Introdução ao Assistente de Provas Coq

Marcus Vinícius Midena Ramos

I Encontro Regional de Matemática Aplicada e Computacional

12 e 13/11/2019

marcus.ramos@univasf.edu.br
(14 de novembro de 2019, 18:04)

Roteiro

1. Apresentação
2. Motivação
3. Assistentes de Prova
4. Aplicações
5. Coq
6. Objetivos
7. Exemplo Completo
8. Teoria
9. Referências
10. Conclusões
11. Exercícios

Apresentação

Marcus Vinícius Midena Ramos

- Engenheiro Eletricista (EPUSP 1982);
- Mestre em Sistemas Digitais (EPSUP 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 (desde 07/2018)
- Trabalha com a formalização matemática de linguagens livres de contexto usando o Coq (desde 2013).

Assistência

Durante o mini-curso, contaremos com a assistência de dois alunos do curso de Engenharia de Computação da UNIVASF:

- Ruan Bahia
- Rafael Klebson

Eles estarão disponíveis nos dois dias e poderão ajudar com dúvidas e também na resolução dos exercícios propostos.

Slides

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

Programação

- PARTE 1:
  Motivação, histórico, características e aplicações
  12 de Novembro de 2019, 3ª feira, das 8:00h às 11:00h
  Seções 01 a 10
- PARTE 2:
  Exercícios práticos
  13 de Novembro de 2019, 4ª feira, das 8:00h às 11:00h
  Seção 11
  (trazer notebook)

Motivação

Sobre o que vamos falar?

- Matemática formal;
- Prova interativa de teoremas;
- Desenvolvimento interativo de programas certificados;
- Assistentes de prova em geral;
- Coq em particular.

Objetivos

- Introduzir os Assistentes de Prova Interativos (também conhecidos como Provadores de Teoremas);
- Discutir o seu papel no desenvolvimento de programas e na prova de teoremas;
- Apresentar alguns projetos de formalização relevantes, tanto na indústria quanto na academia;
- Apresentar tópicos das principais teorias utilizadas;
- Apresentar o assistente de provas Coq;
- Mostrar um exemplo completo;
- Fazer um conjunto de exercícios simples.

Provadores de Teoremas × Assistentes de Prova Interativos

Termos diferentes para designar a mesma coisa:

- "Provadores de Teoremas" é um termo bastante usado mas que não corresponde à realidade das ferramentas:
  Os provadores não provam teoremas, pelo menos não sozinhos;
- Por isso, este é considerado um termo um pouco mais pretensioso (ou ambicioso);
- Neste sentido, o termo "Assistentes Interativos de Provas" é mais razoável:
  Os assistentes ajudam o usuário a construir provas e não tem como objetivo construí-las sozinhos;
- Ainda assim, os Assistentes Interativos de Provas oferecem alguns recursos de automação que servem para casos especiais;
- Nesta apresentação, assim como na maior parte da literatura especializada, os dois termos serão usados de forma indistinta.

História e prática corrente

- Provas de teoremas:
  - Informais;
  - Difíceis de construir;
  - Difíceis de verificar.
- Programas de computador:
  - Informais;
  - Difíceis de construir;
  - Difíceis de testar.
- Coincidência?

História e prática corrente

- **NA VERDADE NÃO**, já que a prova de teoremas e o desenvolvimento de software possuem essencialmente a mesma natureza;
- De acordo com a Correspondência de Curry-Howard, desenvolver um programa é a mesma coisa que provar um teorema, e vice-versa;
- Explorar essa similaridade pode ser benéfica para ambas as atividades:
  - Raciocínio (“reasoning”) pode ser introduzido na programação, e
  - Computação pode ser usada na prova de teoremas.
- Como tirar proveito de tudo isso então?

Perspectivas

- A formalização matemática (*"matemática codificada no computador"*) é a resposta;
- Raciocínio auxiliado por computador;
- Uso de assistentes interativos de provas (provadores de teoremas).

Questões iniciais

- O que são teorias?
- O que são teoremas?
- O que são provas?
- O que são provadores de teoremas?

Questões iniciais

Perguntar não ofende:

- Coisa de maluco?
- Tem aplicação prática?
- Por que eu deveria me interessar por isso?

Questões iniciais

Nova (!??) maneira de:

- Provar teoremas;
- Desenvolver software.

Questões iniciais

Provadores de Teoremas:

- Programa de computador;
- Existem vários disponíveis;
- Mudam a teoria subjacente (por exemplo, clássica ou construtiva), as linguagens e as interfaces;
- Geralmente são gratuitos;
- Disponíveis para várias plataformas (Windows, Linux, iOS);
- Fazem a verificação mecânica da correção de uma prova;
- Funcionam como assistente interativo para elaboração de provas;
- Eventualmente permitem a extração de programas;
- Usam diversas linguagens e teorias.

Questões iniciais

Vantagens para:

- Matemáticos;
- Cientistas da computação;
- Programadores;
- Engenheiros de software;
- Empresas de desenvolvimento de software e hardware;
- Usuários de programas, componentes e aplicativos.

Questões iniciais

Para os matemáticos:

- Formalização;
- Segurança;
- Publicação;
- Compartilhamento;
- Reutilização.

Questões iniciais

- Prova de teoremas e desenvolvimento de software, o que uma coisa tem a ver com a outra?
- Não são coisas completamente diferentes? Teoria e prática?
- Tem tudo a ver!

Questões iniciais

Prova?

- Argumentação incontestável sobre a validade de uma proposição.
- Argumentação?
- Incontestável?
- Proposição?

Dependendo da audiência e das linguagens utilizadas, uma prova pode ou não ser aceita como válida. O desafio é propor uma linguagem que seja simples o suficiente para convencer todas as audiências, incluindo e principalmente as máquinas.

Questões iniciais

Teorema?

- Proposição não-trivial acerca de alguma definição ou conjunto de definições.
- Não-trivial?
- Definição?

Exemplo: $\forall n, n^2 + n$ é par.

Questões iniciais

Teoria?

- Conjunto limitado de definições (uma ou mais);
- Conjunto, geralmente extenso, de teoremas (ou lemas) que dizem respeito à(s) definição(ões);
- Teoremas e lemas são propriedades não-triviais das definições e que, por causa disso, necessitam ser demonstradas (provadas);
- As provas precisam ser formuladas em algum tipo de cálculo;
- A simplicidade do cálculo pode permitir que as provas sejam verificadas automaticamente.

Exemplos: (i) Cálculo Lambda e (ii) Linguagens Livres de Contexto.

Questões iniciais

Provar teoremas e desenvolver sofware?

- Correspondência de Curry-Howard;
- Dedução Natural;
- Cálculo Lambda.

Questões iniciais

Se alguns requisitos forem observados, a prova de um teorema se torna o programa que atende à uma certa especificação.

- Provas $\Leftrightarrow$ Programas;
- Proposições (ou Tipos) $\Leftrightarrow$ Especificações.

Questões iniciais

Conseqüência prática:

- Provar um teorema é a mesma coisa que construir um programa!

Questões iniciais

Atividades usuais:

- Obter uma prova para um teorema, ou seja,
  Prova $\Rightarrow$ Teorema (Proposição);
- Construir um programa que atende uma especificação, ou seja,
  Programa $\Rightarrow$ Especificação (Tipo).

Questões iniciais

Correspondência de Curry-Howard:

- Provas são Programas;
- Programas são Provas;
- Proposições são Especificações;
- Especificações são Proposições.

Questões iniciais

Desta forma, podemos:

- Construir uma especificação para um programa na forma de uma proposição;
- Construir uma prova para esta proposição;
- Obter o programa a partir da prova.

Questões iniciais

Conseqüências:

- Programas certificados;
- Não há necessidade de testes;
- Corretos por construção;
- Maior confiabilidade.

Importância decorre do uso generalizado e crescente de sistemas computacionais, especialmente em aplicações que oferecem risco à vida ou ao patrimônio.

Questões iniciais

Requisitos:

- Conhecer Provadores de Teoremas;
- Conhecer a teoria subjacente;
- Experiência;
- Força de vontade.

