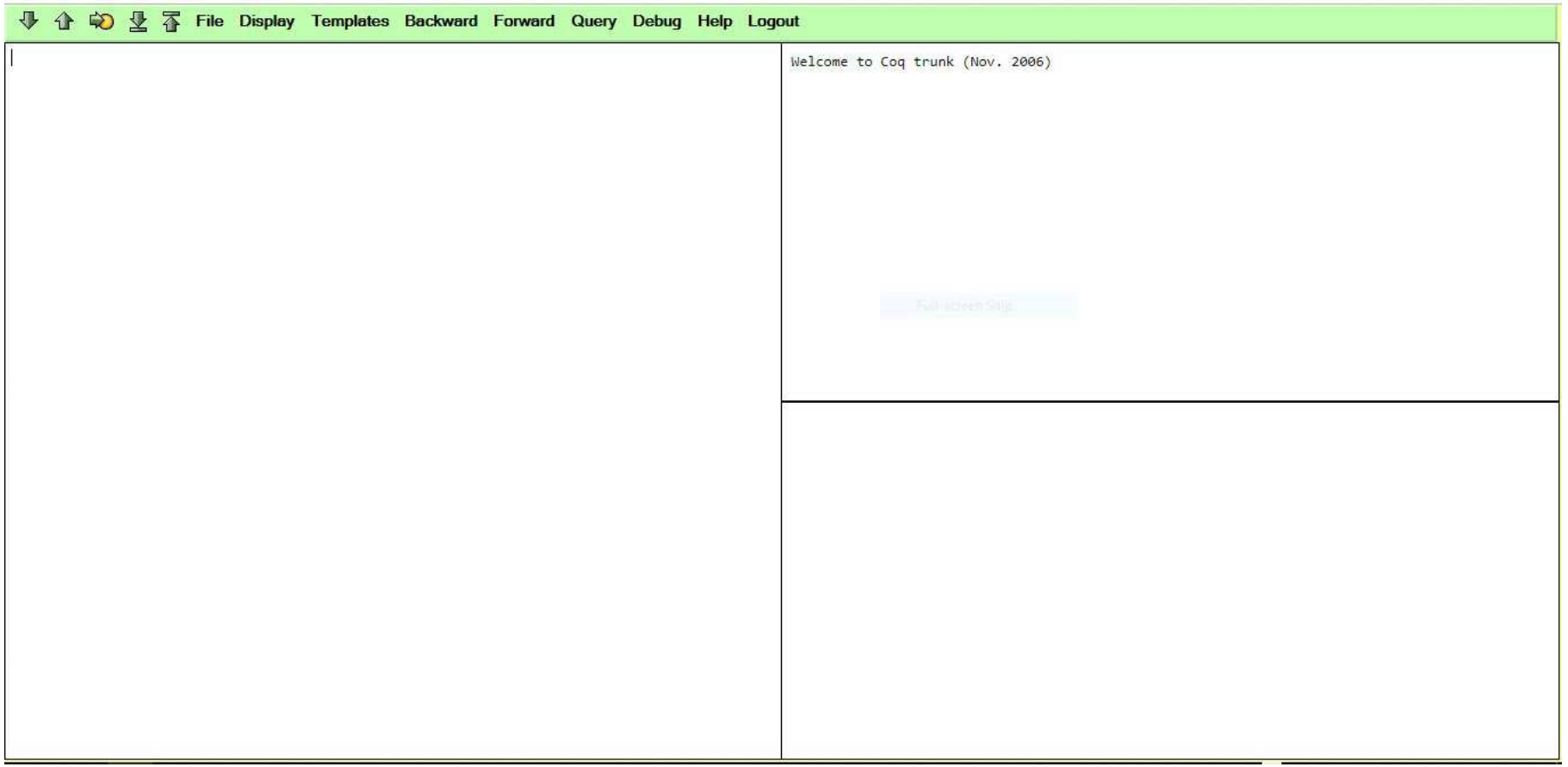
- **Experiment with an empty buffer, select prover:**

Coq ▾
ACCESS THE INTERFACE

- **You are not logged in as a registered user. Go back to main page if guest access is not what you want**
- **Tasks**
- **Select a saved file to load:**

