Introdução ao Sistema Coq de Assistência à Prova

Elementos Lógicos da Programação III

Jorge Sousa Pinto (*jsp@di.uminho.pt*)
Maria João Frade (*mjf@di.uminho.pt*)

Departamento de Informática
Universidade do Minho
1998

Conteúdo

1 Introdução

O presente texto resulta da compilação dos guiões elaborados para as aulas laboratoriais da disciplina de Elementos Lógicos de Programação III, leccionada no segundo semestre do ano lectivo de 1997/98 ao 3º ano da Licenciatura em Matemática e Ciências da Computação da Universidade do Minho.

Pretende ser uma introdução "hands-on" ao sistema **Coq** na sua versão 6.1, cobrindo os tópicos mais importantes desde a teoria de tipos subjacente até à extracção de programas de especificações, passando pela demonstração de teoremas e pela utilização de tipos indutivos.

A interacção com o sistema é imprescindível para a leitura deste documento, uma vez que ele não contém qualquer texto correspondente a respostas às interacções propostas.

O texto encontra-se organizado em 7 capítulos, cada um deles correspondendo a uma sessão laboratorial. Em cada sessão são propostos alguns exercícios simples para resolução, encontrando-se alguns deles numerados e identificados como **Questões**. Estas questões destinaram-se a ser resolvidas durante as aulas laboratoriais, e foram utilizadas como elementos de avaliação.

1.1 O Sistema Coq

O Coq é um sistema de prova assistida para lógicas de ordem superior, que permite o desenvolvimento de programas de modo consistente com a sua especificação (programas certificados). Este sistema baseia-se no Cálculo de Construções Indutivas.

Para entrar no sistema Coq[1] faça:

```
coqtop
```

Vamos, portanto, trabalhar em λ-Calculus com tipos. Vejamos como se representam as expressões λ em Coq:

λ-Calculus	**Coq**
$\lambda x:A.t$	$[x:A]t$
$t\,s$	$(t\,s)$
$t\,s_1\,s_2\ldots s_n$	$(t\,s_1\,s_2\ldots s_n)$
$A \rightarrow B$	$A \rightarrow B$
$\Pi x:A.B$	$(x:A)B$

O Coq divide o universo proposicional, `*`, em duas categorias `Set` e `Prop` , consoante os seus habitantes são vistos como conjuntos concretos ou como proposições. O universo $\Box$ é representado em Coq por `Type`.

[1] Certifique-se que `/usr1/ML/ocaml/COQ/bin` se encontram na sua `PATH`. Caso tal não se verifique inclua o comando `export PATH=${PATH}:/usr1/ML/ocaml/COQ/bin` no fim do seu ficheiro `~/.profile` ou `~/.bashrc`.

2 Declarações, Definições, Inferência de Tipos Simples e Redução de Termos.

2.1 Declarações locais e globais. O mecanismo de secções

Comecemos por efectuar algumas declarações globais. Cada uma declara um nome para um g-termo, associando-lhe um tipo. Declaremos, antes de mais, dois tipos proposicionais:

```
Coq < Variable A : Set.
Coq < Variable B : Set.
```

As declarações feitas com `Variable` são *locais*, ou seja, têm validade num contexto cujo início e fim o utilizador define. Em Coq, estes contextos tomam o nome de **secções**. Como não especificámos ainda nenhuma secção para a validade das declarações acima, elas são válidas *globalmente*, ou seja, em todos os contextos.

Iniciemos agora um novo contexto, e declaremos nesse contexto um termo de um dos tipos acima.

```
Coq < Section s1.
Coq < Variable t : A.
```

Note que o tipo `A` é conhecido dentro da secção `s1`.

Apesar de estarmos dentro do contexto `s1` podemos efectuar declarações globais em qualquer momento, com o comando `Parameter`:

```
Coq < Parameter g : B.
```

2.2 Inferência de Tipos

Voltaremos brevemente ao mecanismo de secções, mas antes disso vejamos como efectuar em Coq a síntese do tipo de um termo. O comando `Check` tem precisamente essa função:

```
Coq < Check [x:A] x.
Coq < Check [x:A] [y:A->B] (y x).
```

Os dois termos cujo tipo pedimos ao sistema para inferir correspondem, na notação tradicional do λ-calculus, a $(\lambda x{:}\,A.x)$ e $(\lambda x{:}\,A.\lambda y{:}\,A \to B.yx)$. Note que se trata de termos do λ-calculus com tipos simples (designado $\lambda_{\to}$), e observe a forma como o Coq infere e representa os seus tipos. De facto, tratando-se de termos anotados com tipos, este processo é simples.

De notar ainda que as variáveis `x` e `y` usadas nas abstracções dos termos acima representados, sendo locais a essas abstracções, não necessitam de ser declaradas no contexto corrente. Se tentarmos:

```
Coq < Check x.
```

o sistema imprimirá uma mensagem de erro.

2.3 Definições locais e globais

Em seguida, vamos efectuar algumas *definições*. Nestas, podemos associar a um qualquer g-termo um nome, novamente de forma local ou global, respectivamente com os comandos `Local` e `Definition`.

```
Coq < Definition id_A := [x:A] x.
Coq < Local K_AB := [x:A] [y:B] x.
```

Sobre estes termos podemos efectuar a mesma operação de extracção do tipo, ou alternativamente utilizar o comando `Print` para visualizar o termo associado a um identificador:

```
Coq < Check id_A.
Coq < Print K_AB.
```

O comando `Print All.` permite visualizar informação sobre todos os identificadores declarados e definidos com visibilidade no contexto actual. O comando `Inspect n.` permite visualizar os `n` últimos identificadores declarados ou definidos. Naturalmente, no caso dos identificadores declarados, apenas o seu tipo está disponível.

```
Coq < Inspect 7.
```

2.4 Secções Revisitadas

Procedamos agora ao *fecho* da secção corrente:

```
Coq < End s1.
```

Qual o efeito do fecho da secção `s1` ?

Avaliemos agora a seguinte sequência de comandos:

```
Coq < Section s2.
Coq < Variable x : A.
Coq < Local tlocal := [y:B] x.
Coq < Definition tglobal := [y:B] x.
Coq < Definition ttglobal := (tlocal g).
Coq < Print tlocal.
Coq < Print tglobal.
Coq < Print ttglobal.
Coq < End s2.
Coq < Print tlocal.
Coq < Print tglobal.
Coq < Print ttglobal.
```

A interpretação do papel das secções como contextos de juízos do sistemas de tipos é imediata: cada secção corresponde à sequência de declarações que forma o lado esquerdo de cada juízo de tipagem de um termo. Assim, o juízo

$$t_1\colon \sigma_1, t_2\colon \sigma_2, \ldots, t_i\colon \sigma_n \vdash t : \sigma.$$

será válido na teoria de tipos do Coq se numa secção onde sejam válidas apenas as declarações presentes no lado esquerdo do sinal $\vdash$, o comando `Check` aplicado ao termo t inferir o tipo σ para aquele termo.

O *fecho* de uma secção em Coq efectuará assim o fecho do λ-termo t, de acordo com a regra de abstracção dos sistemas de tipos.

Para concluir o estudo das secções, relembre a definição dos combinadores básicos K e S, e avalie a seguinte sequência de comandos

```
Coq < Section s3.
Coq < Variables sigma, tau, delta : Set.
Coq < Definition K_SigmaTau := [x:sigma][y:tau] x.
Coq < Definition S_SigmaTauDelta :=
                    [x:sigma->tau->delta][y:sigma->tau][a:sigma](x a (y a)).
```

```
Coq < Print K_SigmaTau.
Coq < Print S_SigmaTauDelta.
Coq < End s3.
Coq < Print K_SigmaTau.
Coq < Print S_SigmaTauDelta.
```

Comente o efeito, e a resposta do sistema, à sequência anterior.

2.5 Redução de Termos

Vejamos agora como poderemos efectuar em Coq a redução-β de termos do λ-calculus:

```
Coq < Variable b : A.
Coq < Definition f := (([a:A] a) b).
Coq < Eval f.
Coq < Eval (([a:A] a) b).
Coq < Compute f.

Coq < Variable a : A.
Coq < Eval (id_A a).
Coq < Compute (id_A a).

Coq < Eval ([C:Set]C A).
Coq < Definition ff := [C:Set][D:Set][x:C][y:C->D](y  x).
Coq < Check ff.
Coq < Compute (ff A B b).
```

O comando `Eval` efectua redução-β até à forma normal, sem no entanto proceder à substituição de identificadores pelos termos que eles representam. Para forçar essa substituição podemos utilizar o comando `Compute`.

2.6 Outros Exemplos

Observe que as seguintes tentativas de extracção de tipo e de redução falham, dada a má formação dos termos $(\lambda x\colon A.xx)$ e $(\lambda x\colon A.x)(\lambda x\colon A.x)$.[2]

```
Coq < Check ([x:A](x x)).
Coq < Check (([x:A]x)([x:A]x)).
Coq < Eval ([x:A]x [x:A]x).
```

[2] No λ-calculus sem tipos os termos resultantes destes depois de eleminadas as anotações de tipos são bem formados.

Naturalmente, um termo para o qual não existe tipo na teoria do Coq é um termo mal formado, pelo que não pode ser reduzido.

A sequência de comandos que se segue também falha devido à mal formação dos termos envolvidos.

```
Coq < Check [x:A] t.
Coq < Check [y:A->C] (y a).
Coq < Eval (([y:A->C] y) a).
```

Note que os identificadores `t` e `C` não estão definidos no contexto.

Exercício: Considere a seguinte função do λ-calculus com tipos simples, que efectua a aplicação de uma função composta consigo mesma a um argumento:

$$double \doteq \lambda f\colon A \to A.\lambda x\colon A.f(fx)$$

Efectue em Coq a definição de um g-termo que represente esta função *double*, verifique qual é o seu tipo, e aplique-a com a função identidade (do tipo A) a um elemento de tipo A. Qual é o resultado?

Para sair do Coq faça:

```
Coq < Quit.
```

3 O λ-Cubo

As entidades bem formadas de uma Teoria de Tipos, são determinadas pelo conjunto de *universos* e pelo conjunto de *regras de inferência* que determinam quais os *juizos* $\Gamma \vdash t : T$ que são válidos[3]

Uma vez definidas estas duas classes de entidades, os termos generalizados (*g-termos*) bem formados ficam completamente determinados. Note-se que estes termos representam igualmente termos e tipos; portanto ficam determinados os termos e os tipos da Teoria de Tipos que estivermos a caracterizar.

Com um conjunto de dois universos $\{*, \Box\}$ e o conjunto de regras a seguir apresentadas, definem-se oito sistemas que podem ser organizados coerentemente nos vértices de um cubo.

Sistemas do λ-Cubo

Sejam $s, s_1, s_2 \in \{*, \Box\}$.

Regras Gerais

(axioma) $\quad \vdash * : \Box$

(assunção) $$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \;,\; x \notin \Gamma$$

(enfraquecimento) $$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \;,\; x \notin \Gamma$$

(aplicação) $$\frac{\Gamma \vdash F : (\Pi\, x{:}A.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]}$$

(abstracção) $$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi\, x{:}A.\, B) : s}{\Gamma \vdash (\lambda x{:}A.b) : (\Pi\, x{:}A.\, B)}$$

(conversão) $$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}$$

Regras específicas

regra $\langle s_1, s_2 \rangle$ $$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi\, x{:}A.\, B) : s_2}$$

<u>Nota</u>: $\quad (\Pi\, x{:}A.\, B) \;\equiv\; A \to B \quad , \quad \text{se} \;\; x \notin \mathcal{L}(B)$

A regra $\langle s_1, s_2 \rangle$ tem várias instâncias consoante s_1 e s_2 tomam o valor de $*$ ou $\Box$. Cada par de universos $\langle s_1, s_2 \rangle$ que determina uma regra de formação Π dá origem a uma *dependência*. Cada sistema é gerado pelas regras de inferência gerais e algumas regras específicas (de acordo com as dependências permitidas).

[3] Vendo o sistema de tipos como uma lógica, as suas proposições (fórmulas) são os juizos $\Gamma \vdash t : T$.

Os sistemas de tipos que iremos considerar contêm todos a dependência $\langle *, * \rangle$. Consoante contém ou não cada uma das restantes três dependências ($\langle *, \square \rangle, \langle \square, * \rangle$ e $\langle \square, \square \rangle$) identificam-se oito *Sistemas Abstractos de Tipos* diferentes, a que se dão nomes específicos (ver quadro).

Dependências

$\lambda_{\to}$	$\langle *, * \rangle$			
$\lambda 2$	$\langle *, * \rangle$	$\langle \square, * \rangle$		
λP	$\langle *, * \rangle$		$\langle *, \square \rangle$	
$\lambda P2$	$\langle *, * \rangle$	$\langle \square, * \rangle$	$\langle *, \square \rangle$	
$\lambda \underline{\omega}$	$\langle *, * \rangle$			$\langle \square, \square \rangle$
$\lambda \omega$	$\langle *, * \rangle$	$\langle \square, * \rangle$		$\langle \square, \square \rangle$
$\lambda P \underline{\omega}$	$\langle *, * \rangle$		$\langle *, \square \rangle$	$\langle \square, \square \rangle$
λC	$\langle *, * \rangle$	$\langle \square, * \rangle$	$\langle *, \square \rangle$	$\langle \square, \square \rangle$

Cada uma das "dependências extra" determina sistemas diferentes designados por: $\lambda 2$ (*termos dependentes de tipos*), λP (*tipos dependentes de termos*) e $\lambda \underline{\omega}$ (*tipos dependentes de tipos*). Estes três sistemas podem considerar-se como os três eixos de um sistema tridimensional, o que permite visualizar os oito sistemas como os vértices de um cubo – o λ-*Cubo*.

O sistema de tipos mais abrangente, onde todas as dependencias são válidas é o sistema λC chamado *Cálculo das Construções*. É este o sistema que o Coq implementa.

