Coq

Coq é um assistente de demonstração interactivo que permite:

- verificação de demonstrações de teoremas.
- a especificação e a obtenção de programas certificados (que satisfaçam essa especificação).
- desenvolvimento de demonstrações matemáticas numa lógica de ordem superior
- ambiente de desenvolvimento de lógicas diversas: temporais, modais, de descrição, etc.
- É baseado num λ-calculus tipificado polimórfico com tipos dependentes e noção primitiva de tipos indutivos: o Cálculo de Construções Indutivas (CoC/pCiC).
- Baseia-se no isomorfismo de Curry-Howard:
  - as demonstrações são objectos
  - as proposições (especificações) são tipos
  - uma demonstração de uma proposição P é um objecto de tipo P
- É constituído por:
  - Um núcleo de verificação de tipos e construção de contextos bem tipificados
  - Uma linguagem (funcional) de especificação (Gallina).
  - Ferramentas de ajuda à construção interactiva de demonstrações: *Tácticas* que utilizam procedimentos automá ticos de decidibilidade de teoria

Lean

É outro demonstrador interactivo também baseado no CoC/CiC `https://leanprover-community.github.io/index.html`

Como executar o Coq

- WWW: `coq.inria.fr`
- Linha de comando: `coqtop`
- Compilador: `coqc`
- Ambiente gráfico: *`coqide`*
- Emacs: `proofgeneral` módulo para vários demonstradores interactivos

Lógica Primeira Ordem Intuicionista

A interpretação construtiva de (Brower-Heyting-Komolgorov) estende-se para:

- Uma construção de $\forall x \varphi(x)$ é um método de transformar qualquer objecto **a** numa construção de $\varphi(\mathbf{a})$.
- Uma construção de $\exists x \varphi(x)$ é um par que consiste num objecto **a** e uma construção de $\varphi(\mathbf{a})$.

Quantificação universal $\forall$: conjunção generalizada...

Quantificação existêncial $\exists$: disjunção generalizada...

Mas: A quantificação universal $\forall$ também se assemelha a $\rightarrow$. Em ambos a construção são métodos.

Tipos dependentes

Generalizam os tipos $\alpha \rightarrow \beta$ que correspondem a funções cujo argumento tem tipo α e os objectos têm tipo β.

Suponhamos o tipo $string(n)$ das strings de tamanho n. Este tipo depende de $n : int$ e pode ser considerado um predicado sobre o tipo int. O *constructor* $string$ tem tipo $int \rightarrow Type$.

Podemos generalizar a outros predicados n-ários ($\tau_1 \rightarrow \cdots \tau_n \rightarrow Type$). E quantificar universalmente

No caso anterior, teríamos:

$$\forall x : int, string(x)$$

que pode ser o tipo de uma função que para qualquer inteiro n retorna uma string de tamanho n .

Produto dependente e Isomorfismo Curry-Howard

$\forall x : \tau, \sigma$ é o tipo de uma função que é aplicada a objectos de tipo τ e retorna um objecto de tipo $\sigma[x/a]$ para todo $a : \tau$.

Se x não ocorre em σ, $\forall x : \tau, \sigma$ corresponde a $\tau \rightarrow \sigma$.

Exemp. 23.1 (Exemplos de tipos dependentes (Coq)). $\forall n : int, n \leq n$

$\forall n, m : nat, n \leq m \rightarrow n \leq m + 1$

$\forall P, Q :$ `Prop`, $P \vee Q \rightarrow Q \vee P$

$nat \rightarrow nat \rightarrow$ `Prop`

$\forall n, p : nat, bin\, n \rightarrow bin\, p \rightarrow bin\, (n + p)$

$\forall n : nat, list\, n$

$\forall A : Set, A \to list\, A \to list\, A$

$\forall A, B : \texttt{Set}, A \to B \to A * B$

$\forall A, B : \texttt{Set}, A * B \to A$

Sintaxe: termos e tipos

Não há distinção sintáctica entre termos e tipos. Mas todos os termos têm tipo: mesmo os tipos.

A sintaxe do CiC é:

$$\begin{array}{rcl} s & := & Set|\ Prop|\ Type \\ t & := & s|\ x|\ c|\ C|\ I| \\ & & \forall x : t.t|\ \lambda x : t.t|\ tt|\ case(t, \ldots, t) \\ & & \texttt{fix}\, x\, x : t := t, \ldots, x : t := t \end{array}$$

onde x são variáveis, c constantes, C construtores e I tipos indutivos.

Esta sintaxe tem muita coisa nova, basta para já considerar o subconjunto de $\lambda- \to$ (sistema de tipos simples).