00Mathias difficult
00Mathias eksempel1
00Mathias ovelse2
00MathiasAssignment1.1
0100011
1
1+1=2trivial
1,v
1.txt
1314-bloeddrukmeter.v
140225-01.v
140226-ElektrischeDeurbel.v
140226-bwsnlamp.v
147352
150225-bwsnlamp.v
1Mathias øvelse2
1lj1
201503345joaolucas.v
201508737.v
201508737keslleylima.v
201616140.v
25453

ProofWeb 4(4)


File
Display
Templates
Backward
Forward
Query
Debug
Help
Logout
Welcome to Coq trunk (Nov. 2006)

Observações ao utilizar o ProofWeb 1(2)

- Bullets `-`, `+`, `*` não são aceitos;
- O comando "`Compute`" deve ser substituído por "`Eval red in`" ou "`Eval vm_compute in`":

```
Compute (next_weekday friday).
Eval red in (next_weekday friday).
```

Observações ao utilizar o ProofWeb 2(2)

- O argumento descrecente deve ser explicitado nas funções recursivas com mais de um argumento:

```
Fixpoint mult (x y: nat): nat:=
match x with
| O ⇒ O
| S z ⇒ plus y (mult z y)
end.
Fixpoint mult (x y: nat) {struct x}: nat:=
match x with
| O ⇒ O
| S z ⇒ plus y (mult z y)
end.
```

Exercícios

Definição do tipo `bool`

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

- Tipo finito;
- Possui apenas dois valores (`false` e `true`);
- Cada valor corresponde à um construtor.

Exercícios

Definição do tipo `bool`

```
Check false.
Check true.
```

- Para verificar o tipo de um valor.

Exemplo

Definição da função negb

Negação booleana:

```
Definition negb (b: bool): bool:=
match b with
| false ⇒ true
| true ⇒ false
end.
```

- Nome da função, parâmetros e tipo do resultado;
- O comando match é usado para fazer análise de valores;
- Em função da análise é possível determinar o resultado.

Exemplo

Definição da função negb

Negação booleana:

```
Eval vm_compute in (negb false).
Eval vm_compute in (negb true).
```

- Para verificar o resultado da execução da função.

Exemplo

Tabelas-verdade

Conjunção $(a \wedge b)$:

	false	true
false	false	false
true	false	true

Disjunção $(a \vee b)$:

	false	true
false	false	true
true	true	true

Implicação $(a \rightarrow b)$:

	false	true
false	true	true
true	false	true

Exercícios

Conjunção booleana

- Escrever uma função que implementa o operador lógico “e” sobre valores do tipo booleano.

```
Definition andb (x y:bool): bool := ...
```

- Testar com:

```
Check andb.
Eval vm_compute in (andb false true).
Eval vm_compute in (andb true true).
Eval vm_compute in (andb false false).
```

Exercícios

Conjunção booleana

Solução:

```
Definition andb (x y:bool): bool :=
match x with
| true ⇒ y
| false ⇒ false
end.
```

ou

```
Definition andb (x y:bool): bool :=
match x with
| true ⇒ match y with
          | true ⇒ true
          | false ⇒ false
        end
| false ⇒ false
end.
```

Exercícios

Disjunção booleana

- Escrever uma função que implementa o operador lógico "ou" sobre valores do tipo booleano.

```
Definition orb (x y:bool): bool := ...
```

- Testar com:

```
Eval vm_compute in (orb false).
Eval vm_compute in (orb true false).
Eval vm_compute in (orb false true).
```

Exercícios

Disjunção booleana

Solução:

```
Definition orb (x y:bool): bool :=
match x with
| true ⇒ true
| false ⇒ y
end.
```

Exercícios

Implicação booleana

- Escrever uma função que implementa o operador lógico "implica" sobre valores do tipo booleano.

```
Definition implyb (x y:bool): bool := ...
```

- Testar com:

```
Eval vm_compute in (implyb false false).
Eval vm_compute in (implyb false true).
Eval vm_compute in (implyb true false).
Eval vm_compute in (implyb true true).
```

Exercícios

Implicação booleana

Solução:

```
Definition implyb (x y:bool): bool :=
match x with
| false ⇒ true
| true ⇒ y
end.
```