3.1 Termos generalizados

Vamos começar pela seguinte definição

```
Coq < Definition id := [A:Set][x:A] x.
```

Acabamos de definir o g-termo: $id \doteq (\lambda A\colon *. \lambda x\colon A.x)$. Qual é o tipo de id ?

```
Coq < Check id.
```

Como o próprio Coq deve confirmar, id tem tipo $(\Pi A\colon *. A \to A)$. Podemos descrever o raciocínio que nos permite chegar a esta conclusão por

$$\begin{array}{rcl} x & : & A \\ (\lambda x\colon A.x) & : & A \to A \\ (\lambda A\colon *. \lambda x\colon A.x) & : & (\Pi A\colon *. A \to A) \end{array}$$

Vejamos como poderia ser feita a prova formal do juizo $\vdash (\lambda A\colon *. \lambda x\colon A.x) : \Pi A\colon *. A \to A$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash * : \square}{A : * \vdash A : *}}{A : *, x : A \vdash x : A} \quad \dfrac{\dfrac{\vdash * : \square}{A : * \vdash A : *} \quad \dfrac{\vdash * : \square}{A : * \vdash A : *}}{A : * \vdash A \to A : *}}{A : * \vdash (\lambda x\colon A.x) : A \to A} \quad \dfrac{\vdash * : \square \quad \dfrac{\dfrac{\vdash * : \square}{A : * \vdash A : *} \quad \dfrac{\vdash * : \square}{A : * \vdash A : *}}{A : * \vdash A \to A : *}}{\vdash (\Pi\, A{:}*.\, A \to A) : *}}{\vdash (\lambda A\colon *. \lambda x\colon A.x) : (\Pi\, A{:}*.\, A \to A)}$$

Repare que para fazermos tal prova, o sistema deve dispôr das dependências $\langle *, * \rangle$ e $\langle *, * \rangle$. As provas formais da validade dos juizos são normalmente muito extensas.

Podemos agora fazer

```
Coq < Variable T : Set.
Coq < Variable t : T.
Coq < Compute (id T t).
```

Repare na cadeia de dedução que justifica este resultado:

$$(id\, T\, t) \equiv (\lambda A{:}\, *.\lambda x{:}\, A.x)\, T\, t \rightarrow (\lambda x{:}\, T.x)\, t \rightarrow t$$

Considere agora a seguinte definição

```
Coq < Definition f := [C:Set][D:Set][x:C][y:C->D](y x).
```

O g-termo acabado de definir é $(\lambda C{:}\, *.\lambda D{:}\, *.\lambda x{:}\, C.\lambda y{:}\, C \rightarrow D.yx)$. Qual o seu tipo ?

$$\begin{array}{rcl}
y & : & C \rightarrow D \\
x & : & C \\
yx & : & D \\
(\lambda y{:}\, C \rightarrow D.yx) & : & (C \rightarrow D) \rightarrow D \\
(\lambda x{:}\, C.\lambda y{:}\, C \rightarrow D.yx) & : & C \rightarrow (C \rightarrow D) \rightarrow D \\
(\lambda D{:}\, *.\lambda x{:}\, C.\lambda y{:}\, C \rightarrow D.yx) & : & \Pi D{:}\, *.C \rightarrow (C \rightarrow D) \rightarrow D \\
(\lambda C{:}\, *.\lambda D{:}\, *.\lambda x{:}\, C.\lambda y{:}\, C \rightarrow D.yx) & : & \Pi C{:}\, *.\Pi D{:}\, *.C \rightarrow (C \rightarrow D) \rightarrow D
\end{array}$$

```
Coq < Check f.
```

Faça agora

```
Coq < Variable P : Set.
Coq < Check P->P.
Coq < Compute (f T (P->P) t).
```

Vamos indicar uma cadeia de redução maximal para $(f\, T\, (P \rightarrow P)\, t)$

$$(f\, T\, (P \rightarrow P)\, t) \equiv (\lambda C{:}\, *.\lambda D{:}\, *.\lambda x{:}\, C.\lambda y{:}\, C \rightarrow D.yx)\, T\, (P \rightarrow P)\, t \rightarrow$$

$$\rightarrow (\lambda D{:}\, *.\lambda x{:}\, T.\lambda y{:}\, T \rightarrow D.yx)\, (P \rightarrow P)\, t \rightarrow (\lambda x{:}\, T.\lambda y{:}\, T \rightarrow P \rightarrow P.yx)\, t \rightarrow (\lambda y{:}\, T \rightarrow P \rightarrow P.yt)$$

Exercício: Avalie a seguinte sequência de comandos:

```
Coq < Section aula.
Coq < Variables A, B, C : Set.
Coq < Definition S := [x:A->B->C][y:A->B][a:A](x a (y a)).
Coq < Print S.
Coq < End aula.
Coq < Print S.
```

Comente o efeito e a resposta do sistema Coq à sequência de comandos anterior.

Exercício: Considere as seguintes definições Coq:

```
Coq < Definition exp1 := [x:A][y:B][C:Set][p:A->B->C] (p x y).
Coq < Definition exp2 := [A:Set][x:A][P:A->Set][y:(P x)] y.
Coq < Definition exp3 := [p:A->(C:Set)C][C:Set][a:A] (p a C).
```

Indique quais os g-termos que estão a ser definidos. Deduza o tipo de cada uma das expressões atrás definidas. Use o Coq para verificar a correcção das suas respostas.

Declare agora

```
Coq < Variable a :A.
Coq < Variable b :B.
Coq < Variable T : Set.
Coq < Variable t : T.
```

e analise o resultado dos comandos que se seguem fazendo, no papel, as respectivas cadeias de redução maximais.

```
Coq < Compute (exp1 a b A ([x:A][y:B] x) ).
Coq < Compute (exp2 T t).
```

Qual é tipos das expressões acima descritas ?

Exercício: Use o sistema Coq para o auxilar na resolução do seguinte problema.
Considere as seguites definições:

$$N \doteq \Pi C : *.C \to (C \to C) \to C$$

$$Z \doteq \lambda A : *.\lambda z : A.\lambda s : A \to A.z$$

$$S \doteq \lambda a : N.\lambda A : *.\lambda z : A.\lambda s : A \to A.s(aAzs)$$

$$soma \doteq \lambda b : N.\lambda a : N.\lambda A : *.\lambda z : A.\lambda s : A \to A.(bA)(aAzs)s$$

Deduza o seu tipo, e reduza à sua forma normal as expressões que se seguem:

i) SZ

ii) $S(SZ)$

iii) $soma\,(SZ)(S(SZ))$

Questão 1A. Considere a seguinte sequência de comandos:

```
Coq < Section exercicio.
Coq < Variable A, B : Set.
Coq < Variable a : A.
Coq < Variable b : B.
Coq < Definition EXP := (([x:A->B][y:A](x y))([y:A] b)(([y:A] y) a)).
```

i) Calcule a forma normal de `EXP`. Use o Coq para conferir a forma por si encontrada.

Faça agora

```
Coq < End exercicio.
```

ii) Comente o efeito do fecho da secção. Argumente com base nos noções teóricas estudadas.

Questão 1B. Considere a seguinte sequência de comandos:

```
Coq < Section exemplo.
Coq < Variable A : Set.
Coq < Definition B := (C:Set)C->C->C.
Coq < Definition T := [x:A][y:A] x.
Coq < Variable a, b : B.
Coq < Definition imp := [A:Set][x:A][y:A](a A (b A x y) x).
Coq < Print T.
Coq < Print imp.
Coq < End exemplo.
```

i) Comente o efeito do fecho da secção. Argumente com base nas noções teóricas estudadas.

ii) Calcule agora a forma normal do g-termo `(imp T T)`. Use o Coq para conferir a forma por si encontrada.

3.2 Alguns Vértices do λ-Cubo

Utilizaremos agora o sistema Coq para efectuar a manipulação de termos do λ-calculus com tipos, com a presença das várias dependências possíveis: termos dependentes de tipos (introduzidos em $\lambda 2$), tipos dependentes de tipos (introduzidos em $\lambda\underline{\omega}$), e tipos dependentes de termos (introduzidos em λP). O sistema onde todas estas dependências são possíveis tem o nome de **Cálculo das Construções**, ou λC.

3.2.1 Quantificação impredicativa em $\lambda 2/F$ – termos polimórficos (dependentes de tipos)

A dependência $\langle \Box, * \rangle$ num Sistema Abstracto de Tipos com a hierarquia linear de universos $* \in \Box$ permite a presença de abstracções que operam uniformemente sobre termos de todos os tipos. Comecemos por definir a função identidade que recebe um tipo (proposicional) seguido de um elemento desse tipo, que depois devolve:

```
Coq < Section lambda2.
Coq < Definition id := [alpha:Set][x:alpha] x.
Coq < Check id.
```

Note-se que:

(i) `id` tem tipo proposicional.

(ii) A aplicação de `id` a um tipo arbitrário resulta na função-identidade desse tipo.

De facto, a execução dos seguintes comandos permite confirmar estas observações:

```
Coq < Check (alpha:Set)alpha->alpha.
Coq < Variable D:Set.
Coq < Compute (id D).
Coq < Variable d:D.
Coq < Compute (id D d).
```

Questão 3A. Considere a seguinte função do λ-calculus com tipos simples.

$$fun \doteq \lambda g\colon A \to B \to C.\lambda h\colon (B \to C) \to A \to B.\lambda x\colon A.h(gx)x.$$

Defina em Coq a versão polimórfica desta função. Verifique o seu tipo e teste-a.

Que dependências são necessárias para garantir a boa formação do termo que acabou de definir. Justifique.

Questão 3B. Considere a seguinte função do λ-calculus com tipos simples que faz a composição de funções.

$$comp \doteq \lambda f\colon B \to C.\lambda g\colon A \to B.\lambda x\colon A.f(gx)$$

Defina em Coq a versão polimórfica desta função. Verifique o seu tipo e teste-a.

Que dependências são necessárias para garantir a boa formação do termo que acabou de definir. Justifique.

Exercício: Interprete a seguinte sequência de comandos. Diga se o termo `strange` é um termo de $\lambda 2$ antes e depois de ser fechado, e porquê.

```
Coq < Section l21.
Coq < Variable beta:Set.
Coq < Definition strange := [a: (alpha:Set)alpha] (a beta).
Coq < Check strange.
Coq < End l21.
Coq < Check strange.
Coq < Print strange.
```

3.2.2 Tipos dependentes de tipos em $\boldsymbol{\lambda\underline{\omega}}$

A dependência $\langle\Box,\Box\rangle$ permite extender o conjunto de *géneros* dos SATs (gerados pela hierarquia $* \in \Box$) pela seguinte instância da regra do produto:

$$\frac{\Gamma \vdash A : \Box \quad \Gamma \vdash B : \Box}{\Gamma \vdash A \to B : \Box}$$

Juntamente com o axioma $\Gamma \vdash * : \Box$, esta regra permite a existência de géneros como $* \to *$ ou $(* \to *) \to (* \to * \to *)$. comecemos por ver como por exemplo o combinador **K** pode ser escrito como operando sobre tipos proposicionais:

```
Coq < End lambda2.
Coq < Section lambdaWweak.
Coq < Check [A:Set][B:Set]A.
Coq < Check Set->Set->Set.
```

(Note-se que `Type` corresponde, em **Coq**, ao universo $\Box$.)

Estudemos em seguida um pequeno exemplo: consideremos o construtor de tipos `Par`, que a partir de dois tipos A e B constroi o tipo dos pares ordenados com a primeira componente em A e a segunda em B:

```
Coq < Variable Par : Set -> Set -> Set.
Coq < Variable A, B : Set.
```

`Par` é um g-termo classificado pelo género $* \to * \to *$, pelo que não é um tipo proposicional, nem em geral um tipo concreto.

Declaremos agora termos representando o constructor essencial dos pares (que agrega dois elementos num par), bem como os seus dois destructores (que desagregam um par nas suas duas componentes):

```
Coq < Parameter mkPair : A->B->(Par A B).
Coq < Parameter proj1 : (Par A B) -> A.
Coq < Parameter proj2 : (Par A B) -> B.
Coq < Check ( [x:(Par A B)] [y:B] (mkPair (proj1 x) y) ).
```

3.2.3 O sistema $\lambda\omega$

Se pretendermos agora declarar termos equivalentes aos constructores e destructores acima declarados, mas que sejam polimórficos, ou seja, capazes de lidar com pares de elementos de quaisquer tipos, poderemos fazer algo como:

```
Coq < Parameter MkPair : (A,B:Set) A->B->(Par A B).
Coq < Check (A,B:Set) A->B->(Par A B).
Coq < Parameter Proj1 : (A,B:Set) (Par A B) -> A.
Coq < Parameter Proj2 : (A,B:Set) (Par A B) -> B.
```

Note-se que os tipos de tais termos são ainda proposicionais (ou seja de género $*$), mas só têm existência na presença das duas dependências $\langle \Box, * \rangle$ e $\langle \Box, \Box \rangle$. O sistema onde ambas estão presentes toma o nome de $\lambda\omega$.

Questão 2A. Recorrendo ao contexto de que dispõe neste momento, defina a função polimórfica que recebe um par com ambas as componentes do mesmo tipo e devolve um par em que a primeira componente é a primeira componente do par recebido e a segunda componente é o próprio par. (ex: $\langle 1, 2 \rangle \mapsto \langle 1, \langle 1, 2 \rangle \rangle$)

Qual o seu tipo?

Questão 2B. Recorrendo ao contexto de que dispõe neste momento, defina a função polimórfica que recebe um par com ambas as componentes do mesmo tipo e devolve um par em que a primeira componente é o próprio par recebido e a segunda componente é a segunda componete do par. (ex: $\langle 1, 2 \rangle \mapsto \langle \langle 1, 2 \rangle, 2 \rangle$)

Qual o seu tipo ?

A coexistência das duas dependências referidas tem efeitos mais complexos do que os que se observam nos termos acima referidos. De facto, veja-se qual o efeito de se fechar a secção actual:

```
Coq < End lambdaWweak.
Coq < Check mkPair.
```

O tipo do termo `mkPair` foi alterado para reflectir o fecho desse termo em relação a identificadores declarados localmente na secção em que trabalhávamos. Como efeito desse fecho, o termo foi tornado polimórfico no sentido sugerido em 3.2.3 (efeito da abstracção nos tipos `A` e `B`), mas foi feita também abstracção no próprio constructor de tipos `Par`.

Note-se que o tipo deste termo `mkPair` é construído pela regra do produto com a dependência $\langle\Box, *\rangle$, mas tendo agora em conta a nova forma possível para os géneros (g-termos classificados por $\Box$), introduzida pela dependência $\langle\Box, \Box\rangle$. O efeito é o de ser agora possível o polimorfismo não só nos tipos mas também nos seus constructores.