Expressões e Tipos

Para ter acesso às bibliotecas (standard): (`init.v`)

```
Require Import
```

Por exemplo: `Arith`, `ArithRing`, `Lists`

Verificação de tipos:

```
Check t.
```

Verifica o tipo de t no ambiente actual.

```
Check 3.
3: nat
Check plus.
plus  : nat -> nat -> nat
Check (nat ->(nat ->nat)).
nat -> nat -> nat: Set
Check (3=4).
3=4: Prop
```

Ambientes e Contextos

São caracterizados por declarações e definições.

Declarações Associa um tipo a um identificador

`Variable x:nat.`

Definições Atribui um valor a um identificador

`Definition ex1:  fun x => x+ 3.`

`Theorem impdist:  (P->Q->R)->(P->Q)->(P->R).`

`Print e.` Retorna o valor associado

Contextos Âmbito local de definições e declarações (Γ)

`Section s.  ...  End s.` s um âmbito local

Ambientes Âmbito global de definições e declarações (E)

Notações

`Scope` Permite interpretar notações ($*$, $+$, etc)

`Open Scope nat.`

`Locate n.` indica os âmbitos dessa notação.

Inferência de tipos

$E, \Gamma \vdash t : A$ atribuição de tipos

Inferência de tipos para identificadores

$$\frac{(x : A) \in E, \Gamma}{E, \Gamma \vdash x : A}\ (Var)$$

```
Check plus.
plus: nat -> nat -> nat
```

Inferência de tipos para aplicação

$$\frac{E, \Gamma \vdash e_1 : A \to B \qquad E, \Gamma \vdash e_2 : A}{E, \Gamma \vdash e_1 e_2 : B}\ (APP)$$

```
Check plus 2 3.
2 + 3: nat
```

Inferência de tipos

Uma abstração $\lambda x : A.e$ é representada por `fun x:A => e`

Inferência de tipos para abstração

$$\frac{E, \Gamma :: (x : A) \vdash e : B}{E, \Gamma \vdash \texttt{fun}\, x : A \Rightarrow e : A \to B}\ (ABS)$$

Inferência de tipos para `let`

`let` $v := t_1$ `in` t_2 corresponde a $t_2[v/t_1]$)

$$\frac{E, \Gamma \vdash t_1 : A \qquad E, \Gamma :: (v := t_1 : A) \vdash t_2 : B}{E, \Gamma \vdash \texttt{let}\, v := t_1\, \texttt{in}\, t_2 : B}\ (Let)$$

Computações –`Eval`

O COQ não é um ambiente para a execução de programas, mas é possível efectuar computações....que podem ser usadas nas tácticas...

Uma computação é obtida por reduções. Para além da redução β existem outras...

Estratégias Por valor (`cbv`) ou `lazy`

Regras de Redução

Redução-δ substituir um identificador pela definição

Redução-β $(\texttt{fun}\, x : A => e_1)e_2 \rightarrow e_1[x/e_2]$

Redução-ζ aplica o `let`

Redução-ι para tipos indutivos

```
Definition mul :=  fun n: nat -> n*n.

Eval cbv delta beta [mul] in (mul 5).
Eval compute  in (mul 5).
```

Tipos e Espécies (`sorts`)

O tipo de um tipo denomina-se **espécie** (sort).

`Set`

Os termos cujo tipo é `Set` dizem-se especificações.

Cada termo cujo tipo é uma especificação é um programa.

Se $A : Set$ e $B : Set$ então $A \rightarrow B : Set$

`Type`

O tipo do `Set` é `Type`. Embora na CiC exista uma hierarquia de tipos `Type` no COQ apenas se considera um.

`Prop`

O tipo para as proposições é `Prop`. Cada termo cujo tipo é uma proposição é uma demonstração. O tipo de `Prop` é `Type`

Se $A : Prop$ e $B : Prop$ então $A \rightarrow B : Prop$

A diferença entre um tipo `Set` e um `Prop` tem haver com a informação que é guardada ... a primeira é para programas e a segunda para proposições.

IPC($\rightarrow$) em Coq

Vamos considerar o fragmento implicacional da lógica proposicional intuicionista em COQ (`minimal.v`).

As variáveis proposicionais são de tipo `Prop`.

```
Section Minimal.
Variables P Q R T:Prop.
Check P.
```

Podia-se usar `Hypothesis` em vez de `Variable`.

No âmbito global pode-se usar `Axiom` ou `Parameter`.

`Theorem` Definição global de identificadores de tipo `Prop.` (ou `Lemma`).

```
Theorem imp_trans: (P->Q)-> (Q->R)->P->R.
```

Exemp. 23.2. *Ver*

Demonstrações dirigidas por objectivos

Objectivo P: $E, \Gamma \stackrel{?}{\vdash} P$

Tácticas: Comandos que aplicados a um objectivo, produzem uma lista possivelmente vazia de sub-objectivos.