Assistentes de Prova

Características

- Ferramentas de software que auxiliam o usuário na prova de teoremas e no desenvolvimento de programas;
- A primeira iniciativa neste sentido data de 1967 (Automath, De Bruijn);
- Diversos provadores de teoremas estão disponíveis atualmente (Coq, Agda, Mizar, HOL, Isabelle, Matita, Nuprl...);
- Vale a pena conferir "The Seventeen Provers of the World"
- Interatividade;
- Interface gráfica;
- Verificação de provas e programas;
- Construção de provas e programas.

The Seventeen Provers of the World

The Seventeen Provers of the World

Compiled by Freek Wiedijk
(and with a Foreword by Dana Scott)

<freek@cs.ru.nl>
Radboud University Nijmegen

Abstract. We compare the styles of several proof assistants for mathematics. We present Pythagoras' proof of the irrationality of $\sqrt{2}$ both informal and formalized in (1) *HOL*, (2) *Mizar*, (3) *PVS*, (4) *Coq*, (5) *Otter/Ivy*, (6) *Isabelle/Isar*, (7) *Alfa/Agda*, (8) *ACL2*, (9) *PhoX*, (10) *IMPS*, (11) *Metamath*, (12) *Theorema*, (13) *Lego*, (14) *Nuprl*, (15) *Ωmega*, (16) *B method*, (17) *Minlog*.

	proof assistant	*author of proof*	*page*
	informal	Henk Barendregt	17
1	HOL	John Harrison, Konrad Slind, Rob Arthan	18
2	Mizar	Andrzej Trybulec	27
3	PVS	Bart Jacobs, John Rushby	31
4	Coq	Laurent Théry, Pierre Letouzey, Georges Gonthier	35
5	Otter/Ivy	Michael Beeson, William McCune	44
6	Isabelle/Isar	Markus Wenzel, Larry Paulson	49
7	Alfa/Agda	Thierry Coquand	58
8	ACL2	Ruben Gamboa	63
9	PhoX	Christophe Raffalli, Paul Rozière	76
10	IMPS	William Farmer	82
11	Metamath	Norman Megill	98
12	Theorema	Wolfgang Windsteiger, Bruno Buchberger, Markus Rozenkranz	106
13	Lego	Conor McBride	118
14	Nuprl	Paul Jackson	127

Utilização

1. O usuário escreve uma sentença (proposição) ou expressão de tipo (especificação) na linguagem da lógica utilizada;
2. Ele constrói (direta ou indiretamente):
   - Uma prova do teorema;
   - Um programa (termo) que satisfaz a especificação.
3. Diretamente: a prova/termo é escrita na linguagem formal aceita pelo assistente;
4. Indiretamente: a prova/termo é construída com a assistência de uma linguagem de "táticas" interativa;
5. Em qualquer caso, o assistente verifica que a prova/termo estão de acordo com o teorema/especificação.

Verificar e/ou construir

- Assistentes de provas verificam que provas/termos tenham sido corretamente construídos;
- Isto é feito por meio de algoritmos de verificação de tipos simples;
- Construção automatizada de provas/termos pode existir em alguns casos, com alguma abrangência, mas este não é o objetivo principal;
- Daí o nome "assistente de provas";
- Prova automática de teoremas pode ser buscada, devido à "irrelevância da prova" (a particular prova não importa, o que importa é a existência de uma prova);
- Desenvolvimento automático de programas, por outro lado, não é realista (o particular programa construído para atender uma especificação importa, e muito).

Principais benefícios

- Provas e programas podem ser mecanicamente verificados, poupando tempo e esforço e aumentando a sua confiabilidade;
- A verificação é eficiente;
- Os resultados podem ser facilmente armazenados e recuperados para uso futuro em diferentes contextos;
- Táticas ajudam o usuário na construção de provas/programas;
- O usuário ganha um entendimento mais profundo sobre a natureza das suas provas/programas, possibilitando melhorias futuras.

Aplicações

- Formalização e verificação de teoremas e teorias completas;
- Verificação de programas de computador;
- Desenvolvimento de software correto;
- Revisão automática de provas longas e complexas submetidas para periódicos especializados;
- Verificação de componentes de hardware e de software.

Desvantagens

- Falhas de infraestrutura podem reduzir a confiança nos resultados (código do assistente de provas, processadores de linguagens, sistema operacional, hardware etc);
- Tamanho das provas formais;
- Reduzido número de pessoas utilizando assistentes de provas;
- Curva de aprendizado cresce muito lentamente;
- Semelhança com código computacional mantém os matemáticos afastados e desinteressados.

Aplicações

Questões iniciais

O mundo está mudando:

- Empresas de software estão usando Provadores de Teoremas;
- Elas estão contratando profissionais que sabem usá-los;
- Competitividade, produtividade e qualidade;
- Aplicações importantes;
- Mercado emergente.

Questões iniciais

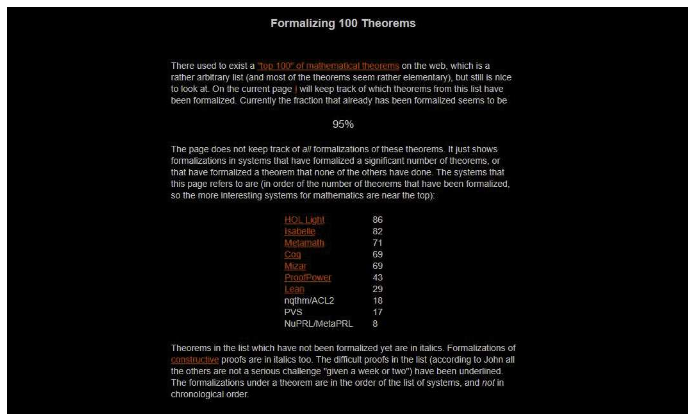
Já mudou alguma coisa?

- Intel;
- Microsoft;
- Compiladores, sistemas operacionais, chips, smart cards etc;
- Visível na Europa e nos EUA;
- Imperceptível no Brasil;
- Oportunidades de carreira e de empreendimento.

Introdução

- Interesse grande e crescente em provas formais e desenvolvimento certificado de programas nos últimos anos;
- Principais áreas:
  - Semântica de linguagens de programação;
  - Matemática em geral;
  - Educação.
- Projetos importantes tanto na academia quando na indústria;
- 100 teoremas mais importantes (95% formalizado até Novembro de 2019);
- Verifique "100 Theorems";
- Caminho de sentido único.

100 Theorems



Formalizing 100 Theorems

There used to exist a "top 100" of mathematical theorems on the web, which is a rather arbitrary list (and most of the theorems seem rather elementary), but still is nice to look at. On the current page I will keep track of which theorems from this list have been formalized. Currently the fraction that already has been formalized seems to be

95%

The page does not keep track of *all* formalizations of these theorems. It just shows formalizations in systems that have formalized a significant number of theorems, or that have formalized a theorem that none of the others have done. The systems that this page refers to are (in order of the number of theorems that have been formalized, so the more interesting systems for mathematics are near the top):

HOL Light	86
Isabelle	82
Metamath	71
Coq	69
Mizar	69
ProofPower	43
Lean	29
nqthm/ACL2	18
PVS	17
NuPRL/MetaPRL	8

Theorems in the list which have not been formalized yet are in italics. Formalizations of constructive proofs are in italics too. The difficult proofs in the list (according to John all the others are not a serious challenge "given a week or two") have been underlined. The formalizations under a theorem are in the order of the list of systems, and *not* in chronological order.

Alguns Projetos Verificados Formalmente

Destaques:

- Teorema das Quatro Cores;
- "Odd Order Theorem";
- Conjectura de Kepler;
- Teoria das Homotopias e Fundamentos Univalentes da Matemática;
- Certificação de Compilador;
- Certificação de Microkernel;
- Certificação de Segurança Digital.