Resta fazer o seguinte comentário: o exemplo que utilizámos, do tipo estruturado polimórfico "Par Ordenado", tem interesse reduzido, dada a ausência de um mecanismo pelo qual asseguremos as equivalências semânticas próprias desse tipo, nomeadamente,

$$\begin{array}{rcl} mkPair\ (proj1\ p)\ (proj2\ p) & = & p, \\ proj1\ (mkPair\ x\ y) & = & x, \\ proj2\ (mkPair\ x\ y) & = & y. \end{array}$$

Veremos, quando estudarmos os tipos indutivos, como tal deficiência pode ser facilmente ultrapassada.

3.2.4 Quantificação Predicativa em λP – Tipos Dependentes de Termos

A última dependência que é possível introduzir na família de SATs que temos vindo a considerar é $\langle *, \Box\rangle$, responsável, quando acrescentada a $\lambda \rightarrow$, pelo aparecimento de géneros da forma $A_1 \rightarrow A_2 \ldots \rightarrow A_i \ldots \rightarrow *$, com A_i tipos proposicionais. A seguinte instância da regra do produto, juntamente com o axioma $\Gamma \vdash * : \Box$, gera todos estes géneros:

$$\frac{\Gamma \vdash A : * \quad \Gamma \vdash B : \Box}{\Gamma \vdash A \rightarrow B : \Box}$$

O sistema resultante toma o nome de λP.

```
Coq < Restore State Initial.
Coq < Section lP.
Coq < Variable A : Set.
Coq < Check A->Set.
Coq < Variable P : A->Set.
Coq < Variable a:A.
Coq < Check (P a).
Coq < Check ((P a) -> Set).
Coq < Check ((P a) -> A -> Set).
```

A um termo como P acima declarado chama-se normalmente, em Teoria de Tipos, um predicado sobre o tipo A.[4]

A regra de formação de tipos-produto, que na sua instância de $\lambda_\rightarrow$ dava origem à formação dos tipos funcionais da forma $A \rightarrow B$, deve em λP ser devidamente reinterpretada. De facto, a razão pela qual em $\lambda_\rightarrow$ se podia escrever o tipo $\Pi x\colon A.B$ simplesmente como $A \rightarrow B$, prendia-se com a impossibilidade de o tipo B depender de alguma forma da variável x de tipo A. Ora, essa dependência torna-se possível em λP: basta observar, com $P : A \rightarrow *$, o tipo-produto $\Pi x\colon A.(Px)$.

[4]Em geral, um predicado será qualquer termo de tipo $A \rightarrow U$, com U um universo e $A : U$.

Estes tipos, cuja existência só em λP se torna possível, dizem-se formados por quantificação predicativa, já que se quantifica em variáveis que percorrem um tipo (A) que é classificado da mesma forma que o tipo quantificado (pelo universo $*$, neste caso).

```
Coq < Check ((a:A)(P a)).
```

Exercício: Será o g-termo $\Pi a\colon A.(Pa) \to *$ bem formado em λP (na secção corrente)? Utilize o **Coq** para confirmar a sua resposta.

Exercício: Escreva um termo cujo tipo seja $\Pi a\colon A.(Pa \to Pa)$, na mesma secção. Utilize o **Coq** para provar que o termo que apresentou habita realmente este tipo.

3.2.5 Cálculo de Construções (λC)

Acrescentando-se a dependência $\langle *, \Box \rangle$ ao sistema $\lambda\omega$ obtém-se o sistema λC.[5]

A coexistência das várias instâncias da regra do produto produz diversos efeitos. Antes de mais, a interacção entre as dependências $\langle *, \Box \rangle$ e $\langle \Box, \Box \rangle$ produz novas formas possíveis para os géneros:

```
Coq < Check ((A -> A -> Set) -> (A -> Set)).
Coq < Check ((P:A->Set)(a:A) (P a) -> Set).
```

Exercício: Apresente g-termos habitantes dos géneros anteriores e verifique os seus tipos em Coq.

Por outro lado, a instância da mesma regra gerada pela dependência $\langle \Box, * \rangle$ vai permitir agora a quantificação impredicativa em variáveis de todos os novos géneros. Observe-se os seguintes tipos proposicionais:

```
Coq < Print A.
Coq < Print P.
Coq < Check ((f:(A -> Set) -> (A -> Set)) (f P a)).
Coq < Check ((P:A->Set)(a:A) (P a) -> (P a)).
```

[5] Os outros sistemas do λ-cubo são os que resultam de se acrescentar esta dependência a $\lambda\underline{\omega}$, resultando em $\lambda P\underline{\omega}$, e a $\lambda 2$, resultando em $\lambda P2$. O sistema λC contém estes e todos os outros sistemas que temos vindo a discutir.

4 O Mecanismo de Prova – Lógicas Mínimas

Introduzimos aqui a utilização do sistema **Coq** para um dos fins essenciais a que se destina: a demonstração de teoremas. O princípio subjacente a esta utilização é a analogia **Proposições como Tipos**, que enunciamos aqui sem qualquer justificação. Seja $\vdash$ a relação de consequência definida numa lógica arbitrária. Dizemos que é válida a analogia proposições como tipos entre essa lógica e um determinado sistema de tipos se existir uma aplicação $[\bullet]$ que associa a cada frase da lógica um tipo proposicional do sistema em questão, e a cada prova de um teorema nessa lógica um termo do mesmo sistema, e se verificar o seguinte:

$$\vdash A \ sse \ \exists t.\ \Gamma_A \vdash t : [A]$$

sendo Γ_A o contexto mais pequeno do sistema de tipos necessário para declarar A como tipo proposicional bem formado.

Em diversos sistemas do λ-cubo (correspondentes a fragmentos da teoria de tipos do **Coq**) é válida a analogia acima apresentada, para uma lógica apropriada. Noutros sistemas, a relação *sse* deve ser substituída por uma implicação. Trata-se dos casos em que a representação referida é correcta mas não completa.

Ao longo deste documento apresentar-se-ão frases de várias lógicas diferentes, que serão apresentadas de forma perfeitamente informal. A demonstração de que uma frase é um teorema será feita pela construção (com sucesso) de um termo em **Coq** que codifique uma prova desse teorema. Naturalmente, o tipo do termo terá de ser o que codifica o teorema, pelo que o problema é equivalente ao de se encontrar, interactivamente, um termo de um determinado tipo.

Começamos por lidar com lógicas mínimas, ou seja, cujas únicas conectivas são a implicação e a quantificação universal. A representação de frases destas lógicas em **Coq** é trivial: a implicação é representada pelos tipos funcionais, e a quantificação pelos tipos-produto dependentes.[6]

4.1 Lógica Proposicional e Lógica Proposicional de Segunda Ordem

Provemos então um facto trivial. Seja A uma proposição. Provemos que $A \to A$ é um teorema procurando um termo `trivial` de tipo `A -> A`.[7]

```
Coq < Variable A : Prop.
Coq < Theorem trivial : A -> A.
```

Observe-se a representação do estado de prova efectuada pelo **Coq**: uma fracção com o objectivo da prova no denominador, e com o numerador vazio. Apliquemos a táctica **Intro** a este estado de prova:

```
trivial < Intro hip1.
```

[6] Por razões meramente pragmáticas, em **Coq** existem dois universos correspondentes ao primeiro universo $*$. Assim, além de `Set` que já conhecíamos, surge agora o universo `Prop`, que utilizamos na visão de proposições como tipos proposicionais.

[7] Vários outros comandos idênticos a `Theorem` podem ser utilizados, nomeadamente `Lemma`, `Fact`, e `Remark`.

O efeito desta aplicação pode ser visto a um nível lógico – a fórmula do lado esquerdo da implicação passará para a lista de hipóteses no contexto corrente, ou seja, para o numerador da fracção – ou ao nível da teoria de tipos – para encontrar um termo de tipo $U \to V$, basta encontrar um termo y de tipo V, no contexto em que se admite a existência de um termo x de tipo U. A regra da abstracção garantirá que o termo $\lambda x\!: U.y$ tem o tipo $U \to V$ pretendido.

O identificador `hip1` acima fornecido à táctica `Intro` será utilizado para identificar o termo que hipoteticamente tem tipo `A`. Se o houvéssemos omitido, o sistema teria utilizado um nome arbitrariamente escolhido.

O passo seguinte consiste em afirmar que dispomos, no contexto corrente, de um termo do tipo A que procuramos. Este termo é naturalmente, e trivialmente, `hip1`. Indiquemos este facto ao sistema usando a táctica `Exact`:

```
trivial < Exact hip1.
trivial < Save.
Coq < Print Proof trivial.
```

O comando `Save` guarda no contexto actual o termo que codifica a prova, e fecha a edição dessa prova. Também poderiamos ter usado o comando `Qed` para o mesmo efeito. Podemos visualizar o termo e o seu tipo com o comando `Print Proof`. O termo de prova no exemplo acima é naturalmente a função identidade do tipo A.

Iniciemos agora uma nova secção onde efectuaremos a próxima prova:

```
Coq < Section Hilbert1.
Coq < Variables A, B : Prop.
Coq < Theorem K : A->B->A.
```

Observe-se que quando se inicia a prova o sistema apresenta no denominador do estado de prova as declarações locais de `A` e `B`.

As duas aplicações sucessivas da táctica `Intro` podem ser efectuadas num único comando, com atribuição automática de nomes às hipóteses:

```
K < Intros.
```

Imediatamente se observa que podemos terminar a prova pois uma das hipóteses presentes no estado actual coincide com o objectivo! Mas desta feita, em vez de explicitarmos qual é esse termo, vamos deixar que o sistema o procure no contexto actual, usando uma nova táctica, `Assumption`:

```
K < Assumption.
K < Save.
Coq < Print Proof K.
```

Reconhecemos no termo de prova o combinador **K** do λ-calculus. Note-se que este termo depende dos dois tipos proposicionais `A` e `B`, declarados na secção local, pelo que se a fecharmos, o termo será quantificado nesses tipos:

```
Coq < End Hilbert1.
Coq < Print K.
```

Este termo, do tipo $\Pi A{:}*. \ \Pi B{:}*. \ A \rightarrow B \rightarrow A$, constitui uma prova da proposição $\forall A, B. \ A \rightarrow B \rightarrow A$, que é uma frase da lógica proposicional de segunda ordem: é quantificada universalmente nas proposições A e B. Como se procederia para efectuar directamente a prova de uma tal frase, sem recurso ao mecanismo de secções? Vejamos o segundo axioma de Hilbert:

```
Coq < Section Hilbert2.
Coq < Theorem S : (A,B,C:Prop) (A->B->C) -> (A->B) -> (A->C).
S < Intro.
S < Intro.
S < Intro.
```

O efeito das três aplicações da táctica `Intro` não devem causar espanto: de facto os tipos funcionais (implicações lógicas) não são mais do que casos particulares de tipos-produto (quantificações universais), pelo que o efeito da aplicação daquela táctica é o mesmo: criação de hipóteses e alteração do objectivo da prova. Para provarmos o teorema quantificado acima, basta provar o mesmo teorema sem qualquer quantificação, no contexto extendido com as 3 hipóteses introduzidas.

Em seguida efectuamos mais três introduções "funcionais":

```
S < Intros H H0 H1.
```

O passo seguinte envolve a utilização de uma nova táctica. Não existe no contexto qualquer prova de C, o objectivo actual. No entanto existe uma prova de $A \rightarrow B \rightarrow C$, pelo que deve ser possível substituir o objectivo actual por dois novos: A e B. A táctica `Apply` utiliza-se nos casos em que a conclusão de uma qualquer hipótese coincide com o objectivo da prova:

```
S < Apply H.
```

Observe-se a presença de *dois objectivos de prova* no estado actual. Um deles, A, pode ser imediatamente eliminado por coincidir com uma hipótese, e o outro, B, merece nova utilização de `Apply`, com o termo de tipo $A \to B$:

```
S < Assumption.
S < Apply H0.
S < Assumption.
S < Save.
Coq < End Hilbert2.
```

Questão 4A. Construa, em Coq, uma prova do seguinte teorema de segunda ordem:

$$\forall A, B, C.(A \to B) \to (B \to C) \to (A \to C)$$

Questão 4B. Construa, em Coq, uma prova do seguinte teorema de segunda ordem:

$$\forall C, D, E.(C \to D) \to (D \to E) \to (C \to E)$$

Estudemos em seguida um outro exemplo, onde se vê como tratar definições, durante a contrução de provas. Antes de mais note-se que a conectiva unária de negação é definida a partir da conectiva 0-ária absurdo (`False`, em **Coq**):

```
Coq < Print not.
```

E tentemos então provar o teorema $(A \to B) \to (\neg B \to \neg A)$. Note-se a sintaxe concreta `~` para o operador `not`:

```
Coq < Section s1.
Coq < Variables A, B : Prop.
Coq < Theorem t1 : (A -> B) -> (~B -> ~A).
t1 < Intros H H0.
```

Os passos que se seguem nesta prova devem consistir na aplicação da definição da negação, no objectivo da prova e numa hipótese, respectivamente:

```
t1 < Unfold not.
t1 < Unfold not in H0.
```

O primeiro dos comandos acima poderia ter sido substituído pelo comando `Red`, que não necessita de argumento (qual a definição a aplicar), uma vez que reescreve sempre o objectivo de prova aplicando a definição mais exterior nele presente. A prova continua trivialmente:

```
t1 < Intro H1.
t1 < Apply H0; Apply H; Exact H1.
t1 < Save.
Coq < End s1.
Coq < Print Proof t1.
```

A sequenciação de tácticas pode ser facilmente realizada pelo operador `;`.