Notações

- Coq permite o uso de notações para melhorar a legibilidade dos termos em substituição à chamada de funções na notação pré-fixada;
- Cada notação é associada com uma função;
- É possível escolher símbolo, associatividade e precedência;
- Exemplos:

```
Notation "x && y" := (andb x y) (at level 40, left associativity).
Notation "x || y" := (orb x y) (at level 50, left associativity).
Notation "~ x" := (negb x).
Notation "x ⇒ y" := (implyb x y) (at level 70, right associativity).
```

Notações

- Testar com:

```
Eval vm_compute in (~ false).
Eval vm_compute in (true && true).
Eval vm_compute in (true || (false && true)).
Eval vm_compute in (false => true).
Eval vm_compute in (true => false).
```

Propriedades e Provas

Para provar o seguinte lema:

```
Lemma test_orb: orb (orb false false) true = true.
```

basta escrever:

```
Lemma test_orb: orb (orb false false) true = true.
Proof.
simpl.
reflexivity.
Qed.
```

- `Proof.` é usado para iniciar um script de prova;
- `Qed.` é usado para terminar um script de prova;
- A prova é construída indiretamente por meio do uso de táticas entre o `Proof.` e o `Qed.`

Tática `simpl`

- Uso: `simpl.`
- Simplifica o "goal" corrente.

Tática `reflexivity`

- Uso: `reflexivity.`
- Prova o "goal" corrente se este for uma equação com o mesmo termo em ambos os lados;
- Eventualmente também simplifica o "goal".

Exercícios

Provar os seguintes lemas sobre propriedades das funções anteriores:

```
Lemma test_andb: andb (andb true false) true = false.
Lemma and_true: ∀ x, andb true x = x.
Lemma imply_equiv: ∀ a b, (implyb a b) = (orb (negb a) b).
```

Exercícios

Solução:

```
Lemma test_andb: andb (andb true false) true = false.
Proof.
simpl.
reflexivity.
Qed.
```

Tática intros

- Uso: `intros <name>.`
- Transfere variáveis (de quantificadores universais) e premissas (lado esquerdo de implicações) para o contexto, indicando os respectivos nomes.

Exercícios

Solução:

```
Lemma and_true: ∀ x, andb true x = x.
Proof.
intros x.
simpl.
reflexivity.
Qed.
```

Tática destruct

- Uso: `destruct <name>.`
- Efetua análise de casos na variável de tipo indutivo <name>. São gerados tantos novos "goals" quantos sejam os construtores do tipo indutivo correspondente.

Exercícios

Solução:

```
Lemma imply_equiv: ∀ a b, (implyb a b) = (orb (negb a) b).
Proof.
intros a b.
destruct a.
simpl.
reflexivity.
simpl.
reflexivity.
Qed.
```

Com as notações, é possível escrever também:

```
Lemma imply_equiv: ∀ a b, (a ⇒ b) = (∼ a || b).
```

Exercícios

Definição do tipo nat

```
Inductive nat : Type :=
| O : nat
| S : nat → nat.
```

- O primeiro construtor (O) é uma função sem argumentos que representa o natural zero;
- O segundo construtor (S) é uma função que aceita como argumento um natural e retorna outro natural (sucessor);
- Tipo infinito;

Exercícios

Definição do tipo nat

- Número natural é representado em unário;
- Possui infinitos valores (O, S O, S (S O), etc);
- O representa 0, S O representa 1, S (S O) representa 2 e assim por diante;
- Cada valor corresponde à aplicação combinada de um par de construtores; existem infinitas combinações.

```
Check O.
Check (S O).
Check (S (S O)).
Check (S (S (S (O)))).
```

- Para verificar o tipo de um valor.

Exemplo

Definição da função pred

Predecessor:

```
Definition pred (n : nat) : nat :=
match n with
| O ⇒ O
| S n' ⇒ n'
end.
```

- Nome da função, parâmetros e tipo do resultado;
- O comando `match` é usado para fazer análise de valores;
- Em função da análise é possível determinar o resultado.

Exemplo

Definição da função pred

Predecessor:

```
Eval vm_compute in (pred 0).
Eval vm_compute in (pred (S 0)).
Eval vm_compute in (pred (S (S (S 0)))).
```

- Para verificar o resultado da execução da função.