Teorema das Quatro Cores

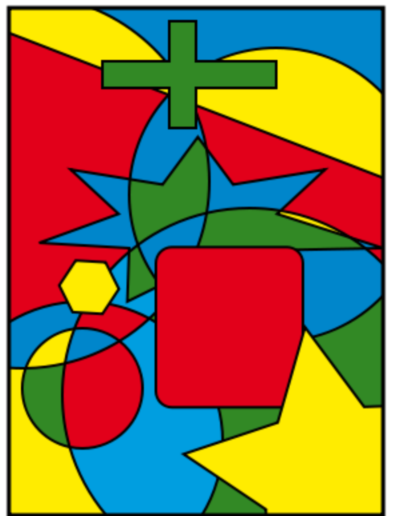
- Formulado em 1852, provado em 1976 e novamente em 1995;
- As duas provas fizeram uso de computadores em alguma extensão, mas não foram totalmente mecanizadas;
- Em 2005 Georges Gonthier (Microsoft Research) e Benjamin Werner (INRIA) produziram um script de prova que foi totalmente verificado por uma máquina;
- Marco na história da prova assistida por computador;
- 60.000 linhas de script Coq e 2.500 lemas;
- Produtos secundários.

Teorema das Quatro Cores

De acordo com a Wikipedia:

> *"In mathematics, the four color theorem, or the four color map theorem, states that, given any separation of a plane into contiguous regions, producing a figure called a map, no more than four colors are required to color the regions of the map so that no two adjacent regions have the same color. Adjacent means that two regions share a common boundary curve segment, not merely a corner where three or more regions meet."*

Teorema das Quatro Cores



Teorema das Quatro Cores

> *"Although this work is purportedly about using computer programming to help doing mathematics, we expect that most of its fallout will be in the reverse direction using mathematics to help programming computers."*

Georges Gonthier

"Odd Order Theorem"

- Também conhecido como Teorema de Feit-Thomson;
- "In mathematics, the Feit–Thompson theorem, or odd order theorem, states that every finite group of odd order is solvable" (Wikipedia);
- Importante na matemática (na classificação de grupos finitos) e na criptografia;
- Conjecturado em 1911, provado em 1963;
- Provado formalmente por uma equipe liderada por Georges Gonthier em 2012;
- Seis anos de dedicação exclusiva em tempo integral;
- Realização importantíssima na história da prova assistida por computador;
- 150.000 linhas de script Coq e 13.000 teoremas;

Último Teorema de Fermat

Oportunidade

Enunciado em Coq:

```
Theorem Fermat:
∀ a b c n: nat,
(a>0)∧ (b>0)∧ (c>0)→ (a^n+b^n=c^n)→ (n≤ 2).
```

- O Último Teorema de Fermat é um dos cinco teoremas que ainda não foi provado formalmente, da relação dos 100 mais importantes.

Último Teorema de Fermat

Oportunidade

De acordo com a Wikipedia:

- "In number theory Fermat's Last Theorem (sometimes called Fermat's conjecture, especially in older texts) states that no three positive integers a, b, and c satisfy the equation $a^n + b^n = c^n$ for any integer value of n greater than 2";
- "This theorem was first conjectured by Pierre de Fermat in 1637 in the margin of a copy of Arithmetica where he claimed he had a proof that was too large to fit in the margin. The first successful proof was released in 1994 by Andrew Wiles, and formally published in 1995, after 358 years of effort by mathematicians. The proof was described as a 'stunning advance' in the citation for his Abel Prize award in 2016".

Certificação de Compilador

- CompCert, um compilador vertificado para um amplo subconjunto da linguagem C e que gera código para a máquina PowerPC;
- O código objeto é certificado para ser compatível (semanticamente equivalente) com o código fonte em todos os casos;
- Aplicações em aviônica e outros sistemas críticos de software;
- CompCert não foi apenas verificado, mas totalmente desenvolvido em Coq;
- Três homens-ano ao longo de um período de cinco anos;
- 42.000 linhas de código Coq.

Certificação de Microkernel

- Componente crítico de sistemas operacionais, é executado em modo privilegiado;
- É mais difícil de ser testado em todas as situações;
- seL4, escrito em C (10.000 linhas), foi totalmente verificado em HOL/Isabelle;
- Sem interrupções, crashes ou execução de qualquer operação insegura em qualquer situação;
- A prova tem 200.000 linhas;
- 11 pessoas-ano podendo chegar à 8, 100% de acréscimo sobre um projeto não-certificado.

Certificação de Segurança Digital

- Plataforma de "smart card" JavaCard;
- Acumula dados pessoais tais como dados bancários, de crédito, saúde etc;
- É usado para várias aplicações de diversas empresas;
- Confiabilidade e integridade (dos dados pessoais) devem ser asseguradas;
- Formalização do comportamento e das propriedades da plataforma foi realizada;
- Certificação completa, nível mais alto foi alcançado;
- Parceria entre INRIA, Schlumberger e Gemalto.

Questões iniciais

O profissional do futuro precisa conhecer e saber usar a teoria. Provadores de Teoremas são apenas uma ferramenta.

Coq

Visão Geral

Coq é simultaneamente:

- Uma linguagem de programação funcional (que pode ser usada para construir programas e realizar computações);
- Um assistente interativo de provas que possibilita a extração de código;
- É justamente o uso combinado destes recursos que o torna uma ferramenta muito poderosa.

Visão Geral

Coq pode ser executado em linha de comando ou através de uma interface gráfica (CoqIDE ou Proof General).

- Coq implementa duas linguagens, Gallina e Vernacular;
- Gallina é a linguagem usada para representar termos (provas e programas) e proposições (teoremas e tipos);
- Gallina é baseada no Cálculo de Construções com Definições Indutivas;
- Vernacular é a linguagem de comando para interação com o usuário.

Visão Geral

- Desenvolvido por Huet/Coquand no INRIA em 1984;
- Primeira versão disponibilizada em 1989, tipos indutivos foram adicionados apenas em 1991;
- Desenvolvimento contínuo e utilização crescente desde então;
- A lógica utilizada é o Cálculo de Construções com Definições Indutivas;
- Este cálculo é implementado por meio de uma linguagem de programação funcional tipada com lógica de alta ordem chamada *Gallina*;

Visão Geral

- A interação com o usuário acontece via uma linguagem de comandos chamada *Vernacular*;
- Utiliza uma lógica construtiva e possui uma extensa biblioteca padrão com muitas contribuições de usuários;
- Ambiente extensível;
- Verifique "Coq Home Page" para downloads, documentação, comunidades, notícias e muito mais.

Sessão de usuário

Caso Direto

Uma prova pode ser construída direta ou indretamente.
No caso direto:

- O usuário deve construir manualmente o termo que representa a prova da proposição em questão;
- O termo deve ser escrito na linguagem *Gallina*;
- Simples quando a proposição é simples, pode ficar muito complicado quando a proposição for mais elaborada;
- Nestes casos, é mais fácil construir a prova de forma indireta.

Sessão de usuário

Caso Indireto

- O objetivo ("goal") inicial é o teorema ou a especificação suprida pelo usuário;
- O contexto inicial é geralmente vazio;
- A aplicação de uma tática, seja no "goal" corrente ou em uma das premissas do contexto, substitui, respectivamente, o "goal" corrente por zero ou mais novos "goals" ou modifica o contexto de forma correspondente;
- Se aplicada ao "goal" corrente, isto cria a noção de uma pilha de "goals", os quais precisam ser todos provados (em ordem reversa);

Sessão de usuário

Caso Indireto

- Se aplicada a alguma premissa do contexto, o mesmo muda e pode incorporar novas premissas;
- O processo é repetido para cada “goal”, até que não existe nenhum novo “goal” a ser provado;
- A prova/expressão é então construída, verificada e armazenada pelo assistente de provas, derivando diretamente da seqüênca de táticas que foi utilizada.