Exercício: Construa uma prova de

$$\forall A.\ A \to \neg\neg A$$

4.2 Lógica de Predicados de Primeira e Segunda Ordem

A representação de frases da Lógica de Predicados de Primeira Ordem quantificadas universalmente obriga à introdução de quantificação predicativa (pela dependência $\langle *, \Box \rangle$) nos tipos proposicionais. Devem ser declarados tipos proposicionais para todos os domínios de discurso, e os predicados serão termos de géneros apropriados (com existência em λP). Nos exemplos que se seguem consideraremos uma assinatura homogénea (um único domínio de discurso que designaremos por D), e dois predicados P, unário, e Q, binário:

```
Coq < Section Pred.
Coq < Variable D:Set.
Coq < Variable P : D -> Prop.
Coq < Variable Q : D -> D -> Prop.
```

Provemos agora o teorema $\forall y.\ (\forall x.\ P(x)) \to P(y)$.

```
Coq < Section Pred1.
Coq < Theorem pred1 : (y:D) ((x:D)(P x)) -> (P y).
pred1 < Intros y H.
pred1 < Apply H.
pred1 < Save.
Coq < End Pred1.
Coq < Print Proof pred1.
```

Outro exemplo trivial: provemos o teorema $(\forall x, y.\ Q(x, y)) \to (\forall x, y.\ Q(y, x))$.

```
Coq < Section Pred2.
Coq < Theorem pred2: ((x,y:D)(Q x y)) -> (x,y:D)(Q y x).
pred2 < Intros H x y; Apply H.
pred2 < Save.
Coq < End Pred2.
```

Já não deve ser surpreendente o efeito do fecho da secção `Pred`, onde estão declarados `D`, `P`, e `Q`: os teoremas que provámos passam a estar quantificados nos predicados e no próprio domínio de discurso! Estamos pois perante teoremas da Lógica de Predicados de Segunda Ordem.

```
Coq < End Pred.
```

Questão 5A. Prove o seguinte teorema da Lógica de Predicados de segunda ordem:

$$\forall A, P.(A \to \forall x.P(x)) \to (\forall x.A \to P(x))$$

Questão 5B. Prove o seguinte teorema da Lógica de Predicados de segunda ordem:

$$\forall A, P.(\forall x.A \to P(x)) \to A \to \forall x.P(x)$$

4.3 Lógica Proposicional: As conectivas que faltam I

As conectivas de implicação e quantificação universal de proposições estão, como já vimos, em correspondência directa com construções da teoria de tipos do **Coq**. As restantes conectivas (absurdo, conjunção, e disjunção[8]) podem ser definidas da seguinte forma:

$$\begin{array}{rcl} \bot & \doteq & \forall A.\ A \\ P \wedge Q & \doteq & \forall A.\ (P \to Q \to A) \to A \\ P \vee Q & \doteq & \forall A.\ (P \to A) \to (Q \to A) \to A \end{array}$$

[8]Vimos já como se define a negação a partir do absurdo.

Observe-se que se trata de definições de segunda ordem, que apenas são realizáveis em **Coq** porque, como já vimos, é possível quantificar sobre proposições na lógica que está em correspondência com a sua teoria de tipos.[9]

Sem qualquer justificação teórica das razões pelas quais as definições acima são válidas, podemos simplesmente provar essa validade em **Coq**. Assim, é um facto que uma daquelas definições será válida se e só se a partir dela se puderem inferir as regras de introdução e eliminação da respectiva conectiva no estilo de Dedução Natural, e vice-versa.

Tomemos como exemplo a conjunção, e comecemos pela sua definição:

```
Coq < Section E.
Coq < Definition e := [P,Q:Prop] (A:Prop) (P->Q->A) -> A.
Coq < Check e.
```

Observe-se que, como não poderia deixar de ser, o operador `e` tem tipo $* \to * \to *$, uma vez que se trata de um constructor de tipos proposicionais a partir de outros dois tipos proposicionais. Note-se pois, que apesar de se efectuar uma codificação da conectiva de conjunção como um termo em que se faz apenas quantificação impredicativa numa proposição (possível com $\langle \Box, * \rangle$), o termo `e` em si tem um género cuja existência só é possível com $\langle \Box, \Box \rangle$. O termo existe pois em $\lambda\omega$.

Provemos antes de mais uma das regras de eliminação da conjunção (a outra é perfeitamente simétrica):

```
Coq < Theorem conj_e1 : (U,V:Prop) (e U V) -> U.
conj_e1 < Intros U V H.
conj_e1 < Unfold e in H.
conj_e1 < Apply H.
```

Observe-se com atenção o efeito desta última táctica, em que o parâmetro da hipótese `H` é instanciado com o objectivo `U`. O resto da prova é trivial.

```
conj_e1 < Intros; Assumption.
conj_e1 < Save.
```

Exercício: Prove a regra de introdução da conjunção:

$$\frac{A \quad B}{A \wedge B}$$

Em seguida, pretendemos provar o oposto: se as regras de dedução natural forem válidas, a definição de ordem superior pode ser derivada como teorema. Para isso utilizaremos uma técnica radicalmente diferente da anterior: não nos interessa *definir* a conectiva de disjunção, mas antes

[9] Note-se que esta *não é* a forma como trataremos no futuro as conectivas acima referidas, mas a sua definição constitui neste ponto um excelente exercício.

declarar um termo de tipo apropriado para ela, e depois declarar três termos de tipos correspondentes aos das regras de dedução natural da conectiva. Estes termos representam provas, assumidas, de que aquelas regras são válidas.[10]

```
Coq < Variable E : Prop -> Prop -> Prop.
Coq < Hypothesis Iconj : (A,B:Prop) A -> B -> (E A B).
Coq < Hypothesis Econj1 : (A,B:Prop) (E A B) -> A.
Coq < Hypothesis Econj2 : (A,B:Prop) (E A B) -> B.
```

O comando `Hypothesis` é perfeitamente equivalente a `Variable`, sendo normalmente preferido quando se interpreta tipos como proposições. Da mesma forma, para declarações globais, existe o comando `Axiom`.

Provemos então, que para duas proposições arbitrárias P e Q, se a sua conjunção é um teorema então a frase $\forall A.\ (P \to Q \to A) \to A$ também o é:

```
Coq < Theorem def_conj : (P,Q:Prop) (E P Q) -> (e P Q).
def_conj < Red.
def_conj < Intros P Q H A0 H0.
def_conj < Apply H0.
def_conj < Apply Econj1.
```

O erro aqui obtido necessita de explanação: se olharmos para o termo `Econj1` como uma regra lógica, vemos que ela se encontra parametrizada nas proposições nela intervenientes, `A` e `B`. A sua aplicação em retrocesso, via `Apply`, instancia sem problemas `A` com `P`, o objectivo actual da prova, mas não existe qualquer pista quanto à instanciação a efectuar de `B` (a regra não verifica a propriedade de sub-fórmula). Assim sendo, o sistema necessita de ajuda, quanto a essa instanciação. Queremos que `B` seja instanciado com `Q`:

```
def_conj < Apply Econj1 with Q.
def_conj < Assumption.
def_conj < Apply Econj2 with P.
def_conj < Assumption.
def_conj < Save.
Coq < End E.
Coq < Check def_conj.
```

Observe-se como, depois de fechada a secção corrente, o teorema que realmente provámos está

[10] Esta forma de se trabalhar com novas conectivas insere-se naquilo a que se costuma chamar Princípio dos **Juízos como Tipos**, na sua variante de **Fórmulas como Tipos**: declaram-se termos de tipos apropriados para codificar todos os elementos a incorporar na lógica, nomeadamente conectivas e regras. Esta metodologia difere radicalmente da outra que vimos, em que cada nova conectiva é um termo fechado, definido na teoria de tipos do **Coq**, sem necessitar de quaisquer declarações adicionais.

parametrizado na conectiva binária, e em provas das suas regras de eliminação. A regra de introdução da conectiva não foi utilizada na prova, pelo que não foi feita parametrização numa sua prova.

Exercício: Reproduza todo o raciocínio acima efectuado para a conectiva de conjunção, agora para a disjunção (i.e, prove que as regras de dedução natural podem ser inferidas a partir da definição de segunda ordem, e que esta pode ser provada se se assumir a validade daquelas regras).

5 O Mecanismo de Prova – outros tópicos

5.1 As conectivas que faltam II

A forma como o **Coq** trata as conectivas habituais da Lógica, além das básicas *implicação* e *quantificação universal*, passa pela utilização de tipos indutivos. A sua utilização pode no entanto ser feita de forma *naïf*, antes do estudo detalhado daqueles tipos.

Quando no objectivo de uma prova ocorre uma fórmula composta por uma das conectivas $\wedge$, ou $\vee$, é possível utilizar tácticas específicas para tratar a sua introdução.[11] A aplicação destas tácticas é semelhante à das regras de introdução, em Dedução Natural: `Split` corresponde à regra de introdução da conjunção, e `Left` e `Right` às duas regras de introdução da disjunção.

Por outro lado, quando numa hipótese aberta no estado de prova corrente ocorre uma fórmula construída por uma das conectivas referidas, ou ainda por $\perp$, o mecanismo de prova a utilizar está intimamente ligado à utilização de tipos indutivos mas pode ser entendido relembrando as definições de segunda ordem das ditas conectivas:

$$\begin{array}{rcl} \perp & \doteq & \forall A.\ A \\ P \wedge Q & \doteq & \forall A.\ (P \to Q \to A) \to A \\ P \vee Q & \doteq & \forall A.\ (P \to A) \to (Q \to A) \to A \end{array}$$

A validade destas igualdades permite à táctica `Elim`:

- Dada uma hipótese $\perp$, provar trivialmente qualquer objectivo.
- Dada uma hipótese da forma $P \wedge Q$, substituir qualquer objectivo A por um outro $P \to Q \to A$.
- Dada uma hipótese da forma $P \vee Q$, substituir qualquer objectivo A por dois outros, $P \to A$ e $Q \to A$.

A táctica `Elim` é de facto mais poderosa do que isto. Ela permite, por exemplo, dada uma hipótese $R \to P \wedge Q$, substituir, por eliminação dessa hipótese, qualquer objectivo A por dois outros, R e $P \to Q \to A$.

Comecemos por uma prova extremamente simples que envolve a eliminação de uma hipótese negada:

```
Coq < Variables P, Q, R, S : Prop.
Coq < Theorem absurd: S /\ ~S  -> Q.
absurd < Intro H.
absurd < Elim H.
absurd < Intros H0 H1.
absurd < Unfold not in H1.
absurd < Elim H1.
```

O último comando efectuou eliminação da hipótese $S \to \perp$. De facto, uma vez que do absurdo se pode inferir a validade de todas as fórmulas, qualquer objectivo de prova pode ser substituído por $\perp$, e este por S, dada a presença da hipótese referida. De notar que não teria sido necessário

[11] Note-se que à conectiva $\perp$, que também não é primitiva em **Coq**, não está associada qualquer regra de introdução.

efectuar a reescrita de $\neg S$ como $S \rightarrow \bot$. Recuemos dois passos na prova:

```
absurd < Undo 2.
absurd < Elim H1.
absurd < Assumption.
absurd < Save.
```

Vejamos agora uma prova de um teorema envolvendo conjunção e disjunção (observe-se a sintaxe concreta `/\` e `\/` para os operadores `and` e `or` respectivamente):

```
Coq < Theorem conj_disj : P/\Q -> P\/Q.
conj_disj < Intro H.
conj_disj < Elim H.
conj_disj < Clear H.
```

Frequentemente, uma hipótese, depois de eliminada, não mais volta a ser usada, pelo que pode ser abandonada. É o que se passa no caso corrente. O comando `Clear` permite apagar uma hipótese do contexto actual.

```
conj_disj < Intros H H0.
conj_disj < Left.
conj_disj < Assumption.
conj_disj < Save.
Coq < Print Proof conj_disj.
```

Note-se que em vez da táctica `Left` se poderia ter usado `Right`, uma vez que ambos os "lados" da disjunção existem como hipóteses.

O próximo exemplo utiliza a táctica `Split`:

```
Coq < Theorem pc : (P -> Q) /\ (R -> S) -> (P /\ R -> Q /\ S).
pc < Intros H H0.
pc < Elim H; Clear H; Elim H0; Clear H0.
pc < Intros H H0 H1 H2.
pc < Split.
pc < Apply H1; Assumption.
pc < Apply H2; Assumption.
pc < Save.
Coq < Check pc.
```

Questão 6A. Construa, em Coq, uma prova para o seguinte teorema:

$$P \wedge (Q \vee R) \to (P \wedge Q) \vee (P \wedge R)$$

Questão 6B. Construa, em Coq, uma prova para o seguinte teorema:

$$(P \wedge Q) \vee R \to (P \vee R) \wedge (Q \vee R)$$

O Quantificador Existencial

Também o quantificador existencial está definido com recurso a um tipo indutivo, sendo no entanto possível a sua utilização de forma simples.

O tipo do quantificador é aquele com que é costume codificar-se qualquer quantificador: $(A \to *) \to *$ para um determinado tipo A. De facto, ele deve construir uma fórmula, a partir de um predicado. Se for por exemplo $P : A \to *$, então será $(\exists P) : *$, uma frase lógica, assim como $(\exists(\lambda x\colon A.Px)) : *$. Esta segunda representação será a que utilizaremos, e a razão para isso compreende-se muito facilmente: imagine-se um predicado binário $R\colon A \to A \to *$. A quantificação existencial em qualquer das variáveis argumentos do predicado pode ser feita muito facilmente (no contexto $\{A : *\}$) como $\exists(\lambda x\colon A.\ Rxy)$ e $\exists(\lambda y\colon A.\ Rxy)$, respectivamente.

```
Coq < Check ex.
```

Em **Coq**, o operador `ex` tem o tipo acima, mas quantificado ainda no tipo A. A representação de $\exists x.\ Cx$, com $C\colon U \to *$, será feita como se mostra:

```
Coq < Variable U : Set.
Coq < Variable C : U -> Prop.
Coq < Check (ex U [x:U](C x)).
```

Existe no entanto a sintaxe concreta `Ex`, que dispensa o primeiro argumento:

```
Coq < Check (Ex [x:U](C x)).
```

`ex` pode ser eliminado, como as restantes conectivas, pela táctica `Elim`, e introduzido no objectivo de prova com a táctica especial `Exists`.

Provemos então o teorema $\exists x.\ (A(x) \to B(x)) \to \forall x.\ A(x) \to \exists x.\ B(x)$:

```
Coq < Variable D : Set.
Coq < Variables A, B : D -> Prop.
Coq < Goal (Ex [x:D]((A x) -> (B x))) ->
Coq <         ((x:D)(A x)) -> (Ex [x:D](B x)).
Unnamed_thm < Intros H H0.
Unnamed_thm < Elim H; Clear H.
Unnamed_thm < Intros y H.
```

O próximo passo é o mais importante: temos como hipóteses $\forall x.A(x)$ e $A(y) \to B(y)$, estando a variável $y : D$ também presente no contexto actual. Então já podemos fornecer o *testemunho* do objectivo actual $\exists x.\ B(x)$: trata-se de y.

```
Unnamed_thm < Exists y
Unnamed_thm < Apply H.
Unnamed_thm < Apply H0.
Unnamed_thm < Abort.
```

Questão 7A. Construa, em Coq, uma prova para o seguinte teorema:

$$(\exists x.A(x) \vee B(x)) \to (\exists x.A(x)) \vee (\exists x.B(x))$$

Questão 7B. Construa, em Coq, uma prova para o seguinte teorema:

$$(\exists x.A(x) \wedge B(x)) \to (\exists x.A(x)) \wedge (\exists x.B(x))$$

Exercício: Averigue se a frase $\forall x.\exists y.P(x,y) \to \exists y.\forall x.P(x,y)$ é um teorema da Lógica de Primeira Ordem.