Exercícios

Somar 2

- Escrever uma função que soma 2 a um valor do tipo natural.

```
Definition plustwo (n : nat) : nat := ...
```

- Testar com:

```
Eval vm_compute in (plustwo 0).
Eval vm_compute in (plustwo (S (S 0))).
```

Exercícios

Somar 2

Solução:

```
Definition plustwo (n : nat) : nat :=
match n with
| O ⇒ S ( S O)
| S n' ⇒ S ( S ( S n') ) )
end.
```

Exercícios

Somar dois números naturais

- Escrever uma função que soma dois números naturais quaisquer.

```
Fixpoint plus (n m: nat) {struct n} : nat := ...
```

- Testar com:

```
Eval vm_compute in (plus 0 (S 0)).
Eval vm_compute in (plus (S 0) (S 0)).
```

- Notar que `Fixpoint` deve ser usado no lugar de `Definition` se a função for recursiva.

Exercícios

Somar dois números naturais

Solução:

```
Fixpoint plus (n m: nat) {struct n} : nat :=
match n with
| O ⇒ m
| S n' ⇒ S ( plus n' m )
end.
```

- A notação abaixo é muito utilizada:

```
Notation "a + b" := (plus a b) (at level 50, left associativity).
```

- Neste caso é possível escrever, por exemplo:

```
Eval vm_compute in (0 + (S 0)).
```

Propriedades e Provas

Para provar o seguinte lema:

```
Lemma plus_O_n : ∀ n, (plus O n) = n.
```

basta escrever:

```
Lemma plus_O_n : ∀ n, (plus O n) = n.
Proof.
intros n.
simpl.
reflexivity.
Qed.
```

Tática induction

- Uso: `induction <name>.`
- Inicia uma prova por indução na variável <name>;
- Para isso, são gerados tantos novos "goals" quantos sejam os construtores do tipo indutivo <name>;
- Um princípio de indução é usado.

Tática `rewrite`

- Uso: `rewrite <name>.`
- Substitui um termo do "goal" pelo lado esquerdo (ou direito) da equação <name>;
- A reescrita pode ser feita da esquerda para a direita ($\rightarrow$, default) ou da direita para a esquerda ($\leftarrow$);
- Reescreve um termo, fazendo para isso uma substituição de subtermos.

Propriedades e Provas

Para provar o seguinte lema:

```
Lemma plus_n_O : ∀ n : nat, n = plus n O.
```

basta escrever:

```
Lemma plus_n_O : ∀ n : nat, n = plus n O.
Proof.
intros n.
induction n as [| n' IHn'].
reflexivity.
simpl.
rewrite ← IHn'.
reflexivity.
Qed.
```

Propriedades e Provas

Informalmente:

$$\forall n, n + 0 = n$$

Prova por indução em n:

- Caso base $(n = 0)$: $0 + 0 = 0$
- Caso indutivo $(n = Sm)$:
  - Deseja-se provar $(m + 0 = m) \Rightarrow (Sm + 0 = Sm)$;
  - Hipótese de indução, $m + 0 = m$;
  - Deve-se então provar $Sm + 0 = Sm$:
    - Sabe-se que $Sm + 0 = S(m + 0)$;
    - Reescrevendo a hipótese de indução, temos $Sm + 0 = Sm$.

Propriedades e Provas

Provar:

$$\forall n, 0 + n = n$$

não requer indução, porém provar:

$$\forall n, n + 0 = n$$

REQUER indução.

Exercícios

Provar os seguintes lemas sobre propriedades das funções anteriores:

```
Lemma plus_Sn_m: ∀ n m : nat, S n + m = S (n + m).
Lemma plus_n_Sm: ∀ n m : nat, S (n + m) = n + S m.
Lemma plus_comm: ∀ x y: nat, plus x y = plus y x.
Lemma plus_assoc: ∀ a b c: nat, plus (plus a b) c = plus a (plus b c).
```

Exercícios

Solução:

```
Lemma plus_Sn_m: ∀ n m : nat, S n + m = S (n + m).
Proof.
intros n m.
simpl.
reflexivity.
Qed.
```