Como se faz uma demonstração?

`Proof`

- Procura de habitantes em tipos.
- Técnica recursiva:
  - tomar um tipo no contexto
  - procurar a forma dum termo com buracos que terá esse tipo
  - recursivamente construir o termo preenchendo os buracos
- Para preencher os buracos usam-se *tácticas*: as de **introdução** criam um objectivo cuja estrutura é gerada por um construtor; de **eliminação** permitem usar factos cuja estrutura é dada por um construtor.
- Quando não há buracos, a demonstração está terminada.

Tácticas: `assumption` e `intro`

`exact x.` ou `assumption.` correspondem à regra (VAR) ou Axioma.

A táctica `intro` introduz novas hipóteses no contexto. Corresponde à aplicação da regra (ABS) (ou introdução da implicação $\rightarrow I$).

Sendo o objectivo $E, \Gamma \overset{?}{\vdash} P \to Q$

`intro H.`

Gera um sub-objectivo da forma $E, \Gamma :: (H : P) \overset{?}{\vdash} Q$.

Se t é um termo de tipo Q então `fun H : P` $\Rightarrow$ `t` é o resultado do objectivo inicial.

Variantes: `intro` v. ou `intros` $v_1 \ldots v_n$ ou `intros.` ou `intro.`

Tácticas: `apply`

Corresponde à aplicação da regra (APP) (ou eliminação da implicação $\to E$ (*Modus Ponens*)).

Tipos cabeça e final

Se t tem tipo $P_1 \to \cdots \to P_n \to Q$ então $P_k \to \cdots \to P_n \to Q$ e Q são tipos da cabeça de nível k de t. Se Q não é uma implicação então é o tipo final.

Se t é um termo de tipo $P_1 \to \cdots \to P_n \to Q$ e o objectivo é $P_k \to \cdots \to P_n \to Q$ então `apply` t. gera k-1 objectivos $P_1, \ldots, P_{k-1}$.

Se esses objectivos sucedem com termos t_1,...,t_{k-1}, então o objectivo inicial sucede com o termo $t\, t_1\, t_2 \cdots t_{k-1}$.

```
Theorem apply_ex: (Q-> R -> T) -> (P-> Q) -> P -> R -> T.
```

Estrutura das demonstrações

`Theorem x:P.` ou `Goal P.`

Mostrar os objectivos correntes

`Show.` ou `Show i.`

`Show Proof.` (termo gerado)

Refazer (n) passos

`Undo.` ou `Undo n.`

Interromper ou Reiniciar

`Abort.` ou `Restart.`

Fim

`Qed.` ou `Save id.` (se `Goal`).

Tentar demonstrar `((P -> Q) -> P) -> P`.

Teoremas *versus* Definições

Os Teoremas são de tipo `Prop` e as definições de tipo `Set`. As mesmas tácticas podem ser usadas em ambos!.

`Theorem nome:T. Proof t.`

`Definition nome:T. := t.`

A diferença essencial é a `Opacidade` e `Transparencia`:

- nas demonstrações de teoremas $x := t : T$ há certos pormenores da construção que não são relevantes (pode ser `opaca`).
- numa definição como corresponde um programa, todos os pormenores interessam para se poder extrair o programa...

Do mesmo modo podemos fazer demonstrações só usando `Section.` e `Hypothesis.` em vez da táctica `intro`.

Composição de Tácticas

Composição $tac_1; tac_2$ aplica tac_1 ao objectivo g e tac_2 a **todos** os subjectivos que a primera gera.

Se alguma falhar, falha todo o processo.

Generatização $tac_1; [tac_2| \ldots |tac_n]$ supõe que a primeira táctica gera n subobjectivos e ao i-ésimo é aplicada tac_i

Orelse $tac||tac'$ Se tac falha, aplica tac'. Se esta falha, falha tudo.

`idtac` é uma táctica que sucede sempre

`fail` é uma táctica que falha sempre

`try try` tac é equivalente a $tac||$`idtac`

Mais tácticas

`cut` se o objectivo é $\Gamma \overset{?}{\vdash} P$ e é possível obter uma demonstração de $Q \to P$ e Q então podemos usar a táctica `cut`Q

`assert` Se achamos que Q ajuda a demonstrar o objectivo P podemos fazer `assert Q`, demonstrar Q e depois usar como lema para demonstrar Q.

`auto` Se o COQ sabe como demonstrar o objectivo (isto é se há um algoritmo de decisão)

`trivial` Como o anterior mas para casos mais simples.

Produto dependente e Isomorfismo Curry-Howard

$\forall x : \tau, \sigma$ é o tipo de uma função que é aplicada a objectos de tipo τ e retorna um objecto de tipo $\sigma[x/a]$ para todo $a : \tau$.