5.2 Comandos de Gestão da Prova

Temos vindo a utilizar alguns comandos indispensáveis no processo de construção interactiva de uma prova. Sistematizemos a função de cada um:

`Show` Este comando permite visualizar o estado de prova corrente, com todos os seus objectivos.

`Clear` Permite remover uma hipótese do contexto corrente. A hipótese não poderá ser usada no futuro.

`Restart` Permite o regresso ao estado de prova inicial, ou seja, anula o efeito de todos os passos de prova efectuados.

`Abort` Utilizado sem argumento, este comando permite cancelar a construção de prova em que se trabalha actualmente. Quando se edita várias provas em simultâneo, recebe um argumento que descrimina qual a que se pretende abortar.

`Undo` Cancela o último passo de prova efectuado. Com um argumento numérico `n`, repete `n` vezes o seu efeito.

`Suspend / Resume` O primeiro comando tem como efeito abandonar temporariamente a edição da prova corrente, regressando-se ao *toplevel* do **Coq**, e o segundo permite retomar aquela edição. Sem argumento permite regressar à última prova em que se trabalhou; o argumento opcional permite especificar qual a prova a que se deseja regressar.

`Focus / Unfocus` O primeiro comando tem como efeito focar a prova no primeiro sub-objectivo a provar; os restantes sub-objectivos não são apresentados no ecran (embora existam). O segundo comando desfaz o efeito de `Focus`.

5.3 Automatização da construção de provas, Lógica Clássica, Táctica de *corte*, e Combinadores de tácticas

Um sistema dedutivo para a Lógica Clássica pode ser facilmente obtido de um sistema dedutivo Intuicionista,[12] bastando para isso adicionar a este um axioma clássico, como seja $\neg\neg A \to A$, que não é mais do que outra forma de escrever o chamado *princípio da redução ao absurdo*, $(\neg A \to \bot) \to A$. Outro axioma equivalente será ainda o *princípio do terceiro excluído*, $A \vee \neg A$.

Comecemos por iniciar uma nova secção onde se considera válido o princípio de redução ao absurdo. Provemos que o princípio do terceiro excluído é um teorema:

```
Coq < Section Classical.
Coq < Hypothesis raa: (A:Prop) ~~A -> A.
Coq < Theorem tnd : (A:Prop) A \/ ~A.
tnd < Intro A0.
tnd < Apply raa.
tnd < Red.
tnd < Intro.
tnd < Elim H.
tnd < Left.
tnd < Apply raa.
tnd < Red.
tnd < Intro.
tnd < Elim H.
tnd < Right.
tnd < Assumption.
```

Dada a dificuldade de construção desta prova em retrocesso, devida à necessidade de se "adivinhar" quando aplicar a regra RAA, e como obter o absurdo, justifica-se a apresentação da árvore de prova em Dedução Natural (*mp* e $\to$ são as regras de eliminação e introdução da implicação, e não se faz

[12]Note-se que a analogia proposições como tipos coloca qualquer subsistema do Cálculo de Construções em correspondência com uma determinada lógica construtiva.

referência explícita às aplicações da definição da negação):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\neg A]}{A \vee \neg A}\,Right \qquad [\neg(A \vee \neg A)]}{\bot}\,mp}{A}\,raa}{A \vee \neg A}\,Left \qquad [\neg(A \vee \neg A)]}{\bot}\,mp}{A \vee \neg A}\,raa$$

Observe-se agora a seguinte árvore de prova alternativa:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A]}{A \vee \neg A}\,Left \qquad [\neg(A \vee \neg A)]}{\bot}\,mp}{\neg A}\,\rightarrow \qquad \dfrac{\dfrac{\dfrac{[\neg A]}{A \vee \neg A}\,Right \qquad [\neg(A \vee \neg A)]}{\bot}\,mp}{\neg\neg A}\,\rightarrow}{\bot}\,mp}{A \vee \neg A}\,raa$$

Como reproduzir este raciocínio alternativo em **Coq**?

```
tnd < Restart.
tnd < Intro A0.
tnd < Apply raa.
tnd < Red.
tnd < Intro.
```

Esta parte inicial da prova é idêntica. Mas segue-se um passo não trivial: O objectivo $\bot$ deve agora ser substituído pelos dois objectivos $\neg A$ e $\neg A \to \bot$. A táctica `Cut` permite em geral substituir o objectivo V por $U \to V$ e U, mediante o comando `Cut U`:[13]

```
tnd < Cut ~A0.
tnd < Intro H0.
tnd < Elim H.
tnd < Right; Assumption.
tnd < Red.
tnd < Intro H0.
tnd < Elim H.
tnd < Left; Assumption.
```

Não vamos ainda abandonar esta prova. Antes disso, utilizemo-la para demonstrar a utilização da táctica `Auto` de automatização do processo de construção de prova. Esta táctica sistematiza algumas sequências correntes, como sejam aplicações de `Intro` e `Assumption`, e ainda `Apply` de algumas hipóteses, nomeadamente as de introdução das conectivas, como `Left` ou `Split`. A táctica

[13] Uma outra táctica útil é `Absurd`: O comando `Absurd U` substitui qualquer objectivo de prova pelos dois objectivos U e $\neg U$.

não faz qualquer aplicação de `Elim`, nem tenta adivinhar testemunhos para a táctica `Exists`.

```
tnd < Restart.
tnd < Intro A0.
tnd < Apply raa.
tnd < Red.
tnd < Intro.
tnd < Cut ~A0.
tnd < Auto.
tnd < Red.
tnd < Auto.
```

A prova acima pode ser resolvida com uma única táctica, construída com os combinadores `;` e `[ ... | ... | ... ]`. O primeiro permite sequenciar aplicações de tácticas, enquanto o segundo é útil na aplicação de tácticas que gerem múltiplos objectivos de prova. Assim, por exemplo, a táctica composta `t1; [t21 | t22 | t23]` consiste na aplicação de `t21` ao primeiro objectivo gerado por `t1`, `t22` ao segundo, e `t23` ao terceiro. A táctica composta `t1; t2` consiste na aplicação da táctica `t1` ao objectivo corrente e depois a táctica `t2` a todos os sub-objectivos gerados por `t1`.

```
tnd < Restart.
tnd < Intro A0; Apply raa; Red; Intro;
tnd <     Cut ~A0; [Auto | Red; Auto].
```

A prova pode ainda ser um pouco mais automatizada do que se mostrou acima. De facto, se se começar por remover todas as ocorrências da negação, a táctica `auto` consegue atacar a prova mais cedo:

```
tnd < Restart.
tnd < Unfold not; Unfold not in raa; Intro; Apply raa; Auto.
tnd < Save.
Coq < Check tnd.
```

Provamos pois que o princípio do terceiro excluído é derivado na Lógica Clássica, mas observe-se agora o efeito do seguinte:

```
Coq < End Classical.
Coq < Check tnd.
```

Ao fechar a secção onde se assumiu como válido o princípio da redução ao absurdo, o sistema abstraiu o termo `tnd` numa prova daquele princípio, como seria de esperar. O teorema provado por este termo é intuicionista! Este efeito poderia ter sido evitado se se tivesse declarado globalmente `raa` com `Axiom` em vez de localmente com `Hypothesis`.

Efectuemos agora a prova em sentido contrário: do princípio do terceiro excluído podemos inferir a redução ao absurdo:

```
Coq < Section Classical2.
Coq < Hypothesis te : (X:Prop) X \/ ~X.
Coq < Theorem raa : (Y:Prop) ~~Y -> Y.
raa < Intros Y H.
raa < Elim te.
```

Obtivemos uma mensagem de erro ao tentar eliminar a disjunção presente em `te`. De facto, a táctica `Elim` necessita de conhecer a instância específica daquela hipótese que se pretende eliminar:

```
raa < Elim te with Y.
```

Surgem dois objectivos, dos quais o primeiro pode ser imediatamente provado, mas o segundo necessita de alguma ajuda da nossa parte:

```
raa < Auto.
raa < Intro H0.
raa < Elim H.
raa < Assumption.
```

Ou numa só táctica:

```
raa < Restart.
raa < Intros Y H; Elim te with Y;
raa <     [Auto | Intro H0; Elim H; Assumption].
raa < Save.
```

Vejamos finalmente um exemplo de uma prova clássica em Lógica de Predicados. Efectuaremos a prova na secção corrente, tendo como hipótese o princípio do terceiro excluído. O teorema que desejamos provar é $(\forall x.\neg A(x)) \vee (\exists x.A(x))$.

```
Coq < Theorem dq : ((x:D) ~(A x)) \/ (Ex [x:D] (A x)).
```

Esta construção de prova seguirá o padrão típico de "análise de casos", cuja utilização só é possível com o princípio do terceiro excluído. Começamos por eliminar a instância deste axioma relativa à validade de $\exists x.A(x)$ ou da sua negação:

```
dq < Elim (te (Ex [x:D] (A x))).
```

O sistema gera dois objectivos de prova. O primeiro pode ser facilmente atingido, uma vez que a hipótese introduzida coincide com o lado direito da disjunção. A táctica `auto` consegue tratar este primeiro objectivo. O segundo objectivo terá de ser atingido via o lado esquerdo da disjunção: provaremos para um x arbitrário $\neg A(x)$, a partir de $\neg\exists x.A(x)$:

```
dq < Auto.
dq < Intro H.
dq < Left.
dq < Intro x.
```

Os passos seguintes consistem, depois da reescrita do objectivo da prova (que tem a forma de uma negação) na eliminação da negação acima representada pela hipótese `H`, e na nomeação do testemunho que verifica $\exists x.A(x)$:

```
dq < Red; Intro.
dq < Elim H.
dq < Exists x.
dq < Exact H0.
```

Reproduzimos a prova usando uma táctica composta:

```
dq < Restart.
dq < Elim (te (Ex [x:D] (A x)));
dq <   [Auto | Intro H; Left; Intro x; Red; Intro; Elim H;
dq <           Exists x; Exact H0].
dq < Save.
Coq < End Classical2.
```

Exercício: Construa provas dos seguintes teoremas clássicos, usando (alternadamente) os axiomas de redução ao absurdo e do terceiro excluído.

1. $(A \to B) \to (\neg A \vee B)$.
2. $\neg\exists x.\neg A(x) \to \forall x.A(x)$.

5.4 Tácticais

Tácticais são operadores sobre tácticas que dão a possibilidade de descrever estratégias de prova de maneira mais sintética.

Para além dos operadores ; e [...|...|...] o Coq disponibiliza mais alguns tácticais. Passemos a apresentar, em resumo, os tácticais existentes:

`Idtac` Esta táctica não faz nada (o que pode ser util, como veremos).

`Do` *num tac* Repete *num* vezes a aplicação da tática *tac*.

tac_1 `Orelse` tac_2 Tenta aplicar a táctica tac_1 e caso esta falhe aplica tac_2.

`Repeat` *tac* Repete a aplicação da táctica *tac*, enquanto ela não falha.

tac_1 ; tac_2 Sequêncializa da aplicação de tácticas. Aplica tac_1 e em seguida aplica tac_2 a todos os objectivos erados pela táctica tac_1.

tac ; [$tac_1|...|tac_n$] Paraleliza a aplicação de tácticas. Aplica tac_i ao i-ezimo sub-objectivo gerado por *tac*.

`Try` *tac* Tenta aplicar a táctica *tac*, se não conseguir não dá erro. Portanto nunca falha.

Assim por exemplo:

- `Do 2 Intro.` é equivalente a `Intro. Intro.`
- `Intros.` tem o mesmo efeito de `Repeat Intro.`
- `Try` *tac*. é equivalente a *tac* `Orelse Idtac.`

6 Introdução aos Tipos Indutivos – os números naturais

Seguidamente passamos a estudar os **tipos indutivos** em **Coq**. Revestindo-se estes de uma importância primordial em teoria de tipos, por permitirem a definição de estruturas matemáticas concretas e o raciocínio sobre estas, eles assumem em **Coq** uma importância ainda maior. De facto, toda a concepção do sistema se encontra intimamente ligada aos tipos indutivos. Basta observar que, como veremos, as próprias conectivas lógicas são definidas às custas de tipos indutivos.

Aproveitamos também para tecer algumas considerações sobre a definição da **igualdade** em sistemas de tipos e no **Coq** em particular.

6.1 Motivação – Os números naturais e os axiomas de Peano

Começamos por definir em **Coq** um tipo 'estruturado' para os números naturais, e por efectuar algumas provas básicas envolvendo elementos deste tipo.

A axiomatização do comportamento destes números passa pela definição de dois construtores, *zero*, e *sucessor*, pelo que iniciamos o nosso trabalho como se segue:

```
Coq < Section Natural.
Coq < Variable natural : Set.
Coq < Variable zero : natural.
Coq < Variable suc : natural -> natural.
Coq < Variables soma, prod : natural -> natural -> natural.
```

Definimos depois duas operações binárias sobre naturais, a soma e o produto. Pode-se ainda incluir as seguintes linhas:

```
Coq < Infix 8 "+" soma.
Coq < Infix 9 "*" prod.
```

cujo efeito será o de permitir a representação infixa da soma e do produto pelos símbolos $+$ e $*$, respectivamente. Note-se no entanto que o significado de `(suc 2 + 3)` será o de `(suc (soma 2 3))` e não o de `(soma (suc 2) 3)`.

Em seguida passamos à axiomática propriamente dita. Os dois primeiros axiomas de Peano afirmam, respectivamente, a ortogonalidade dos dois construtores (ou seja, *zero* não pode ser o sucessor de qualquer número, ou de outra forma, um natural não pode ser construído simultaneamente por mais do que um construtor), e a sua injectividade, que só se aplica ao *sucessor* por ser o único construtor funcional.

```
Coq < Hypothesis s_ne_z : (x:natural) ~(suc x) = zero.
Coq < Hypothesis suc_inj : (x,y:natural)
Coq <                      (suc x) = (suc y) -> x = y.
```

Note-se que esta última implicação em sentido inverso é verificada trivialmente, por ser uma das características definidoras de qualquer relação de igualdade a *congruência* em relação a todos os operadores. Teremos algo mais a dizer sobre isto.