Uso das táticas

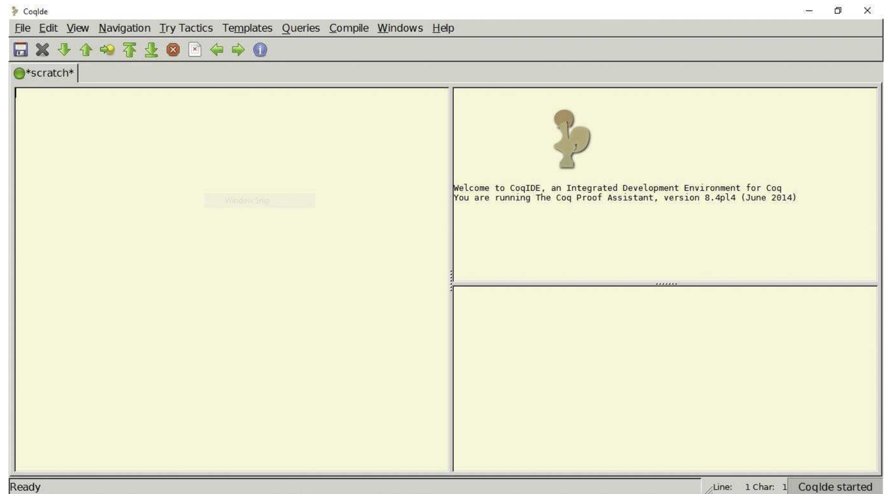
- Regras de inferência mapeiam premissas em conclusões;
- *Raciocínio direto, ou "Forward reasoning"* é o processo de ir das premissas em direção às conclusões;
  - Exemplo: de uma prova de a e uma prova de b é possível provar $a \wedge b$;
- *Raciocínio inverso, ou "Backward reasoning"* é o processo de ir das conclusões em direção às premissas;
  - Exemplo: para provar $a \wedge b$ deve-se provar a e também b;
- "Forward reasoning" é usado nas premissas do contexto;
- "Backward reasoning" é usado nos "goals";
- Cada regra de inferência está associada a uma ou mais táticas;
- Uma tática reduz o "goal" a novos "goals" (também conhecids como "subgoals"), se for o caso, modifica o contexto ou simplesmente prova o "goal" corrente.

CoqIDE

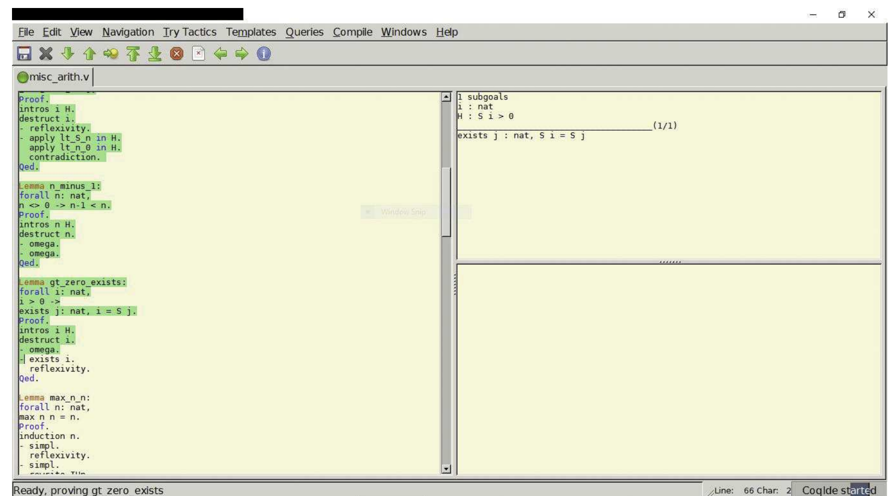
Uma interface gráfica para uso do Coq:

- Menus, atalhos e preferências;
- Lado esquerdo: editor de scripts (seqüência de definições e lemas/teoremas da teoria que se deseja formalizar);
- Lado direito em cima: contexto usado na prova ("goal" corrente e conjunto de premissas disponíveis);
- Lado direito embaixo: mensagens do sistema para o usuário.

CoqIDE



CoqIDE



CoqIDE

```
1 subgoals
n : nat
n1 : nat
n2 : nat
H1 : n > 1
H2 : n1 < n2
______________________________________(1/1)
n ^ n1 < n ^ n2
```

CoqIDE

Exemplo de sessão, prova da proposição:

```
Lemma example:
∀ a b c: Prop,
(a→ (b→ c))→ (b→ (a→ (b∧ c))).
```

- Seqüência de táticas;
- Pode-se avançar (ctrl-arrow-down) ou retroceder (ctrl-arrow-up) tática por tática;
- Táticas já processadas tornam-se verdes e ficam "travadas";
- Observe a mudança do "goal" e a geração de novos "subgoals";
- Observe a mudança do contexto.

CoqIDE

Prova da Implicação ($\rightarrow$)

- $a \rightarrow b$
- Regra de inferência para a implicação;
- Admite-se a como premissa verdadeira e tenta-se provar b, ou
- Prova-se que a não é verdadeiro;
- Táticas `intros` e `apply`:
  - `intros` para transferir a hipótese (a) para o contexto;
  - `apply` para unificar a conclusão de uma implicação do contexto (b em $a \rightarrow b$) com o "goal" corrente (b). Gera um novo "subgoal" relativo à premissa (a).

Regras de Inferência para a Implicação ($\rightarrow$)

Introdução / *Regra da Prova Condicional (RPC)*:

$$\frac{\begin{matrix}[a] \\ \dots \\ b\end{matrix}}{a \rightarrow b}\ (\rightarrow I)$$

Eliminação / *Modus Ponens (MP)*:

$$\frac{a \rightarrow b \qquad a}{b}\ (\rightarrow E)$$

CoqIDE

Prova da Conjunção ($\wedge$)

- $a \wedge b$
- Regra de inferência para a conjunção;
- Deve-se provar a, e também
- Deve-se provar b;
- Tática `split`:
  - Divide o "goal" ($a \wedge b$) corrente em dois novos "subgoals" (a e b);
  - Cria uma pilha de "subgoals" que devem ser provados individualmente (a e b).

Regras de Inferência para a Conjunção ($\wedge$)

Introdução / *Conjunção (C)*:

$$\frac{a \qquad b}{a \wedge b}\ (\wedge I)$$

Eliminação 1 / *Separação (S1)*:

$$\frac{a \wedge b}{a}\ (\wedge E_1)$$

Eliminação 2 / *Separação (S2)*:

$$\frac{a \wedge b}{b}\ (\wedge E_2)$$

Exemplo

Prova em Dedução Natural

Teorema:

$$(a \to (b \to c)) \to (b \to (a \to (b \land c)))$$

Prova:

$$\dfrac{\dfrac{\dfrac{[b] \qquad \dfrac{\dfrac{[a \to (b \to c)] \qquad [a]}{b \to c}\ (\to E) \qquad [b]}{c}\ (\to E)}{b \land c}\ (\land I)}{\dfrac{a \to (b \land c)}{b \to (a \to (b \land c))}\ (\to I)}\ (\to I)}{(a \to (b \to c)) \to (b \to (a \to (b \land c)))}\ (\to I)$$

CoqIDE

Script que constrói a prova

```
Lemma example:
∀ a b c: Prop,
(a→ (b→ c))→ (b→ (a→ (b∧ c))).
Proof.
intros a b c H1 H2 H3.
split.
– exact H2.
– apply H1.
  + exact H3.
  + exact H2.
Qed.
```

CoqIDE

Termo de prova

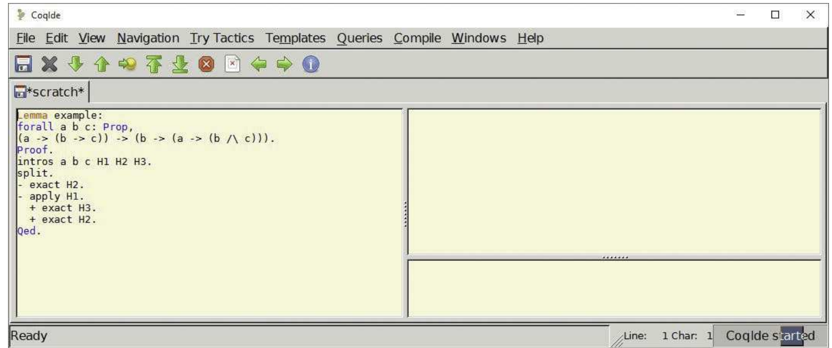
```
Print example.

example =
fun (a b c : Prop) (H1 : a → b → c) (H2 : b) (H3 : a)
⇒ conj H2 (H1 H3 H2)
: ∀ a b c : Prop, (a → b → c) → b → a → b ∧ c
```