Se x não ocorre em σ, $\forall x : \tau, \sigma$ corresponde a $\tau \to \sigma$.

Na lógica intuicionista, uma demonstração de $\forall x : A, Px$ é um método p que transforma cada $a \in X$ numa demonstração de Pa. Isto é:

$$\Pi_{x:X} Px = \{f : X \to \cup_{x:X} Px \mid \forall a : X,\ fa : Pa\}$$

Exemplos de tipos dependentes

$\forall n : nat, n \leq n$

$\forall n, m : nat, n \leq m \to n \leq m + 1$

$\forall P, Q : \texttt{Prop},\ P \ \vee \ Q \to Q \ \vee \ P$

$nat \to nat \to \texttt{Prop}$

$\forall n, p : nat, bin\, n \to bin\, p \to bin\, (n + p)$

$\forall n : nat, list\, n$

$\forall A : Set, A \to list\, A \to list\, A$

$\forall A, B : Set, A \to B \to A * B$

$\forall A, B : Set, A * B \to A$

Porquê "produto"?

No isomorfism C-H um tipo (fórmula) α é interpretado como o conjunto de demonstrações de α, $[\![\alpha]\!]$. Temos, em termos de operações entre conjuntos:

$$\begin{array}{lcl}
[\![\alpha \ \wedge \ \beta]\!] & = & [\![\alpha]\!] \times [\![\beta]\!] \\
[\![\alpha \ \vee \ \beta]\!] & = & [\![\alpha]\!] \cup [\![\beta]\!] \\
[\![\alpha \to \beta]\!] & = & [\![\alpha]\!] \to [\![\beta]\!] \\
[\![\mathrm{F}]\!] & = & \emptyset \\
[\![\forall x \alpha(x)]\!] & = & \Pi_{a:A}.[\![\alpha(a)]\!] \\
[\![\exists x \alpha(x)]\!] & = & \Sigma_{a:A}.[\![\alpha(a)]\!]
\end{array}$$

onde

$$\begin{array}{lcl}
A \times B & = & \{(a, b) \mid a \in A \text{ e } b \in B\} \\
A \cup B & = & \{(0, a) \mid a \in A\} \cup \{(1, b) \mid b \in B\} \\
A \to B & = & \{f \mid \forall a \in A\, f(a) \in B\} \\
\Pi_{a:A}.Pa & = & \{f : A \to \cup_{a:A} Pa \mid \forall a : A,\ f(a) \in Pa\} \\
\Sigma_{a:A}.Pa & = & \{(a, p) \mid a \in A \text{ e } p \in Pa\}
\end{array}$$

e onde $\{Pa\}_{a:A}$ é uma familia indexada de conjuntos.

Inferência de Tipos

Aplicação

$$\frac{E, \Gamma \vdash t_1 : \forall x : A, B \qquad\qquad E, \Gamma \vdash t_2 : A}{E, \Gamma \vdash t_1\, t_2 : B[x/t_2]}\ (APP)$$

Abstração

$$\frac{E, \Gamma :: (x : A) \vdash t : B}{E, \Gamma \vdash \texttt{fun}\, x : A \Rightarrow t : \forall x : A, B}\ (ABS)$$

Se x não ocorre em B então $\forall x : A, B$ é equivalente a $A \to B$

Exemplos

```
Check le_n.
le_n : forall n : nat, n <= n

Check le_S.
le_S: forall n m : nat, n <= m -> n <= S m

Definition le_36_37 := le_S 36 36 (le_n 36).

Check (le_S _ _ (le_S _ _ (le_n 36))).

Theorem le_i_SSi : forall i:nat, i <= S (S i).
Proof (fun i:nat => le_S _ _ (le_S _ _ (le_n i))).
```

Argumentos implícitos

```
Definition compose : forall A B C : Set, (A->B)->(B->C)->A->C
   := fun A B C f g x => g (f x).
```

Podemos usar variáveis anónimas:

```
Check (fun (A:Set)(f:Z->A) => compose _ _ _ Z_of_nat f).
```

Ou definir logo com

```
Reset compose.
Set Implicit Arguments.
Definition compose (A B C:Set)(f:A->B)(g:B->C)(a:A):= g(f a).
```

Lógica em Coq: dedução natural para lógica de primeira ordem

As conectivas $\to$ e $\forall$ são intrínsecas ao sistema de tipos do Coq e portanto as regras de introdução e eliminação correspondem às regras de inferência. As tácticas correspondentes são `intro` e `apply`.

```
Theorem imp_trans:forall P Q R:Prop,(P->Q)->(Q->R)->P ->R.
Proof.
intros P Q R H H0 p.
apply H0. apply H. assumption.
Qed.

Theorem all_imp_dist: forall (A:Type)(P Q:A->Prop),
(forall x:A, P x -> Q x)->(forall y:A, P y)->(forall z:A, Q z).
```

As restantes conectivas são definidas indutivamente (vamos ver mais tarde), mas podemos usá-las sabendo para já quais os constructores e os seus tipos, e quais as tácticas correspondentes.