Seguem-se os axiomas das operações de soma e produto:

```
Coq < Hypothesis Sum1 : (x:natural) (soma zero x) = x.
Coq < Hypothesis Sum2 : (x,y:natural)
Coq <                   (soma (suc y) x) = (suc (soma y x)).
Coq < Hypothesis Prod1 : (x:natural) (prod zero x) = zero.
Coq < Hypothesis Prod2 : (x,y:natural)
Coq <                 (prod (suc y) x) = (soma x (prod y x)).
```

O último axioma será o princípio de indução nos números naturais: "Se um predicado é válido em *zero* e ao assumir-se a sua validade em x se pode concluir da sua validade em $sucessor(x)$, então o predicado será universalmente válido." Note-se que se trata de um axioma de segunda ordem, quantificado no predicado:

```
Coq < Hypothesis induct : (P:natural->Prop)
Coq <      (P zero) -> ((x:natural) (P x) -> (P (suc x)))
Coq <       -> (x:natural) (P x).
```

6.2 Algumas notas sobre a *igualdade* em Coq

A existência de um predicado universal[14] de igualdade em **Coq** leva-nos a reflectir um pouco sobre quais devem ser as propriedades de tal predicado. Além de ter de se tratar obrigatoriamente de uma relação de equivalência (reflexiva, simétrica e transitiva), tem de verificar ainda duas propriedades:

1. a de substituição, que afirma que se dois termos t e t' são iguais, e σ é uma substituição das suas variáveis, então tem de ser ainda $\sigma(t) = \sigma(t')$. Esta propriedade é automaticamente verificada em ferramentas de prova, pela utilização da unificação.

2. a de congruência, em relação a todos os operadores definidos na assinatura de trabalho. Assim, no caso corrente, terão de ser válidos os seguintes teoremas:

$$\begin{array}{rcl} & x = y & \to & suc(x) = suc(y) \\ x = y & \to & x' = y' & \to & x + x' = y + y' \\ x = y & \to & x' = y' & \to & x * x' = y * y' \end{array}$$

Em **Coq**, em vez de se encontrarem axiomatizados todos estes princípios, a igualdade é definida por um axioma de ordem superior:

$$x = y \to \forall P.\ P(x) \to P(y)$$

[14]no sentido em que é válida para todos os `Sets`.

Este axioma, juntamente com o de reflexividade, permite derivar todos os outros, e leva a que a forma preferencial de se lidar com a igualdade em **Coq** seja por utilização de tácticas especiais de **reescrita**. Assim, se o objectivo actual de prova for $P(y)$ e existir uma hipótese $H : y = x$, podemos invocar a táctica `Rewrite` com argumento H para substituir aquele objectivo por $P(x)$. Se a hipótese fosse $H : x = y$, a táctica a utilizar deveria ser `Rewrite <-`, que permite que a equação seja utilizada da direita para a esquerda.

Como ilustração deste facto, demonstremos a propriedade de transitividade:

```
Coq < Theorem trans : (A:Set)(x,y,z:A)x=y->y=z->x=z.
trans < Intros A x y z H H0.
```

Podemos agora efectuar duas reescritas diferentes, correspondentes a cada uma das hipóteses (equações). Note-se que cada uma delas deve ser aplicada num sentido diferente:

```
trans < Rewrite H.
trans < Undo.
trans < Rewrite <- H0.
trans < Auto.
trans < Abort.
```

Os três axiomas definidores das propriedades da igualdade como relação de equivalência, apesar de poderem ser deduzidos desta forma, encontram-se também disponíveis através de três tácticas, `Reflexivity`, `Symmetry`, e `Transitivity`.

Veremos ainda, na próxima sessão de trabalho, que esta igualdade, conhecida por **Igualdade de Leibniz**, é definida em `Coq` com recurso a um tipo indutivo.

6.3 Algumas provas sobre os números naturais, utilizando reescrita

Comecemos por provar um facto simples: $1 + 1 = 2$:

```
Coq < Theorem dois :  (suc (suc zero)) = (soma (suc zero) (suc zero)).
dois < Rewrite Sum2; Rewrite Sum1; Reflexivity.
```

Concluímos a prova, no entanto poderíamos ter tomado partido da táctica `Auto` para efectuar alguns dos passos por nós. Quando lidamos com tipos definidos como `natural`, é normalmente muito útil declarar como `Hints` (*dicas*) teoremas relativos à congruência da igualdade em relação a cada um dos construtores funcionais do tipo. No caso corrente dispomos apenas do *sucessor*:

```
dois < Restart.
dois < Lemma SucCong : (x,y:natural) x=y -> (suc x) = (suc y).
SucCong < Intros x y H; Rewrite H; Reflexivity.
SucCong < Save.
dois < Hint SucCong.
```

Sempre que o objectivo actual for uma igualdade, a táctica `Auto` tentará agora efectuar `Apply SucCong`. Isto só será possível, naturalmente, se ambos os membros da igualdade forem construídos com `suc`. Para se visualizar as `Hints` disponíveis, basta fazer:

```
Coq < Print Hint.
```

Continuemos então a nossa prova:

```
dois < Rewrite Sum2; Auto.
dois < Save.
```

`Auto` conseguiu terminar a prova com sucesso porque aplicou o lema `SucCong` e depois `Sum1`. Note-se que `Sum1` não está declarado como `Hint`, mas pelo simples facto de se encontrar no contexto corrente (no numerador do estado de prova) a sua aplicação é considerada pela táctica `Auto`. Vejamos mais um exemplo:

```
Coq < Theorem um : (prod (suc zero) (suc zero)) = (suc zero).
um < Rewrite Prod2; Rewrite Sum2; Rewrite Prod1; Auto.
um < Save.
```

Em seguida efectuemos algumas provas por indução. Provemos, em primeiro lugar, que zero é o elemento neutro da soma. Este teorema é 'simétrico' do primeiro axioma definidor da soma. Tenha-se em atenção que para efectuar a unificação de ordem superior do axioma `induct` com o objectivo do estado de prova corrente, o sistema necessita de alguma ajuda da nossa parte:

```
Coq < Theorem Sum1r : (n:natural) (soma n zero) = n.
Sum1r < Intro n; Apply induct with x:=n;
Sum1r < [ Auto | Intros x H; Rewrite Sum2; Auto ].
Sum1r < Save.
Coq < Hint Sum1r.
```

Note-se que a declaração desta `Hint` permitirá a `Auto` lidar com objectivos de prova da forma `(soma x zero) = x`, mas não, por exemplo, com `(soma x zero) = t` em que `t` é qualquer termo equivalente a `x`. Para isto, `Auto` teria que conseguir fazer reescrita do lado esquerdo da igualdade por `Sum1r`, o que não acontece: `Auto` limita-se a fazer `Apply Sum1r`.

Provemos agora o simétrico do segundo axioma da soma:

```
Coq < Theorem Sum2r : (m,n:natural)
Coq <               (soma m (suc n)) = (suc (soma m n)).
Sum2r < Intros m n; Apply induct with x := m;
Sum2r < [ Rewrite Sum1; Auto
Sum2r < | Intros x H; Rewrite Sum2; Rewrite Sum2; Auto].
Sum2r < Save.
Coq < Hint Sum2r.
```

Questão 8A. Prove a comutatividade da soma.

Questão 8B. Prove o seguinte teorema

$$\forall x.(prod\ x\ zero) = zero$$

Por fim, um comentário que julgamos importante: poderia parecer que falta incluir na axiomática um axioma de *inicialidade*: os números naturais são todos os gerados pelos dois construtores *zero* e *sucessor*, e *apenas esses*, ou seja, todo o número natural é gerado por um dos construtores. De facto tal axioma não é necessário, uma vez que o princípio de indução permite derivar este facto, como se demonstra:

```
Coq < Theorem inicial : (n:natural)
Coq <                (n=zero) \/ (Ex [x:natural] n=(suc x)).
inicial < Intro n.
inicial < Apply induct with x:=n;
inicial <  [ Left; Auto
inicial <  | Intros x H; Elim H;
inicial <     [ Clear H; Intro H; Right; Exists zero; Auto
inicial <     | Clear H; Intro H; Right; Elim H; Clear H;
inicial <       Intros x0 H; Exists (suc x0); Auto
inicial <     ]
inicial <  ].
inicial < Save.
Coq < Reset Natural.
```

Observe-se que em vez de `End Natural`, usámos o comando `Reset`, cujo efeito é muito mais forte: remove do estado do sistema todos os identificadores definidos depois de (e incluindo) `Natural`.Uma acção tão radical é justificada pelo facto de todos os teoremas provados na secção anterior serem declarados globalmente, e uma vez que na próxima secção voltaremos a provar teoremas idênticos, é desejável que os respectivos nomes estejam 'limpos'.

6.4 Tipos Indutivos em Coq

A existência, em algumas linguagens de programação, de primitivas específicas para a definição de novos tipos indutivos generaliza e automatiza os princípios referidos relativamente aos números naturais. Assim, quando numa linguagem de programação se cria um novo tipo indutivo, associando-se-lhe um nome e um conjunto de construtores,[15] tem-se em geral as seguintes garantias:

1. Os construtores são ortogonais;
2. Os construtores são injectivos;

Também em **Coq** se terá estas garantias ao declarar um novo tipo indutivo, pelo que a declaração explícita de hipóteses correspondentes a estas propriedades, como efectuámos em cima com os números naturais, se torna desnecessária. Além disso, sendo o **Coq** um sistema orientado para a construção de provas, o princípio de indução de cada novo tipo indutivo é gerado automaticamente, pelo que é demonstrável, para cada tipo, que todo o habitante do tipo terá de ser gerado por um (e apenas um, dada a sua ortogonalidade) dos seus construtores.

Vejamos então como declarar o tipo indutivo dos números naturais:

```
Coq < Section Nati.
Coq < Inductive nati : Set := zero : nati
Coq <                       | suc  : nati -> nati.
```

Comecemos por provar imediatamente a congruência da igualdade em relação ao operador *sucessor*, para a podermos incluir na lista de *hints*:

```
Coq < Theorem SucCong : (x,y:nati) x=y -> (suc x) = (suc y).
SucCong < Intros x y H; Rewrite H; Auto.
SucCong < Save.
Coq < Hint SucCong.
```

E vejamos agora como efectuar uma prova indutiva. O **princípio de indução**, gerado automaticamente, tem sempre por nome o nome do tipo, seguido do sufixo `_ind`:

```
Coq < Check nati_ind.
```

Observe-se que ele corresponde de facto ao axioma que tínhamos incluído na nossa primeira teoria para os números naturais. Provemos o teorema de inicialidade:

[15] sendo cada um destes de um tipo funcional cujo codomínio seja o tipo indutivo que se define.

```
Coq < Theorem inicial : (n:nati)
Coq <             (n=zero) \/ (Ex [x:nati] n=(suc x)).
inicial < Intro n.
inicial < Apply nati_ind with n:=n.
```

De novo foi necessário ajudar na unificação de ordem superior. O **Coq** disponibiliza uma táctica própria que facilita um pouco esta tarefa: a indução pode ser invocada a partir da variável de tipo `nati` que se encontra declarada no contexto actual e sobre a qual se pretende efectuar o raciocínio indutivo. A dita táctica chama-se `Elim`[16]:

```
inicial < Undo.
inicial < Elim n.
```

Uma outra táctica, de utilização ainda mais útil porque faz por nós o passo inicial de introdução da variável quantificada no contexto, é a táctica `Induction`. Segue-se o resto da prova:

```
inicial < Restart.
inicial < Induction n.
inicial < Left; Auto.
inicial < Intros.
inicial < Right; Elim H;
inicial < [ Intros; Exists zero; Auto | Intro; Exists n0; Auto ].
inicial < Save.
```

Passamos a demonstrar a injectividade e ortogonalidade dos construtores. Para este efeito poderemos recorrer às duas tácticas especiais `Injection` e `Discriminate`, como se exemplifica:

```
Coq < Theorem suc_inj : (x,y:nati) (suc x) = (suc y) -> x = y.
suc_inj < Intros x y H; Injection H; Auto.
suc_inj < Save.
Coq <
Coq < Theorem s_ne_z : (x:nati) ~(suc x) = zero.
s_ne_z < Intro; Discriminate.
s_ne_z < Save.
```

A táctica `Injection` baseia-e na garantia de que os construtores são injectivos, enquanto que a táctica `Discriminate` apoia-se no facto de os construtores serem ortogonais.

[16] Sim, é a mesma que utilizávamos na eliminação de frases lógicas formadas por conectivas não-primitivas.

6.5 Recursão Primitiva

Segue-se a definição das funções de soma e produto de números naturais. Em tipos indutivos, a definição de tais funções é feita por utilização de um construtor especial, dito o **operador de recursão primitiva** do tipo, cuja criação em **Coq** é automática aquando da declaração de cada tipo, sendo o seu nome igual ao do tipo, acrescentado do sufixo `_rec`:[17]

```
Coq < Check nati_rec.
```

As últimas versões do **Coq** disponibilizam uma sintaxe especial, por concordância de padrões (*a la* ML), que evita a invocação explícita do recursor na definição de funções. Vejamos como definir então as funções *soma* e *produto*:

```
Coq < Fixpoint soma [n:nati] : nati -> nati :=
Coq <    [m:nati] Cases n of
Coq <                 zero   => m
Coq <               |  (suc p) => (suc (soma p m))
Coq <               end.
Coq <
Coq < Fixpoint prod [n:nati] : nati -> nati :=
Coq <    [m:nati] Cases n of
Coq <                zero   => zero
Coq <               | (suc p) => (soma m (prod p m))
Coq <               end.
```

`Fixpoint` é um comando específico para a definição de funções recursivas. A construção `Cases ... of ... end` é uma *macro* que permite a escrita de expressões com análise de casos e concordância de padrões. Juntos constituem um esquema genérico para a definição de funções em **Coq** com argumentos de tipos indutivos, no estilo da linguagem **ML**.