CoqIDE

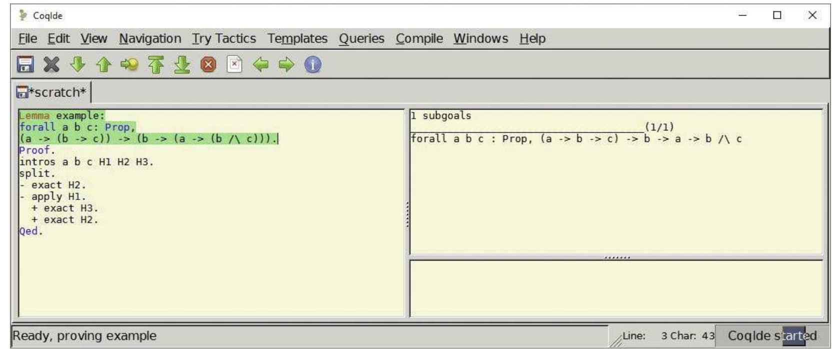
Termo de prova

- O termo anterior, do Cálculo de Construções com Definições Indutivas (baseado no Cálculo Lambda Tipado) representa a prova da proposição original;
- Trata-se de uma função que aceita como argumentos três variáveis (a, b e c) e três provas (H1, H2 e H3) sobre estas três variáveis;
- O corpo da função mostra como combinar estes argumentos para gerar a prova de $b \wedge c$;
- Em particular, ele faz a conjunção da premissa H2 (b) com o resultado da aplicação da premissa H1 ($a \rightarrow b \rightarrow c$) na premissa H3 ($a$) e depois novamente na premissa H2 (b).