Dedução natural para LP

	Introdução	**Eliminação**
$\wedge$	$\dfrac{\phi \quad \psi}{\phi \wedge \psi}\wedge\mathbf{I}$	$\dfrac{\phi \wedge \psi}{\phi}\wedge\mathbf{E}_1 \qquad \dfrac{\phi \wedge \psi}{\psi}\wedge\mathbf{E}_2$
$\vee$	$\dfrac{\phi}{\phi \vee \psi}\vee\mathbf{I}_1 \qquad \dfrac{\psi}{\phi \vee \psi}\vee\mathbf{I}_2$	$\dfrac{\phi \vee \psi \qquad \begin{matrix}[\phi] \\ \vdots \\ \gamma\end{matrix} \qquad \begin{matrix}[\psi] \\ \vdots \\ \gamma\end{matrix}}{\gamma}\vee\mathbf{E}$
F	$\dfrac{\begin{matrix}[\phi] \\ \vdots \\ \neg\phi\end{matrix}}{\mathrm{F}}\mathrm{F}\mathbf{I}$	$\dfrac{\mathrm{F}}{\phi}$
$\rightarrow$	$\dfrac{\begin{matrix}[\phi] \\ \vdots \\ \psi\end{matrix}}{\phi\rightarrow\psi}\rightarrow\mathbf{I}$	$\dfrac{\phi \quad \phi\rightarrow\psi}{\psi}\rightarrow\mathbf{E}$
$=$	$\dfrac{}{t=t}=\mathbf{I}$	$\dfrac{t_1=t_2 \quad \phi[t_1/x]}{\phi[t_2/x]}=\mathbf{E}$ e x é substituível por t_1 e por t_2 em ϕ
$\forall$	$\dfrac{\begin{matrix}[v] \\ \vdots \\ \phi[v/x]\end{matrix}}{\forall x\ \phi}\forall\mathbf{I}$ onde v é uma variável nova (não ocorre antes)	$\dfrac{\forall x\ \phi}{\phi[t/x]}\ \forall\mathbf{E}$ onde x é substituível por t em ϕ
$\exists$	$\dfrac{\phi[t/x]}{\exists x\ \phi}\ \exists\mathbf{I}$ onde x é substituível por t em ϕ	$\dfrac{\exists x\ \phi \qquad \begin{matrix}[v \quad \phi[v/x]] \\ \vdots \\ \psi\end{matrix}}{\psi}\ \exists\mathbf{E}$ onde v é uma variável nova que não ocorre antes nem em ψ

Conectivas lógicas

$\wedge$:

I: `conj` $\forall A\, B : Prop,\ A \rightarrow B \rightarrow A \wedge B$ (`split`)

E: `and_ind` $\forall A\, B\, P : Prop, (A \rightarrow B \rightarrow P) \rightarrow A \wedge B \rightarrow P$ (`elim`)

$\vee$:

I: `or_introl` $\forall A\, B : Prop, A \rightarrow A \ \vee\ B$ (`left`)

`or_intror` $\forall AB : Prop, B \to A \ \vee \ B$ (`right`)

E: `or_ind` $\forall A\, B\, P : Prop, (A \to P) \to (B \to P) \to A \ \vee \ B \to P$ (`elim`)

Conectivas lógicas

F `False`

E: $\forall P : Prop, False \to P$ (`False_ind`)

$\neg$ $\neg P$ é definido por $P \to$ F

E: (`elim`)

Existencial

$\exists$ `ex`: $\forall A : Type (A \to Prop) \to Prop$

`ex P` é $\exists x.Px$

I: `ex_intro` $\forall (A : Type)(P : A \to Prop)(x : A), Px \to ex\, P$

(`exists` v)

E: `ex_ind` $\forall A\, P\, P0 : (\forall x : A,\ P\, x \to P0) \to ex\, P \to P0$

(`elim`)

```
Lemma ex_imp_ex :
 forall(A:Type)(P Q:A->Prop)(ex P)->
           (forall x:A,Px->Qx)->(ex Q).
```

Igualdade

$=$ `eq`: $\forall (A : Type),\ A \to A \to Prop$

I: `refl_equal` $\forall (A : Type)(x : A),\ x = x$ (`reflexivity`)

E: `eq_ind` (`rewrite`)

$$\forall (A : Type)(x : A)(P : A \to Prop), Px \to \forall (y : A), x = y \to Py$$

Se e é um termo de tipo $\forall (x_1 : T_1) \cdots (x_n : T_n), a = b$ e o objectivo é da forma $P\, a$ a táctica `rewrite` e, gera um sub-objectivo da forma $P\, b$.

```
Theorem eq_sym' : forall (A:Type)(a b:A), a=b->b=a.
```

Variantes No `rewrite` pode-se orientar a igualdade: `rewrite <-` e ou `rewrite ->` e. Pode-se escolher a hipótese, etc...