Observe os seguintes resultados.

```
Coq < Compute (soma (suc zero) zero).
Coq < Compute (soma (suc zero) (suc (suc zero))).
Coq < Compute (prod (suc (suc zero)) (suc (suc zero))).
Coq < Compute (prod (suc zero) zero).
```

Graças a uma táctica especial, `Simpl`, as provas em **Coq** que envolvam funções definidas por recursão primitiva tornam-se muito simples:

[17] O operador de recursão primitiva é um termo de tipo igual ao princípio de indução do tipo indutivo, mas com `Prop` substituído por `Set`. Em teoria abstracta de tipos, estes dois operadores são de facto um único, que pode ser interpretado de duas formas diferentes, tal como o género `*`.

```
Coq < Theorem dois :
Coq <     (suc (suc zero)) = (soma (suc zero) (suc zero)).
dois < Simpl.
dois < Reflexivity.
dois < Save.
```

A táctica `Simpl` simplifica por recursão primitiva os termos presentes no objectivo de prova.

```
Coq < Theorem um : (prod (suc zero) (suc zero)) = (suc zero).
um < Simpl; Reflexivity.
um < Save.
```

As provas fazem-se de maneira muito mais simples agora do que na definição anterior dos naturais: as reescritas não têm de ser feitas explicitamente. Vejamos mais exemplos disto, agora com provas indutivas:

```
Coq < Theorem Sum1r : (n:nati) (soma n zero) = n.
Sum1r < Induction n; [Simpl; Auto | Simpl; Auto ].
```

Nos casos, como este, em que a aplicação de uma táctica gera dois (ou mais) objectivos aos quais vai ser aplicada a mesma táctica, a táctica composta respectiva pode ser simplificada:

```
Sum1r < Restart.
Sum1r < Induction n; Simpl; Auto.
Sum1r < Save.
Coq < Hint Sum1r.
Coq <
Coq < Theorem Sum2r : (m,n:nati)
Coq <       (soma m (suc n)) = (suc (soma m n)).
Sum2r < Induction m; Simpl; Auto.
Sum2r < Save.
Coq < Hint Sum2r.
```

Questão 9A. Prove o seguinte teorema

$$\forall x.(prod\ x\ zero) = zero$$

Questão 9B. Prove a comutatividade da soma.

Definamos agora a função `pred`, que calcula o antecessor de um número:

```
Coq < Fixpoint pred [d:nati] : nati -> nati :=
Coq <    [n:nati] Cases n of
Coq <                zero    => d
Coq <              | (suc p) => p
Coq <              end.
```

Atente-se na sintaxe utilizada. A função tem que receber um argumento adicional, cujo papel é simplesmente o de elemento a devolver no caso de se tentar calcular o antecessor de *zero*. A função *pred* tem portanto tipo $nati \to nati \to nati$.

Exercício: Considerando que o antecessor de *zero* é *zero*, tente provar que $suc(pred(x)) = x$, e $pred(suc(x)) = x$. Deverá experimentar uma dificuldade, num dos casos.

Exercício: Defina a função $factorial$, e teste-a calculando o factorial de 3. Tente ainda redefinir a mesma função, sem incluir uma cláusula para tratar o natural *zero*, e ainda considerando, na cláusula recursiva, $(n + 1)! = (n + 1) * (n + 1)!$. Interprete os erros que obtém em cada caso.

Para ilustrar diferentes aspectos da concordância de padrões, vamos definir uma função para calcular o máximo de dois números naturais, de diversas maneiras alternativas:

```
Fixpoint max_alt1 [n,m:nati] : nati :=
        Cases n of
            zero   => m
        |  (suc p) => Cases m of
                           zero   => (suc p)
                      |   (suc q) => (suc (max_alt1 p q))
                      end
        end.

Fixpoint max_alt2 [n,m:nati] : nati :=
        Cases n m of
             zero    _      => m
        |   (suc p)  zero   => (suc p)
        |   (suc p) (suc q) => (suc (max_alt2 p q))
        end.

Fixpoint max_alt3 [n:nati] : nati -> nati :=
   [m:nati] Cases n m of
                 zero    _          => m
            |   ((suc p) as N) zero => N
            |   (suc p) (suc q)     => (suc (max_alt3 p q))
            end.
```

Observe-se como em `max_alt1` se têm duas expressões geradas por análise de casos, uma dentro da outra, e como em `max_alt2` se usa concordância de padrões múltipla. Em `max_alt3` usa-se `as` que permite definir localmente (dentro da expressão) um novo identificador.

7 Tipos Indutivos – exemplos

7.1 Pares Ordenados

Relembremos o exemplo das secções 2.2 e 2.3, relativo à definição de um tipo de *pares ordenados* de elementos de quaisquer dois tipos. Tínhamos definido na altura:

$$\begin{array}{rcl} Pair & : & * \to * \to * \\ A, B & : & * \\ mkPair & : & \Pi A, B{:}\,*.\ A \to B \to (Pair\ A\ B) \\ proj1 & : & \Pi A, B{:}\,*.\ (Pair\ A\ B) \to A \\ proj2 & : & \Pi A, B{:}\,*.\ (Pair\ A\ B) \to B \end{array}$$

O maior problema desta definição estava, como referimos na altura, na ausência de qualquer interpretação semântica para as funções construtoras e destrutoras. Outros problemas estavam no facto de nada impedir a geração de $Pair$s por outros meios que não pela função $mkPair$, e de nada garantir a injectividade desta função.

A declaração do tipo dos pares ordenados como tipo indutivo resolve todos estes problemas. Comecemos por declarar este tipo sem preocupações de polimorfismo, recorrendo a dois tipos concretos A e B:

```
Coq < Section Pairs.
Coq < Variables A,B : Set.
Coq < Inductive Pair : Set := mkPair : A -> B -> Pair.
Coq < Check Pair.
```

O tipo indutivo proposicional `Pair` tem portanto um único construtor. Esta definição assegura por um lado a impossibilidade de existência de elementos de tipo `Pair` que não sejam construídos pelo construtor `mkPair`, e por outro a injectividade deste. Permite ainda a definição de funções por análise de casos[18] / concordância de padrões, como se exemplifica com as funções destrutoras:

```
Coq < Fixpoint proj1 [p:Pair] : A :=
Coq <         Cases p of
Coq <            (mkPair x _) => x
Coq <         end.
Coq <
Coq < Fixpoint proj2 [p:Pair] : B :=
Coq <         Cases p of
Coq <            (mkPair _ y) => y
Coq <         end.
Coq <
Coq < Check proj1.
```

[18]neste exemplo, do *único* caso possível.

Para obtermos a versão polimórfica destas definições, e transformar `Pair` numa família de tipos, de género $* \to * \to *$, basta fechar a secção corrente. Observe-se como é de facto útil o mecanismo de secções em **Coq**:

```
Coq < End Pairs.
Coq < Print Pair.
Coq < Check Pair.
Coq < Check mkPair.
Coq < Print proj1.
```

Vejamos ainda como se poderia obter o mesmo efeito na definição do tipo indutivo sem recorrer ao mecanismo de secções.

```
Coq < Reset Pairs.
Coq < Inductive Pair [A:Set;B:Set] : Set :=
Coq <                  mkPair : A->B->(Pair A B).
Coq <
Coq < Fixpoint proj1 [A:Set; B:Set; p: (Pair A B)] : A :=
Coq <        Cases p of
Coq <           (mkPair x _) => x
Coq <        end.
Coq <
Coq < Fixpoint proj2 [A:Set; B:Set; p: (Pair A B)] : B :=
Coq <        Cases p of
Coq <           (mkPair _ y ) => y
Coq <        end.
```

Note-se que na concordância de padrões de elementos de um tipo indutivo parametrizado, os construtores nao esperam os parametros do tipo como argumento. O valor desses parametros é calculado automaticamente pelo sistema. Assim, por exemplo, escrevemos `(mkPair x _)` em vez de `(mkPair A B x _)` na definição de `proj1`.

A definição do tipo indutivo `Pair` é também equivalente à seguinte:

```
Coq < Inductive Pair : Set->Set->Set  :=
Coq <            mkPair : (A,B:Set) A -> B -> (Pair A B).
```

A primeira forma é preferível, uma vez que poupa abstracções nos tipos dos construtores do tipo indutivo, que teriam de ser replicadas se este possuísse vários construtores.

São agora demonstráveis trivialmente as igualdades que de outra forma teriam de ser impostas artificialmente sobre este tipo.

Vejamos, por exemplo, como provar $(mkPair\ (proj1\ p)\ (proj2\ p)) = p$:

```
Coq < Theorem eq1 : (A,B:Set)(p:(Pair A B))
Coq <           (mkPair A B (proj1 A B p) (proj2 A B p)) = p.
eq1 < Intros.
```

Relembre-se que a táctica `Elim` mais não faz do que `Apply` do princípio de indução do tipo (indutivo), neste caso, de `p`, que podemos observar:

```
eq1 < Check Pair_ind.
eq1 < Elim p.
```

Repare como, no objectivo, `p` é substituido pelo padrão `(mkPair A B y y0)`

```
eq1 < Intros.
eq1 < Simpl.
eq1 < Reflexivity.
eq1 < Save.
```

Depois de `Elim`, a táctica `Auto` teria construído a prova num passo.

Exercício: Prove que $proj1\ (mkPair\ x\ y) = x$, para todos os tipos de x e y.

Questão 10A. Relembre a questão 2A. Defina a função `f` polimórfica que recebe um par com ambas as componentes do mesmo tipo e devolve um par em que a primeira componente é a primeira componente do par recebido e a segunda componente é o próprio par. (ex: $\langle 1, 2\rangle \mapsto \langle 1, \langle 1, 2\rangle\rangle$)

Em seguida, prove que qualquer que seja o tipo das componentes do par `p`, se tem

$$f\,p = mkPair\,(proj1\,p)\,p.$$

Questão 10B. Relembre a questão 2B. Defina a função `f´` polimórfica que recebe um par com ambas as componentes do mesmo tipo e devolve um par em que a primeira componente é o próprio par recebido e a segunda componente é a segunda componete do par. (ex: $\langle 1, 2\rangle \mapsto \langle\langle 1, 2\rangle, 2\rangle$)

Em seguida, prove que qualquer que seja o tipo das componentes do par `p`, se tem

$$f'\,p = mkPair\,p\,(proj2\,p).$$

7.2 Listas Polimórficas

Definamos agora o tipo das listas de elementos de (um mesmo) tipo arbitrário, e duas funções básicas sobre elas, a *concatenação* e a *inversão* de listas:

```
Coq < Inductive List [A:Set] : Set :=
Coq <       nil : (List A)
Coq <    | cons : A -> (List A) -> (List A).
Coq <
Coq < Check List.
Coq < Check cons.
Coq < Check nil.
Coq <
Coq < Fixpoint cat [A:Set; l:(List A)] : (List A) -> (List A) :=
Coq <        [l':(List A)] Cases l of
Coq <                          nil      => l'
Coq <                        | (cons h t) => (cons A h (cat A t l'))
Coq <                        end.
Coq <
Coq < Fixpoint rev [A:Set; l:(List A)] : (List A) :=
Coq <              Cases l of
Coq <                  nil      => (nil A)
Coq <              | (cons h t) => (cat A (rev A t) (cons A h (nil A)))
Coq <              end.
```

Comecemos por provar um lema simples mas útil:

```
Coq < Lemma lemma1 : (A:Set)(l:(List A))(cat A l (nil A))=l.
lemma1 < Intro.
lemma1 < Induction l; Clear l.
lemma1 < Auto.
lemma1 < Intros.
lemma1 < Simpl.
lemma1 < Rewrite H.
lemma1 < Auto.
lemma1 < Save.
Coq < Hint lemma1.
```

Note-se que o comando `Clear l` aplica-se a ambos os objectivos de prova gerados por `Induction l`. De facto, depois de aplicado o princípio de indução do tipo `List` a `l`, de nada mais nos serve a sua presença no contexto corrente.

O polimorfismo das listas complica bastante a sintaxe da aplicação das funções. Vamos optar por abdicar desse polimorfismo, mas dentro de uma secção onde o tipo seja parametrizado, para que se possa recuperar automaticamente o polimorfismo ao fechar a secção:

```
Coq < Reset List.
Coq < Section SetA.
Coq < Variable A:Set.
Coq <
Coq < Inductive List : Set :=
Coq <            nil : List
Coq <         | cons : A -> List -> List.
Coq <
Coq < Check List.
Coq < Check cons.
Coq < Check nil.
Coq <
Coq < Fixpoint cat [l:List] : List -> List :=
Coq <      [l':List] Cases l of
Coq <                  nil  => l'
Coq <                | (cons h t) => (cons h (cat t l'))
Coq <                end.
Coq <
Coq < Fixpoint rev [l:List] : List :=
Coq <             Cases l of
Coq <                nil   => nil
Coq <             | (cons h t) => (cat (rev t) (cons h nil))
Coq <             end.
Coq <
Coq < Token "::".
Coq < Token "@".
Coq < Infix 7 "::" cons.
Coq < Infix 6 "@" cat.
```

Utilizámos também aqui as facilidades sintáticas do **Coq** para a utilização de operadores infixos. A precedência de cada operador tem obrigatoriamente de ser um inteiro entre 6 e 9, sendo a precedência tanto maior quanto menor for aquele valor. Note-se que todos os operadores assim definidos são *associativos à direita*.

Efectuemos então de novo a mesma prova, reparando na notação muito simplificada:

```
Coq < Lemma lemma1 : (l:List) ((l @ nil) = l).
lemma1 < Induction l; Clear l.
lemma1 < Auto.
lemma1 < Intros.
lemma1 < Simpl.
lemma1 < Rewrite H.
lemma1 < Auto.
lemma1 < Save.
Coq < Hint lemma1.
```

Continuemos com uma prova mais complexa: a operação de concatenação é associativa.

```
Coq < Theorem catAssoc : (l1,l2,l3:List)
Coq <         (l1 @ (l2 @ l3)) = (cat (l1 @ l2) l3).
catAssoc < Induction l1; Clear l1.
catAssoc < Auto.
catAssoc < Intros.
catAssoc < Simpl.
catAssoc < Rewrite H.
catAssoc < Auto.
catAssoc < Save.
Coq < Hint catAssoc.
```

Ainda outra prova: A lista inversa da concatenação de duas listas é a concatenação das suas inversas, por ordem contrária:

```
Coq < Theorem revCat : (l1,l2:List)
Coq <       (rev (l1 @ l2)) = (cat (rev l2) (rev l1)).
revCat < Induction l1; Clear  l1.
revCat < Intro.
revCat < Simpl.
revCat < Rewrite lemma1; Auto.
revCat < Intros.
revCat < Simpl.
revCat < Rewrite H.
revCat < Auto.
Coq < Save.
```

O sistema Coq quando arranca carrega, por defeito, o módulo *Prelude* aonde estão já definidos, entre outras coisas, os naturais e algumas funções e predicados sobre números naturais. Observe, atentamente, o resultado dos seguintes comandos:

```
Coq < Print nat.
Coq < Print plus.
Coq < Print le.
```

Questão 11A. Defina em Coq a função `comp` que calcula o comprimento de uma lista.