Exercícios

Solução:

```
Lemma plus_n_Sm: ∀ n m : nat, S (n + m) = n + S m.
Proof.
induction n.
intros m.
simpl.
reflexivity.
intros m.
simpl.
rewrite ← IHn.
reflexivity.
Qed.
```

Exercícios

Solução:

```
Lemma plus_comm: ∀ x y: nat, x+y=y+x.
Proof.
intros x.
induction x.
intros y.
simpl.
rewrite ← plus_n_O.
reflexivity.
intros y.
rewrite plus_Sn_m.
rewrite IHx.
rewrite plus_n_Sm.
reflexivity.
Qed.
```

Exercícios

Solução:

```
Lemma plus_assoc: ∀ a b c: nat, plus (plus a b) c = plus a (plus b c).
Proof.
intros a.
induction a.
simpl.
reflexivity.
intros b c.
rewrite plus_Sn_m.
rewrite plus_Sn_m.
rewrite IHa.
rewrite ← plus_Sn_m.
reflexivity.
Qed.
```

Exercícios

Definição do tipo nat_list

```
Inductive nat_list: Type :=
| nil: natlist
| cons: nat → nat_list → nat_list.
```

- O primeiro construtor (nil) é uma função sem argumentos que representa a lista vazia;
- O segundo construtor (cons) é uma função que aceita como argumento um natural, uma lista de naturais e retorna outra lista de naturais (adiciona o elemento na frente);

Exercícios

Definição do tipo nat_list

```
Inductive nat_list: Type :=
| nil: nat_list
| cons: nat → nat_list → nat_list.
```

- Tipo infinito;
- Possui infinitos valores (`nil`, `cons 0 nil`, `cons (S 0) (cons 0 nil)`, etc);
- `nil` representa [], `cons 0 nil` representa [0], `cons (S 0) (cons 0 nil)` representa [1;0] e assim por diante;
- Cada valor corresponde à aplicação combinada de um par de construtores; existem infinitas combinações.

Exercícios

Definição do tipo nat

```
Check nil.
Check (cons 0 nil).
Check (cons (S 0) (cons 0 nil)).
```

- Para verificar o tipo de um valor.

Notações

- Lista vazia;
- Lista com um único elemento;
- Lista com vários elementos;
- Concatenação de listas;
- Acréscimo de elemento na frente de lista.
- Exemplos:

```
Notation "[ ]" := nil.
Notation "[ n ]" := (cons n nil).
Notation "[ x ; y ; .. ; z ]" := (cons x (cons y .. (cons z nil) ..)).
Notation "l1 ++l2" := (cat l1 l2) (at level 50, left associativity).
Notation "x :: y" := cons x y.
```

Exercícios

Comprimento de uma lista de naturais

- Escrever uma função que retorna o número de elementos em uma lista de números naturais.

```
Fixpoint length (l : nat_list) {struct l}: nat := ...
```

- Testar com:

```
Eval vm_compute in (length nil).
Eval vm_compute in (length (cons (S (S 0)) nil)).
Eval vm_compute in (length (cons 0 (cons (S (S 0)) nil)) ).
Eval vm_compute in (length []).
Eval vm_compute in (length [(S (S 0))]).
Eval vm_compute in (length [ 0 ; 0; 0 ]).
```

Exercícios

Comprimento de uma lista de naturais

Solução:

```
Fixpoint length (l : nat_list) {struct l}: nat :=
match l with
| nil ⇒ 0
| cons n l' ⇒ S ( length l')
end.
```

Exercícios

Concatenação de duas listas de naturais

- Escrever uma função que retorna uma lista correspondente à concatenação de duas outras listas de números naturais.

```
Fixpoint cat (l1 l2 : nat_list) {struct l1}: nat_list := ...
```

- Testar com:

```
Eval vm_compute in (cat [] [S O ; S (S O)]).
Eval vm_compute in (cat [] []).
Eval vm_compute in (cat [O ; O] []).
```

Exercícios

Concatenação de duas listas de naturais

Solução:

```
Fixpoint cat (l1 l2 : nat_list) {struct l1}: nat_list :=
match l1 with
| nil ⇒ l2
| cons n l' ⇒ cons n ( cat l' l2)
end.
```