Tácticas

Cada conectiva lógica é tratada por dois géneros de tácticas, uma para uso em hipóteses - tácticas de eliminação - e outras para uso em objectivos - tácticas de introdução.

	$\rightarrow$	$\forall$	$\wedge$	$\vee$	$\exists$
Hipóteses	`apply`	`apply`	`elim`	`elim`	`elim`
Objectivos	`intros`	`intros`	`split`	`left` ou `right`	`exists` v
	$\neg$	$=$			
Hipóteses	`elim`	`rewrite`			
Objectivos	`intro`	`reflexivity`			

onde:

`split` é equivalente a `intros; apply conj`

`left` é equivalente a `intros; apply or_introl`

analogo para `right.`

Tipos Indutivos

São definidos por

- um nome
- um tipo (ou família de tipos)
- construtores
- processo de computação: por casos ou recursão
- princípios de indução

Pattern-matching permite a descrição de funções por casos

Tipos Indutivos

- Enumerações

```
Inductive dias: Set := Seg: dias | Ter:dias | Qua:dias
           Qui: dias | Sex: dias | Sab: dias | Dom: dias.

Check dias_ind.
```

- Records

```
Inductive plane: Set := point : Z -> Z -> plane.

Record plane: Set := point { abs: Z; ord:Z}.
```

Tipos Indutivos

- Records com variantes

```
Inductive vehicle : Set :=
  | bicycle : nat -> vehicle
  | motorized : nat -> nat -> vehicle.
```

- Recursivos:

```
Inductive nat : Set := 0: nat | S: nat ->nat.
```

- Polimórficos:

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

Tácticas

`pattern m` $\Rightarrow$ $(\lambda x.P(x))m$

case Se t tem um tipo indutivo , `case` t substitui todas as instâncias de t no objectivo com todos os casos possíveis.

destruct Análoga, mas apenas para tipos não recursivos

apply `apply T_ind`

elim A táctica `elim` faz a ligação entre o tipo indutivo e o princípio indutivo correspondente. Se t é um termo de tipo T, de acordo com a espécie s do objectivo `elim t`, escolhe o princípio, `T_ind`, `T_rec` ou `T_rect`. Todos estes princípios têm uma variável quantificada universalmente P de tipo `T ->s` e terminam com em $\forall x : T.(Px)$. Equivale a

```
pattern m; apply T_ind
```

`match`

Construir funções com análise de casos.

```
Definition month_length (leap:bool)(m:month) : nat :=
  match m with
  | January => 31 | February => if leap then 29 else 28
  | March => 31   | April => 30    | May => 31  | June => 30
  | July => 31    | August => 31   | September => 30
  | October => 31 | November => 30 | December => 31
  end.
```

```
Definition nb_wheels (v:vehicle) : nat :=
  match v with
  | bicycle x => 2
  | motorized x n => n
  end.
```

Mais Tácticas

`induction` `induction t` permite que t não esteja no contexto: `intros until t; elim t` seguida de `intros`...

`simpl` realiza reduções $\beta, \iota, ...$

`change` permite substituir um objectivo por outro convertível

`discriminate` **e** `injection`

Para lidar com termos que se pretendem ou não equivalentes

Relaciona igualdade com os tipos indutivos:

- os dois construtores não são iguais e
- os construtores são injectivos

Não podem ser usados com termos da classe `Prop`.

`rewrite`

Se t é do tipo $\forall(x_i : T_i)_{i=1..n}, a = b$ e o objectivo é da forma Pa, a táctica `rewrite t` dará um sub-objectivo Pb. Pode-se indicar a direcção em que a reescrita é aplicada.