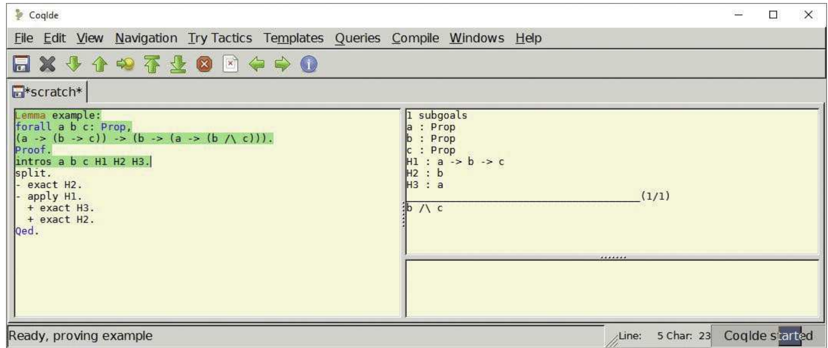
CoqIDE



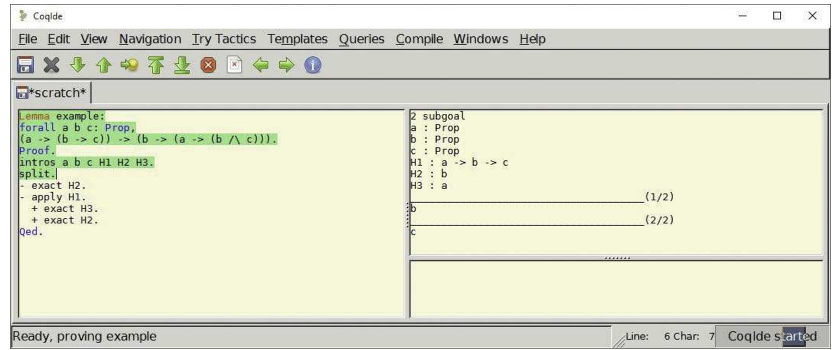
CoqIDE



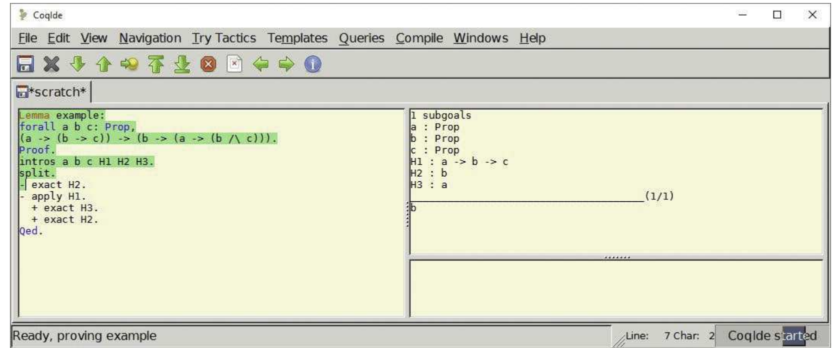
CoqIDE



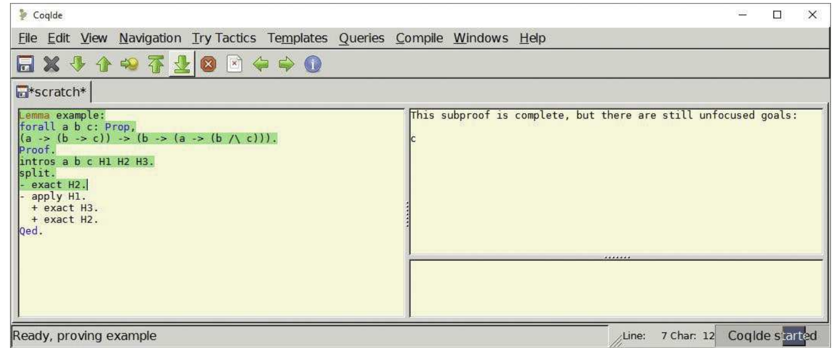
CoqIDE



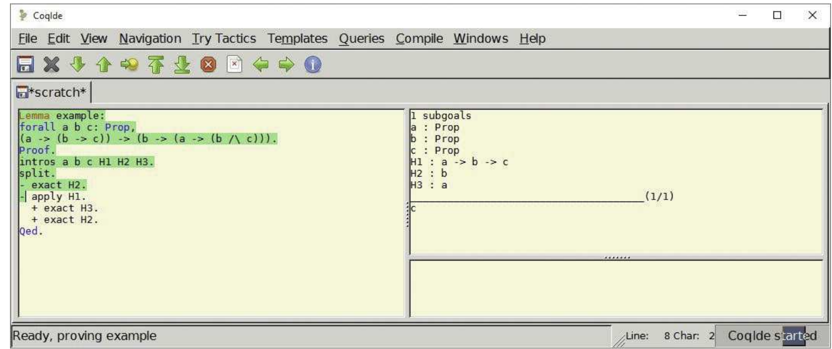
CoqIDE



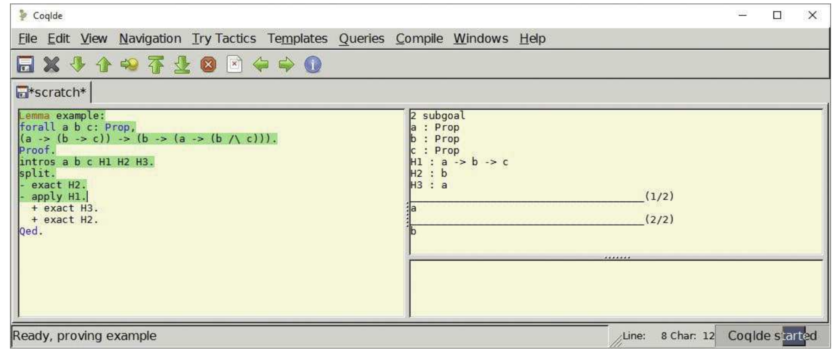
CoqIDE



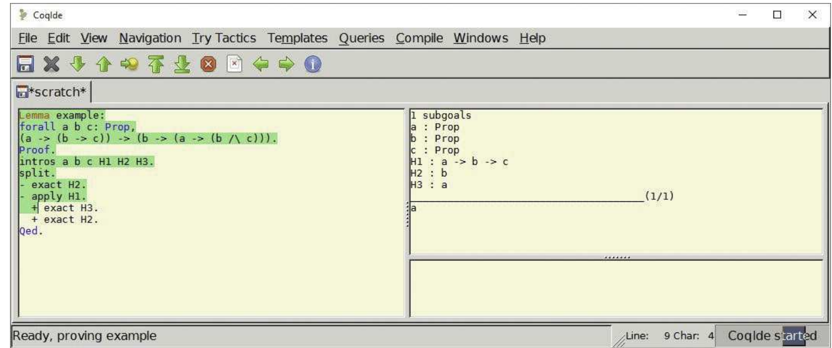
CoqIDE



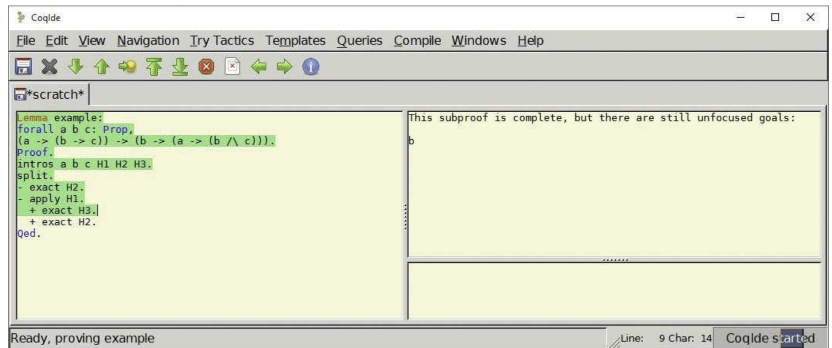
CoqIDE



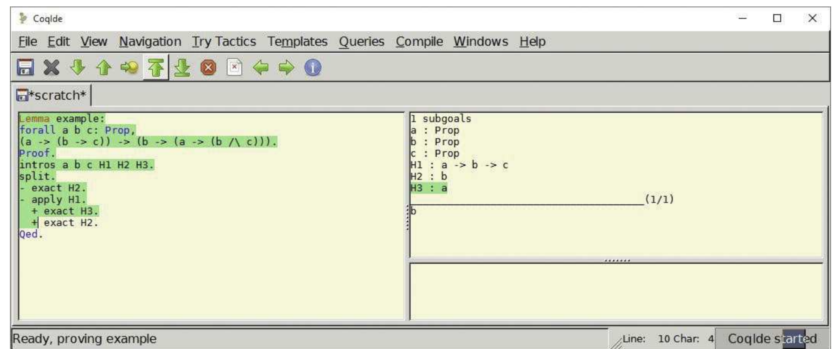
CoqIDE



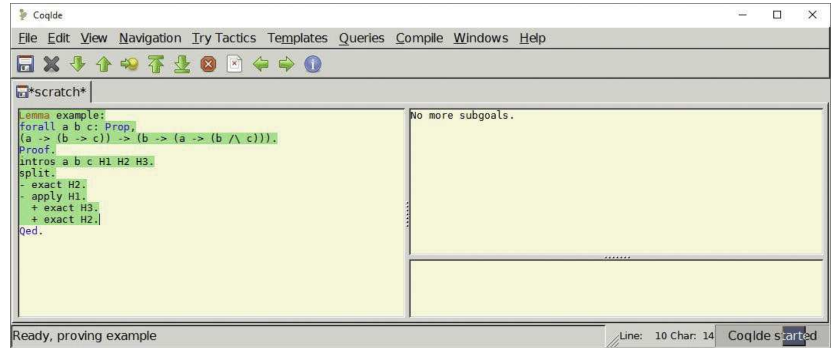
CoqIDE



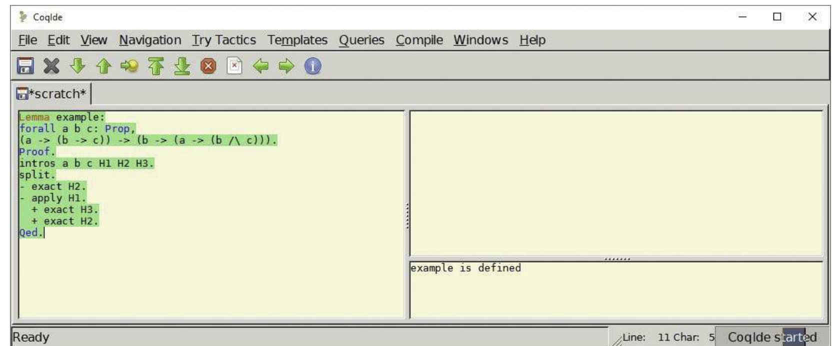
CoqIDE



CoqIDE



CoqIDE



Objetivos

Questões iniciais

Objetivos:

- Despertar o interesse pelo assunto;
- Apresentar uma técnica inovadora que está mudando a forma de se fazer matemática e de se desenvolver software;
- Estudar Provadores de Teoremas e Coq em particular;
- Entender o que é formalização matemática;
- Provar teoremas simples;
- Aprender como usar Coq para o desenvolvimento de software certificado;
- Incentivar o estudo continuado e a atuação na área, com pesquisas e publicações;
- Estimular a participação no nosso grupo de estudos.

Questões iniciais

Teorias envolvidas:

- Lógica;
- Teoria de Provas;
- Dedução Natural;
- Cálculo Lambda (não-tipado e tipado);
- Teoria de Tipos;
- Curry-Howard;
- Construtivismo;
- Técnicas de prova (indução etc)
- etc.

Questões iniciais

Objetos de estudo:

- Teorias (slide anterior);
- Coq;
- Exemplos;
- Exercícios;
- Estudos de caso;
- Slides, artigos e livros.

Muito estudo, muita persistência, muito tempo e muita dedicação.

Questões iniciais

Em resumo:

- Não é fácil;
- Aprendizado lento;
- Exige muita dedicação;
- Área ativa de pesquisa;
- Aplicações comerciais e acadêmicas de grande relevância;
- Muitas oportunidades na academia e na indústria;
- Tendência irreversível na matemática e na computação;
- É o futuro (da matémática e do desenvolvimento de software);
- Grande oportunidade.

Exemplo Completo

Especificação de um Programa

Aloritmo de ordenação

Como especificar um algoritmo de ordenação?

- Definir o domínio (listas de números inteiros);
- Relacionar entrada, saída e requisitos:
  - Entrada: uma lista de números inteiros (repetições são permitidas);
  - Saída: uma lista de números inteiros;
  - Requisito 1: as listas possuem os mesmos elementos (permutação);
  - Requisito 2: a lista de saída deve estar "ordenada".
- Escrever a proposição/especificação;
- Provar o teorema/construir o programa que implementa a especificação.

Objetivo

- Construir um programa certificado que ordena uma lista de números inteiros;
- Passos:
  - Formular a especificação do programa na forma de uma proposição da lógica de predicados;
  - Tratar a especificação como um teorema e construir a prova do mesmo;
  - Extrair o programa certificado a partir da prova.
- Extraído do livro:
  Interactive Theorem Proving and Program Development
  Yves Bertot e Pierre Castéran

Observações gerais

- Muitos detalhes;
- Não se preocupem em entender tudo;
- Busquem apenas uma intuição inicial do que está sendo feito e como está sendo feito;
- Mais importante é ter uma visão geral da dinâmica e do tipo de trabalho envolvido;
- A plena compreensão virá depois, com o tempo e a prática.

Script Coq

- Texto corrido;
- Processado de cima para baixo, esquerda para a direita;
- Mensagens de erros e interação com o usuário;
- Definições (indutivas e não-indutivas);
- Funções (recursivas e não-recursivas);
- Lemas e teoremas (proposições provadas de forma interativa usando um conjunto de táticas e regras de inferência; as provas são criadas indiretamente).

Script Coq

- Novos nomes são introduzidos em cada nova definição, lema, teorema ou outro;
- Utilização nas etapas seguintes;
- (lema A é usado para provar B, que por sua vez é usado para provar C e assim por diante)
- Computação e dedução;
- Lema ou teorema final;
- Provas completas;
    - Contexto;
    - Indução;
    - O script não é a prova!
- Extração de código.

Objetivo

Construir um programa certificado que ordena listas de números inteiros.

- Número inteiro?
- Lista?
- Lista ordenada?
- Qual seria a especificação deste programa?
- Uma vez especificado, como construímos a prova?
- Da prova, como extraímos o programa certificado?
- Série de definições (algumas indutivas outras não) e lemas.