Exercício

Provar o seguinte lema sobre propriedades das funções anteriores:

```
Lemma length_cat: ∀ l1 l2 : nat_list,
length (cat l1 l2) = length l1 + length l2.
```

Exercício

Solução:

```
Lemma length_cat: ∀ l1 l2 : nat_list,
length (cat l1 l2) = length l1 + length l2.
Proof.
intros l1 l2.
induction l1.
simpl.
reflexivity.
induction n.
simpl.
rewrite → IHl1.
reflexivity.
simpl.
rewrite → IHl1.
reflexivity.
Qed.
```

Exercício

Definir uma função que retorna o reverso de uma lista de números naturais:

```
Definition rev (l: nat_list): nat_list:= ...
```

Exercício

Solução:

```
Fixpoint rev (l: nat_list): nat_list:=
match l with
| nil ⇒ nil
| cons x l' ⇒ cat (rev l') (cons x nil)
end.
```

Exercícios

Provar os seguintes lemas sobre propriedades das funções anteriores:

```
Lemma cat_nil:
∀ l: nat_list, cat l nil = l.
Lemma cat_assoc:
∀ l1 l2 l3: nat_list, cat (cat l1 l2) l3 = cat l1 (cat l2 l3).
Lemma cat_rev:
∀ l1 l2: nat_list, rev (cat l1 l2) = cat (rev l2) (rev l1).
Lemma rev_involutive:
∀ l: nat_list, rev (rev l) = l.
```

Exercícios

Solução:

```
Lemma cat_nil:
∀ l: nat_list, cat l nil = l.
Proof.
induction l.
simpl.
reflexivity.
simpl.
rewrite IHl.
reflexivity.
Qed.
```

Exercícios

Solução:

```
Lemma cat_assoc:
∀ l1 l2 l3: nat_list, cat (cat l1 l2) l3 = cat l1 (cat l2 l3).
Proof.
induction l1.
simpl.
reflexivity.
intros l2 l3.
simpl.
rewrite IHl1.
reflexivity.
Qed.
```

Exercícios

Solução:

```
Lemma cat_rev:
∀ l1 l2: nat_list, rev (cat l1 l2) = cat (rev l2) (rev l1).
Proof.
induction l1.
simpl.
intros l2.
rewrite cat_nil.
reflexivity.
intros l2.
simpl.
rewrite IHl1.
rewrite cat_assoc.
reflexivity.
Qed.
```

Exercícios

Solução:

```
Lemma rev_involutive:
∀ l: nat_list, rev (rev l) = l.
Proof.
induction l.
simpl.
reflexivity.
simpl.
rewrite rev_cat.
simpl.
rewrite IHl.
reflexivity.
Qed.
```

Conclusões

- Vimos apenas algumas táticas básicas (`simpl`, `reflexivity`, `intros`, `destruct`, `induction` e `rewrite`);
- Existem muitas outras táticas e muitas variações das mesmas;
- Vimos apenas os tipos `bool`, `nat` e `nat_list`;
- Existem muitos outros e variações (incluindo tipos polimórficos);
- Existe uma biblioteca padrão bastante extensa e pronta para ser usada;
- Não discutimos nada sobre a teoria (Cálculo Lambda, Teoria de Tipos, Lógica Construtiva, Curry-Howard etc);
- Conforme se aprofunda no Coq, torna-se necessário conhecer a teoria subjacente (Cálculo de Construções com Definições Indutivas);
- O aprendizado efetivo só vem com o estudo da teoria e a prática do uso da ferramenta na formalização matemática e/ou no desenvolvimento de software certificado.

Referências

Referências

Estão todas disponíveis na página do grupo de estudos:
"Provadores de Teoremas e suas Aplicações"
`http://marcusramos.com.br/univasf/provadores/`

- Artigos;
- Livros;
- Slides;
- Links;
- Exercícios com solução;
- e muito mais.

Recomendações especiais

- Introdução ao Coq:
  Software Foundations Vol. 1 - Logical Foundations
  (Pierce et al)
- Coq:
  Interactive Theorem Proving and Program Development
  (Bertot & Castéran)
- Teoria do Coq (Cálculo de Construções):
  Type Theory and Formal Proof
  (Nederpelt & Geuvers)