Tipos Recursivos

Funções recursivas

$$\texttt{Fixpoint}\ f(x_1 : T_1) : T = expr$$

Mais geral

$$\texttt{Fixpoint}\ f(x_1 : T_1)...(x_n : T_n)\ \{\texttt{struct}\ x_i\} : T = expr$$

mas é necessário que a recursão termine!

```
Fixpoint mult2 (n:nat) : nat :=
   match n with
     0 => 0
   | S p => S (S (mult2 p))
   end.
```

Tipos Indutivos – `nat`

```
nat
```

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

```
Fixpoint plus (n m:nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (p + m)
  end

Lemma plus_n_O : forall n:nat, n = n + 0.
Lemma plus_n_Sm: forall n m:nat, S n + m = S (n+m).

Fixpoint mult (n m:nat) {struct n} : nat :=
  match n with
  | O => 0
  | S p => m + mult (p  m)
  end
```

Relações como tipos indutivos

```
Inductive le (n : nat) : nat -> Prop :=
    le_n : n <= n
  | le_S : forall m : nat, n <= m -> n <= S m
```

Tácticas: `constructor n.` Provar que

```
Theorem zz: 0<= 0.
Proof.
constructor 1.
Qed.

Theorem ut: 1<= 3.
Proof.
constructor 2.
constructor 2.
constructor 1.
Qed.
```

ou usar só `repeat constructor.`

Tipos Indutivos Polimórficos - `List`, `option`, `prod`

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

Exerc. 23.1. *1. Concatenação de listas*

2. Dada uma lista retornar uma lista com os 2 primeiros elementos

3. Dada uma lista e n retornar uma lista com os n primeiros elementos

4. Dada uma lista de inteiros retornar a sua soma.

5. Dado n retornar uma lista com os n primeiros inteiros.

◇

Tipos Polimórficos

```
Fixpoint app (A:Set)(l m:list A){struct l} : list A :=
  match l with
  | nil => m
  | cons a l1 => cons a (app A l1 m)
  end.
```

`option` –**funções parciais**

```
Inductive option (A : Type) : Type :=  Some : A -> option A
| None : option A
```

`prod` – **pares**

```
Inductive prod (A : Type) (B : Type) : Type :=
 pair : A -> B -> A * B
```

Tipos Indutivos Dependentes - `ltree`

Árvores binárias

```
Inductive Z_btree : Set :=
Z_leaf: Z_btree | Z_bnode: Z-> Z_btree-> Z_btree ->Z_btree.
```

Exerc. 23.2. • *Determinar a soma de todos os nós duma árvore binária*

• *Determinar se uma árvore binária tem algum zero.*

◇

Árvore com nós menores que um dado n

```
Inductive ltree (n:nat) : Set:=
| lleaf : ltree n
| lnode : forall p:nat, p<= n -> ltree n -> ltree n -> ltree n.
```

Tipos Indutivos Dependentes - `htree`

Árvores com ramos do mesmo comprimento

```
    Inductive htree (A:Set) : nat->Set :=
  | hleaf : A->htree A 0
  | hnode : forall n:nat, A -> htree A n -> htree A n
              -> htree A (S n).


Fixpoint htree_to_btree (n:nat)(t:htree Z n){struct t}:
Z_btree :=
  match t with
  | hleaf x => Z_bnode x Z_leaf Z_leaf
  | hnode p v t1 t2 =>
      Z_bnode v (htree_to_btree p t1)(htree_to_btree p t2)
  end.
```

Predicados Indutivos (propriedades)

Par

```
    Inductive even : nat->Prop :=
  | O_even : even 0
  | plus_2_even : forall n:nat, even n -> even (S (S n)).
```

Lista ordenada

```
Inductive sorted (A:Set)(R:A->A->Prop) : list A -> Prop :=
  | sorted0 : sorted A R nil
```

```
  | sorted1 : forall x:A, sorted A R (cons x nil)
  | sorted2 :
      forall (x y:A)(l:list A),
        R x y ->
        sorted A R (cons y l)-> sorted A R (cons x (cons y l)).
```

Construção de Predicados indutivos

- Os construtores são axiomas
- Os construtores devem corresponder a casos mutuamente exclusivos

Demonstrações por indução usando Predicados indutivos

```
Theorem sum_even : forall n p:nat, even n -> even p
               -> even (n+p).
Proof.
 intros n p Heven_n; elim Heven_n.
 trivial.
 intros x Heven_x Hrec Heven_p; simpl.
 apply plus_2_even; auto.
Qed.


Theorem lt_le : forall n p:nat, n < p -> n <= p.
Proof.
 intros n p H; elim H; repeat constructor; assumption.
Qed.
```

Conectivas lógicas

Verifica os princípios de indução `_ind`

True

```
Inductive True : Prop :=  I : True
```

False

```
Inductive False : Prop :=.
```

$\wedge$

```
Inductive and (A : Prop) (B : Prop) : Prop :=
conj : A -> B -> A /\ B
```

$\vee$

```
Inductive or (A : Prop) (B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B
```

Conectivas lógicas

Verifica os princípios de indução `_ind`

$\exists$

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : forall x : A, P x -> ex P
```

$=$

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
refl_equal : x = x
```

Funções recursivas como Predicados

Um dos problemas é garantir a terminação!