Número Natural

- Um tipo de dados definido de maneira indutiva:
  - Existe pelo menos um caso base;
  - Existe pelo menos um caso indutivo.
- Dois construtores apenas;
- Construtores são funções usadas para construir os valores do tipo que está sendo definido;
- A expressão à direita do ":" representa o tipo da função;
- O construtor (função) `O` não tem argumentos e representa o valor "zero" (caso base);
- O construtor (função) `S` tem um único argumento e representa "sucessor" de um número natural, que também é um número natural (caso indutivo);
- Os números naturais são representados em unário.

Número Natural

Definição de número natural em Coq:

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

Exemplos:

O	(0)
S O	(1)
S (S O)	(2)
S (S (S O))	(3)
...	(...)

Lista

Definição de lista em Coq:

- Um tipo de dados definido de maneira indutiva:
  - Existe pelo menos um caso base;
  - Existe pelo menos um caso indutivo.
- Parametrizado em função do tipo do elemento (A);
- Dois construtores apenas:
  - `nil` representa "lista vazia" (caso base);
  - `cons` representa "acréscimo de elemento à esquerdasucessor" (caso indutivo).
- Tipo polimórfico (serve para qualquer tipo de elemento);
- Faz uso intensivo de "notações".

Lista

Definição de lista em Coq:

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

Exemplos:

nil	nil	[]
cons 3 nil	3 :: nil	[3]
cons (3 (cons (4 (cons 5 nil))))	3 :: 4 :: 5 :: nil	[3;4;5]

Lista Ordenada

Definição de lista ordenada em Coq:

- Uma proposição definida de maneira indutiva:
  - Existe pelo menos um caso base;
  - Existe pelo menos um caso indutivo.
- Uma coleção infinita de proposições definida de maneira indutiva;
- Usaremos números inteiros (`Z`) no lugar de números naturais (`nat`).
- Três construtores:
  - `sorted0`: lista vazia é ordenada por definição;
  - `sorted1`: lista com um único elemento é ordenada por definição;
  - `sorted2`: um número menor ou igual que o cabeça de uma lista ordenada, quando inserido no início da mesma, produz uma lista igualmente ordenada;
- Os construtores de uma proposição definida de maneira indutiva são considerados axiomas, proposições que são aceitas válidas sem provas.

Lista Ordenada

Definição de lista ordenada em Coq:

```
Inductive sorted : list Z → Prop :=
| sorted0 : sorted nil
| sorted1 : ∀ z:Z,  sorted (z::nil)
| sorted2 : ∀ z1 z2:Z,
            ∀ l:list Z,
            z1 ≤ z2 → sorted (z2::l) → sorted (z1::z2::l).
```

Exemplos:

Proposição	Construtores utilizados
sorted nil	sorted0
sorted (3::nil)	sorted1
sorted (2::3::nil)	sorted1 e sorted2
sorted (1::2::3::nil)	sorted1, sorted2 e sorted2

Prova de Ordenação

Prova de que a lista `2::3::5::7::nil` é ordenada:

```
Lemma sorted_example:
sorted (2::3::5::7:: nil).
Proof.
apply sorted2.
omega.
apply sorted2.
  + omega.
  + apply sorted2.
    * omega.
    * apply sorted1.
Qed.
```

Sublista Ordenada

Teorema auxiliar que prova que a remoção do elemento cabeça de uma lista ordenada preserva a ordenação da lista restante:

```
Theorem sorted_inv :
∀ z:Z,
∀ l:list Z,
sorted (z::l) → sorted l.
Proof.
intros z l H.
inversion H.
apply sorted0.
exact H3.
Qed.
```

Número de Ocorrências

Função recursiva que computa o número de ocorrências de um mesmo elemento numa lista:

```
Fixpoint nb_occ (z:Z) (l:list Z): nat:=
match l with
| nil => 0
| (z' :: l') =>
  match Z_eq_dec z z' with
  | left _ => S (nb_occ z l')
  | right _ => nb_occ z l'
  end
end.
```

Permutação

Proposição (predicado) que indica se uma lista é ou não é ordenada:

```
Definition permutation (l l':list Z) : Prop :=
∀ z:Z,
nb_occ z l = nb_occ z l'.
```

A palavra-chave "`Definition`" também é usada para introduzir funções não-recursivas. Se a função for recursiva deve-se usar "`Fixpoint`".

Inserção

Função recursiva que insere um número inteiro numa lista ordenada, de modo que a mesma continue ordenada:

```
Fixpoint insert (z:Z) (l:list Z): list Z :=
match l with
| nil => z :: nil
| cons a l' =>
  match Z_le_gt_dec z a with
  | left _ => z :: a :: l'
  | right _ => a :: (insert z l')
  end
end.
```

Em Resumo

Temos todos os elementos para formular a proposição que se deseja provar:

- Sabemos os que é um número natural (e inteiro);
- Sabemos o que é uma lista;
- Sabemos o que é uma lista ordenada;
- Sabemos o que é uma permutação;
- Sabemos inserir numa lista preservando a ordenação.

Portanto, podemos formular a especificação que desejamos provar.

Objetivo

Provar a proposição:

```
Lemma sort_correct:
∀ l: list Z,
∃ l': list Z,
permutation l l' ∧ sorted l'.
```

- A prova desta proposição garante a existência de uma lista ordenada equivalente (com os mesmos elementos) para qualquer outra que se considere;
- Um programa certificado pode ser extraído desta prova.

Script Coq da Prova

```
Proof.
induction l.
- ∃ nil.
 split.
 + apply permutation_refl.
 + apply sorted0.
- destruct IHl as [l' [H1 H2]].
 ∃ (insert a l').
 split.
 + apply permutation_trans with (l2:= a :: l').
  * apply permutation_cons.
   exact H1.
  * apply insert_permutation.
 + apply insert_sorted.
  exact H2.
Qed.
```

A Prova

```
sort_correct =
fun l : list Z =>
list_ind
  (fun l0 : list Z => exists l' : list Z, permutation l0 l' /\ sorted l')
  (ex_intro (fun l' : list Z => permutation nil l' /\ sorted l') nil
     (conj (permutation_refl nil) sorted0))
  (fun (a : Z) (l0 : list Z)
     (IHl : exists l' : list Z, permutation l0 l' /\ sorted l') =>
   match IHl with
   | ex_intro _ l' (conj H1 H2) =>
       ex_intro (fun l'0 : list Z => permutation (a :: l0) l'0 /\ sorted l'0)
         (insert a l')
         (conj
            (permutation_trans (a :: l0) (a :: l') (insert a l')
               (permutation_cons a l0 l' H1) (insert_permutation l' a))
            (insert_sorted l' a H2))
   end) l
     : forall l : list Z, exists l' : list Z, permutation l l' /\ sorted l'
```

O Programa Extraído 1(4)

```
type __ = Obj.t
let __ = let rec f _ = Obj.repr f in Obj.repr f
type 'a list =
| Nil
| Cons of 'a * 'a list
type comparison =
| Eq
| Lt
| Gt
(** val compOpp : comparison -> comparison **)
let compOpp = function
| Eq -> Eq
| Lt -> Gt
| Gt -> Lt
type sumbool =
| Left
| Right
type positive =
| XI of positive
| XO of positive
| XH
type z =
| Z0
| Zpos of positive
| Zneg of positive
```

O Programa Extraído 2(4)

```
module Pos =
 struct
  (** val compare_cont : comparison -> positive -> positive -> comparison **)
  let rec compare_cont r x y =
    match x with
    | XI p ->
      (match y with
       | XI q -> compare_cont r p q
       | XO q -> compare_cont Gt p q
       | XH -> Gt)
    | XO p ->
      (match y with
       | XI q -> compare_cont Lt p q
       | XO q -> compare_cont r p q
       | XH -> Gt)
    | XH ->
      (match y with
       | XH -> r
       | _ -> Lt)
  (** val compare : positive -> positive -> comparison **)
  let compare =
    compare_cont Eq
 end
```

O Programa Extraído 3(4)

```
module Z =
 struct
  (** val compare : z -> z -> comparison **)
  let compare x y =
    match x with
    | Z0 ->
      (match y with
       | Z0 -> Eq
       | Zpos _ -> Lt
       | Zneg _ -> Gt)
    | Zpos x' ->
      (match y with
       | Zpos y' -> Pos.compare x' y'
       | _ -> Gt)
    | Zneg x' ->
      (match y with
       | Zneg y' -> compOpp (Pos.compare x' y')
       | _ -> Lt)
 end
```

O Programa Extraído 4(4)

```
(** val z_le_dec : z -> z -> sumbool **)
let z_le_dec x y =
  match Z.compare x y with
  | Gt -> Right
  | _ -> Left
(** val z_le_gt_dec : z -> z -> sumbool **)
let z_le_gt_dec x y =
  z_le_dec x y
(** val insert : z -> z list -> z list **)
let rec insert z0 = function
| Nil -> Cons (z0, Nil)
| Cons (a, l') ->
  (match z_le_gt_dec z0 a with
   | Left -> Cons (z0, (Cons (a, l')))
   | Right -> Cons (a, (insert z0 l')))
(** val sort_correct : __ **)
let sort_correct =
  __
```

Teoria

Introdução

Formalização Matemática

- Construção de provas assistida por máquina;
- Verificação mecanizada de provas;
- Velocidade, confiabilidade e reutilização;
- Matemática e Ciência da Computação;
- Prova interativa de teoremas;
- Desenvolvimento certificado de hardware e software.

Casos reais

Formalização matemática é uma atividade madura:

- Usada ao longo dos anos;
- Diversidade de assistentes de provas e teorias subjacentes;
- Desenvolvimento da tecnologia dos assistentes de provas;
- Tamanho, complexidade e importância de diferentes projetos;
- Orientação teórica e tecnológica;
- Indústria e academia;
- Uma tendência clara;
- Ponto sem volta.

Quadro Geral

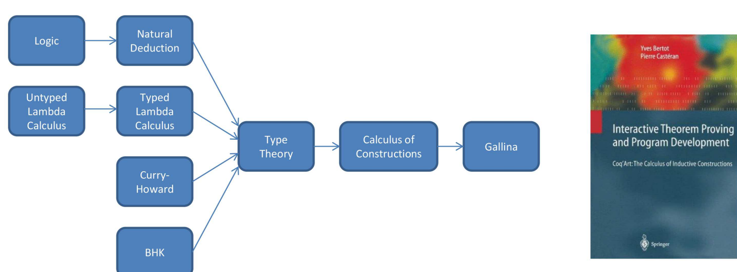
- Matemática "informal":
  - Diferentes níveis de abstração podem esconder erros difíceis de serem identificados;
  - Notação não-uniforme também pode constituir um problema.
- Formalização matemática (*"matemática codificada no computador"*) é uma tendência clara na direção do desenvolvimento teórico e da representação da teoria;
- Raciocínio auxiliado por computador e o uso de assistentes interativos de prova;
- Verificação mecânica de provas e programas, permitindo:
  - Verificação de cada passo de inferência contra um conjunto de regras de inferência da lógica subjacente;
  - Notação uniforme.
- Vantagens:
  - Menos esforço e tempo;
  - Maior confiabilidade.

Requisitos

Requisitos teóricos para usar e entender o Coq:

- Lógica;
- Dedução Natural;
- Cálculo Lambda Não-Tipado;
- Cálculo Lambda Tipado;
- Correspondência de Curry-Howard;
- Teoria de Tipos;
- Construtivismo e BHK;
- Teoria de Tipos Intuicionística de Martin Löf;
- Cálculo de Construções com Definições Indutivas.

Background



Referências

Referências

Estão todas disponíveis na página do nosso 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)

Conclusões

Fato

Provadores de Teoremas são o futuro (e o presente):

- Da matemática;
- Do desenvolvimento de software.

Tanto na industria quanto na academia.

Matemática

Principais características do processo:

- Verificação mecânica de provas;
- Assistência na construção de provas;
- Reaproveitamento de scripts;
- Repositórios;

Principais benefícios derivados:

- Produtividade;
- Correção;
- Uniformidade;
- Agilidade na publicação de originais.

Desenvolvimento de Software Certificado

Roteiro básico:

1. Escreva as especificações como expressões de tipo;
2. Use uma lógica poderosa o suficiente e certifique-se de que a especificação esteja correta;
3. Interprete a especificação como um teorema;
4. Construa a prova do teorema usando uma lógica construtiva;
5. Use o provador de teoremas para verificar a prova;
6. Converta a prova para um programa de computador usando o recurso de extraçao de código.

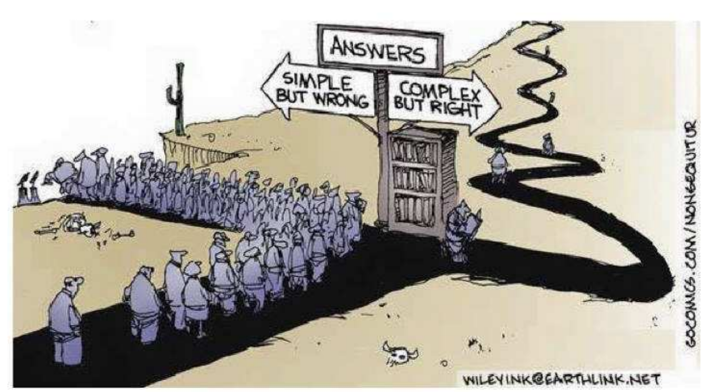
Uso de métodos matemáticos no lugar de métodos empíricos e subjetivos.

Certificação de Software já Desenvolvido

Roteiro básico:

1. Construir um termo que represente o programa;
2. Obter a expressão de tipo deste termo;
3. Verificar se a mesma corresponde à especificação desejada.

Computadores e Matemática



Computadores e Matemática

- Não é fácil mas é muito recompensador;
- Espero que vocês tenham gostado;
- Me perguntem se quiserem mais referências;
- Me escrevam se tiverem perguntas ou sugestões;
- Me avisem caso planejem trabalhar nesta área.

Obrigado!

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.

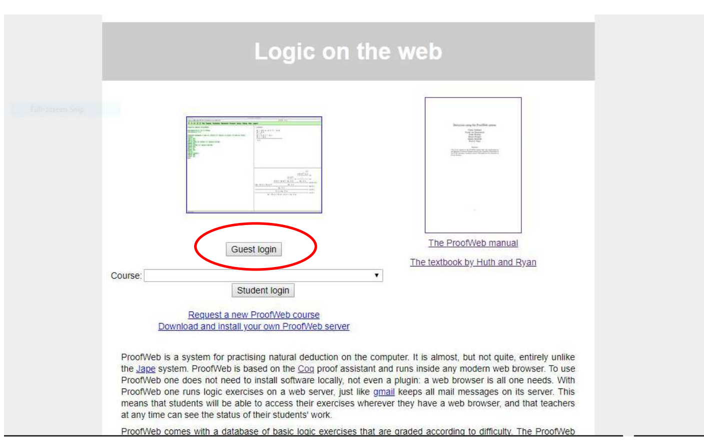
ProofWeb

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.

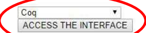
ProofWeb 1(4)

ProofWeb 2(4)



ProofWeb 3(4)

- Experiment with an empty buffer, select prover:



- 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
1ljl
201503345joaolucas.v
201508737.v
201508737keslleylima.v
201616140.v
25453

ProofWeb 4(4)

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

- 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 O).
Eval vm_compute in (plustwo (S (S O))).
```

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

- 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
| 0 => 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_0_n : ∀ n, (plus 0 n) = n.
```

basta escrever:

```
Lemma plus_0_n : ∀ n, (plus 0 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 O nil).
Check (cons (S O) (cons O 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 O)) nil)).
Eval vm_compute in (length (cons O (cons (S (S O)) nil)) ).
Eval vm_compute in (length []).
Eval vm_compute in (length [(S (S O))]).
Eval vm_compute in (length [ O ; O; O ]).
```

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 0 ; S (S 0)]).
Eval vm_compute in (cat [] []).
Eval vm_compute in (cat [0 ; 0] []).
```

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.

Conclusões

Obrigado mais uma vez!