As definições indutivas permitem introduzir restrições que garantam a terminação:

$f : A \to B$ pode ser descrita pelos pares $(x, f(x))$ por um predicado de tipo $A \to B \to Prop$.

```
Inductive Pfact : Z->Z->Prop :=
  Pfact0 : Pfact 0 1
| Pfact1 : forall n v:Z, n <> 0 -> Pfact (n-1) v -> Pfact n (n*v).
```

$Pfact\, n\, m$ a computação do factorial de n termina e retorna m.

```
Theorem pfact3 : Pfact 3 6.
```

Podemos determinar o domino e o contradomínio(***):

```
Theorem fact_def_pos:forall x y:Z,Pfact x y ->  0 <= x.
Theorem Zle_Pfact:forall x:Z,0<= x -> exists y:Z,Pfact x y.
```

`inversion`

```
Theorem not_even_1 : ~even 1.
Proof.
 unfold not; intros H.
 inversion H.
Qed.
```

Se se usasse `elim H` não era possível unificar objectivo (`False` com o predicado P do principio indutivo `even_ind`). Para aplicar `inversion` é necessário que uma hipótese tenha o tipo indutivo. No exemplo, os construtores `O_even` e `plus_2_even` são considerados para `1`, e ambos falham (`discriminate`).

`inversion`

Usada quando se pretende raciocinar negativamente sobre os construtores de um predicado indutivo.

Esta táctica é equivalente a usar `generalize e` (que modifica o objectivo `O` para `e -> O`) seguida de `pattern` (para obter P) e **discriminate**s (caso e seja uma igualdade).

No entanto, novos objectivos podem ser gerados:

```
Theorem plus_2_even_inv: forall n:nat,even (S (S n))->even n.
Proof.
 intros n H; inversion H.
 assumption.
Qed.
```

Tácticas (resumo)

`intros`: transforma um objectivo com implicações e/ou quantificação universal num objectivo mais simples, onde as hipóteses e variáveis quantificadas vão para o contexto.

`apply H`: aplica `H` ao objectivo corrente, adicionando as premissas de H como subobjectivos.

`change e` Se o objectivo é `e'`, muda o objectivo para `e`. Mas `e` e `e'` têm de ser equivalentes por computações.

Tácticas para a igualdade

`reflexivity`: igualdade entre dois termos equivalentes

`rewrite H`: Reescreve o objectivo usando a igualdade da hipótese `H`. `rewrite H1 with H2` efectua a reescrita em `H2`. `rewrite <- H1` utiliza a igualdade da direita para a esquerda.

Tácticas

Tácticas de simplificação

`simpl`: simplifica o objectivo usando regras de redução computacionais (conversões). `simpl in H` simplifica a hipótese `H` ou `simpl in *` para simplificar o objectivo e todas as hipóteses.

`unfold nome`: Sendo `nome` é uma definição, expande todas as ocorrências do nome no objectivo. Pode ser usada nas hipóteses como `simpl`. Cuidado que pode complicar o objectivo...

Tácticas para tipos indutivos

`case`: raciocíno por casos em `e`

`elim`: raciocínio indutivo

`discriminate` Demonstra uma fórmula desde que nas hipóteses exista uma igualdade entre construtores de um tipo indutivo (que devem corresponder a termos distintos...)

`injection H` Se `H` é uma hipótese que iguala dois valores com o mesmo construtor, adiciona as igualdades que são geradas por ela (construtores são injectivos).

Tácticas

Tácticas compostas

`desctruct e` raciocínio por casos em `e`, criando um sub-objectivo para cada constructor do tipo de `e` e substituindo `e` pela aplicação desse construtor a uma variável nova. (Pode utilizar `intros` antes).

`induction e` Igual ao anterior, mas os sub-objectivos podem ter hipóteses adicionais. `e` pode ser uma variável quantificada no objectivo.

Tácticas automáticas

`trivial` Demonstra objectivos simples

`tauto` Demonstra tautologias proposicionais `auto` e `eauto`

`Hint Resolve` adiciona termos a uma base de dados de tácticas

`intuition`

`autorewrite`

`Hint Rewrite`

`congruence` Demonstra objectivos que são consequência de igualdades e propriedades básicas dos construtores

Tácticas

Tácticas numéricas

Correspondem a resolutores automáticos nos respectivos conjuntos.

Naturais `nat`, Inteiros $\mathbb{Z}$ e Reais $\mathbb{R}$

`ring` Para inteiros, resolve equações polinomiais em anéis ou semi-anéis

`omega` Sistemas de equações lineares e inequações em `nat` e $\mathbb{Z}$

`fourier` Sistemas de equações lineares e inequações para reais.

Referências

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. SV, 2004.

[PU96] Morten B. Sorensen Pawel Urzyczyn. Lecture on the curry-howard isomorphism. Technical report, University of Copenhagen, 1996.