Prove que para quaisquer `x:A` e `l:List`, o comprimento de `l` é sempre menor ou igual ao comprimento de `(cons x l)`.

Questão 11B. Defina em Coq a função `comp` que calcula o comprimento de uma lista.

Prove que para quaisquer `l, l':List`, o comprimento de `(l @ l')` é igual à soma dos comprimentos de `l` e de `l'`.

A próxima definição, da função de pertença de um elemento a uma lista, trata-se agora da definição de um *predicado*, uma função com `Prop` por co-domínio.

```
Coq < Fixpoint member [l:List] : A -> Prop :=
Coq <      [x:A] Cases l of
Coq <                      nil => False
Coq <                 | (cons h t) => (x=h) \/ (member t x)
Coq <            end.
```

Uma definição alternativa (com um tipo diferente) desta função seria:

```
Coq < Fixpoint member2 [x:A; l:List] : Prop :=
Coq <     Cases l of nil   => False
Coq <       |  (cons h t)  => (x=h) \/ (member2 x t)
Coq <     end.
```

Efectuemos agora uma prova usando este predicado: se um elemento pertence a uma de duas listas, então ele pertence seguramente à sua concatenação:

```
Coq < Theorem membcat1 : (l1,l2:List) (x:A)
Coq <     (member l1 x)\/(member l2 x) -> (member (l1 @ l2) x).
membcat1 < Induction l1; Clear  l1.
```

Efectuamos indução sobre a primeira lista. O caso de `l1 = nil` é simples:

```
membcat1 < Intros.
membcat1 < Simpl.
membcat1 < Elim H; Clear  H.
```

A eliminação da disjunção criou dois objectivos a partir do primeiro. Ambos são fáceis de eliminar:

```
membcat1 < Intro H.
membcat1 < Elim H.
membcat1 < Auto.
```

O caso indutivo requer um pouco mais de cuidado. Convém não caír na tentação de fazer o número máximo de introduções possível, uma vez que uma das hipóteses se presta a ser simplificada antes de introduzida como hipótese:

```
membcat1 < Intros a l H l2 x.
membcat1 < Simpl.
membcat1 < Intro H0.
```

A prova procede por eliminação de disjunções e por aplicação da hipótese de indução `H`:

```
membcat1 < Elim H0; Clear H0.
membcat1 < Intro H0; Elim H0; Clear H0.
membcat1 < Auto.
membcat1 < Intro H0.
membcat1 < Right.
membcat1 < Apply H.
membcat1 < Auto.
membcat1 < Intro H0.
membcat1 < Right.
membcat1 < Apply H.
membcat1 < Auto.
membcat1 < Save.
```

Exercício: A estratégia acima utilizada para a construção da prova não é a mais fácil. Tente efectuar a mesma prova começando por introduzir no contexto as hipóteses e eliminando em seguida a disjunção. Em cada um dos objectivos de prova resultantes, deverá então efectuar raciocínio indutivo separadamente.

Exercício: Prove a implicação inversa da que se provou acima.

Exercício: Prove que um elemento pertence a uma lista se e só se pertence à sua inversa.

7.3 Predicados como definições indutivas

Para finalizar, vamos definir o predicado `member` de uma forma alternativa, cuja utilidade justificaremos apenas na próxima sessão de trabalho. O predicado, que continuará a ter o mesmo tipo `List -> A -> Prop`, será definido ele próprio indutivamente. Para isso, ele terá dois construtores, cada um deles correspondendo a uma das duas situações em que um elemento pode ocorrer numa lista: ou é a sua cabeça, ou pertence à sua cauda.

```
Coq < Inductive memberP : List -> A -> Prop :=
Coq <    memberPhead : (x,h:A)(t:List)
Coq <                  (x=h) -> (memberP (cons h t) x)
Coq <  | memberPtail : (x,h:A)(t:List)
Coq <                  (memberP t x) -> (memberP (cons h t) x).
Coq <
Coq < Hint memberPhead memberPtail.
```

Esta definição introduz um novo predicado `memberP:List->A->Prop` e dois construtores `memberPhead` e `memberPtail` que são as cláusulas definidoras de `memberP`. Isto é, temos não só os "axiomas" `memberPhead` e `memberPtail` mas também a propriedade reciproca de um `(memberP l a)` se e só se essa asserção pode ser obtida como consequência dessas cláusulas definidoras. Ou seja, `memberP` é o "mais fraco" predicado que verifica `memberPhead` e `memberPtail`.

Veremos que a declaração dos construtores como `Hints` será muito útil. Observe-se que a seguinte definição alternativa não é válida em **Coq**:

```
Coq < Inductive memberP : List -> A -> Prop :=
Coq <    memberPnil  : (x:A) ~(memberP nil x)
Coq <  | memberPhead : (x,h:A)(t:List)
Coq <                  (x=h) -> (memberP (cons h t) x)
Coq <  | memberPtail : (x,h:A)(t:List)
Coq <                  (memberP t x) -> (memberP (cons h t) x).
```

De facto, o construtor `memberPnil` tem, se relembrarmos a definição da negação, um tipo em que o tipo que está a ser definido ocorre do lado esquerdo de um tipo produto. Diz-se que se trata de uma *ocorrência negativa* do tipo.

Não é, no entanto, difícil entender que o construtor `memberPnil` seria redundante. Não pode haver termos deste tipo indutivo construídos por qualquer outro meio que não por um dos construtores `memberPhead` e `memberPtail`, pelo que não é necessário explicitar, neste exemplo, que nenhum elemento pode pertencer à lista vazia.

Vejamos então como construir a mesma prova de há pouco:

```
Coq < Theorem membPcat1 : (l1,l2:List) (x:A)
Coq <  (memberP l1 x)\/(memberP l2 x)->(memberP l1 @ l2 x).
membPcat1 < Intros l1 l2 x H.
membPcat1 < Elim H; Clear H.
membPcat1 < Intro H.
```

Começámos por eliminar a hipótese disjuntiva. No primeiro objectivo daí resultante, introduzimos como hipótese $(memberP\ l1\ x)$.

Façamos agora uma pausa para observar o princípio de indução do predicado `memberP`:

```
membPcat1 < Check memberP_ind.
```

Justifica-se a sua reprodução em notação tradicional:

$$\begin{array}{ll} & \Pi P : List \to A \to *. \\ & (\Pi x, h : A.\ \Pi t : List.\ (x = h) \to (P\ (h :: t)\ x)) \\ \to & (\Pi x, h : A.\ \Pi t : List.\ (memberP\ t\ x) \to (P\ t\ x) \to (P\ (h :: t)\ x)) \\ \to & (\Pi l : List.\ \Pi a : A.\ (memberP\ l\ a) \to (P\ l\ a)) \end{array}$$

Observando atentamente a última linha desta definição, conclui-se que este princípio de indução pode ser utilizado para mostrar, a partir de uma hipótese da forma $(memberP\ l\ a)$, qualquer frase $(P\ l\ a)$, com P um predicado de tipo $P : List \to A \to Prop$. Quando o objectivo for desta forma[19] e existir uma hipótese como a referida, a táctica `Elim`, invocada com a dita hipótese, terá como efeito a substituição do objectivo por dois outros, correspondentes às segunda e terceira linhas acima reproduzidas.

O estado de prova actual presta-se a esta operação: O primeiro objectivo é $(memberP\ l1@l2\ x)$, que pode ser visto como $(P\ l1\ x)$, com $P \doteq (\lambda l\colon List.\ \lambda x\colon A.(memberP\ l@l2\ x)) : List \to A \to Prop$. Podemos então eliminar a hipótese $H : (memberP\ l1\ x)$:

```
membPcat1 < Elim H; Clear H.
```

Esta eliminação gera os dois objectivos referidos, o primeiro dos quais tratamos da seguinte forma:

```
membPcat1 < Simpl.
membPcat1 < Intros.
membPcat1 < Rewrite H.
membPcat1 < Apply memberPhead.
```

[19] em rigor, quando a unificação de ordem superior for possível.

Esta sequência de tácticas, que culmina na eliminação do objectivo por aplicação do primeiro construtor de `memberP`, pode ser feita de forma mais automática, graças ao conhecimento dos construtores como `Hints`:

```
membPcat1 < Undo 4.
membPcat1 < Simpl; Auto.
membPcat1 < Simpl; Auto.
```

O segundo objectivo foi também eliminado automaticamente, pelo que nos resta um único, nomeadamente $(memberP\ l2\ x) \rightarrow (memberP\ l1@l2\ x)$. Também este poderia ser tratado por introdução e eliminação da hipótese como há pouco, mas neste ponto essa estratégia complicaria muito a prova. De facto, é impossível provar, só com recurso aos construtores de `memberP`, este segundo objectivo, pelo que optamos por uma indução tradicional em $l1$:

```
membPcat1 < Elim l1; Clear l1.
membPcat1 < Auto.
membPcat1 < Simpl; Auto.
membPcat1 < Save.
```

Do ponto de vista formal, notemos que a definição de `memberP` corresponde à de um termo de tipo $List \rightarrow A \rightarrow *$, ou seja, uma família de tipos, tal como `Pair` no início desta sessão. A diferença está no facto da indexação desses tipos ser feita agora por termos dos tipos $List$ e A, em vez de por outros tipos, como era o caso com `Pair`.

Exercício: Considere a seguinte definição indutiva, alternativa para a definição de `rev`:

```
Coq < Inductive revP : List -> List -> Prop :=
Coq <       revPnil : (revP nil nil)
Coq <    | revPcons : (h:A)(t,tr,lh:List)
Coq <         (revP t tr) -> (lh = (cat tr (cons h nil))) ->
Coq <                   (revP (cons h t) lh).
Coq <
Coq < Hint revPnil revPcons.
```

Observe atentamente o princípio de indução desta definição e tente provar o seguinte teorema:

```
Coq < Theorem revRevP : (l,lr:List) (revP l lr) -> (revP lr l).
```

Podemos finalmente fechar a secção actual para transportar todos os resultados obtidos para as listas polimórficas:

```
Coq < End SetA.
```

7.4 Ainda as conectivas lógicas e a igualdade em Coq

Depois da definição de `memberP`, não será difícil entender finalmente a forma como em **Coq** as conectivas lógicas são definidas indutivamente, e a razão pela qual a táctica `Elim` se utiliza para o tratamento de hipóteses construídas com estas conectivas.

Analisamos atentamente os seguintes resultados:

```
Coq < Print and.
Coq < Check and.
Coq < Check and_ind.
Coq < Print or.
Coq < Check or.
Coq < Check or_ind.
Coq < Print False.
Coq < Check False.
Coq < Check False_ind.
```

Cada definição possui tantos construtores quantas as formas possíveis de se provar um teorema construído com a respectiva conectiva lógica:[20] duas para a disjunção, uma para a conjunção, e nenhuma para o absurdo. O tipo de cada construtor não suscita quaisquer dúvidas.

Os princípios de indução gerados automaticamente, por seu lado, correspondem às definições de segunda ordem de cada conectiva, como já tínhamos visto. É por esta razão que a utilização da táctica `Elim` é útil para a eliminação de hipóteses destes tipos indutivos.

Assim, por exemplo, o princípio de indução da disjunção será um termo

$$or\_ind : \Pi A, B, P : Prop.\ (A \to P) \to (B \to P) \to (A \vee B) \to P.$$

Se H for um termo de tipo $U \vee V$, o comando `Elim H` permite substituir o objectivo actual de prova, X, por dois outros, $U \to X$ e $V \to X$.

Também a igualdade em **Coq** é definida indutivamente e merece a nossa atenção:

```
Coq < Print eq.
Coq < Check eq.
Coq < Check eq_ind.
```

[20] correspondendo a outras tantas regras de introdução da conectiva, em Dedução Natural.

Reproduzimos aqui a definição indutiva bem como o princípio de indução:

$$\begin{aligned} \mathit{Inductive}\ eq : \quad & \Pi A{:}\,{*}.\ A \to A \to {*} \doteq \\ & refl\_equal : \Pi A{:}\,{*}.\ \Pi x{:}\,A.\ (eq\ A\ x\ x) \\ eq\_ind : \quad & \Pi A{:}\,{*}.\ \Pi x{:}\,A.\ \Pi P{:}\,A \to {*}.\ (P\ x) \to \Pi y{:}\,A.\ (eq\ A\ x\ y) \to (Py) \end{aligned}$$

Note-se que apesar de se tratar de uma definição polimórfica no tipo A, a sintaxe concreta com o operador = dispensa que se forneça o primeiro argumento em cada aplicação do operador, e permite que este seja utilizado na forma infixa. Aqui preferimos abdicar destas facilidades sintáticas.

O que se conclui da definição acima é pois, que a igualdade possui um único construtor $refl\_equal$, cujo tipo exprime a reflexividade da relação. O princípio de indução assim gerado não é mais do que a expressão da **Igualdade de Leibniz**, descrita na sessão de trabalho anterior.

8 Introdução à verificação e síntese de programas em Coq

Nesta sessão de trabalho introduzimos algumas facilidades do sistema **Coq** de apoio ao desenvolvimento de *programas certificados*. Em particular, efectuaremos dois tipos de exercício com interesse: a extracção de programas a partir de provas das suas especificações, e a construção automática de provas de correcção de programas face às suas especificações.

Como caso de estudo, continuaremos a utilizar o tipo indutivo das Listas Polimórficas, na sequência da sessão de trabalho anterior. Vamos estudar uma função muito simples, a *concatenação* de listas, para a qual obtivemos uma possível realização na sessão anterior.

O seguinte prelúdio é suficiente para nos permitir trabalhar nesta sessão:

```
Coq < Require Natural.
Coq < Require Extraction.
Coq <
Coq < Section SetA.
Coq < Variable A:Set.
Coq < Inductive List : Set :=
Coq <            nil : List
Coq <         | cons : A -> List -> List.
Coq < Token "::".
Coq < Infix 1 "::" cons.
```

8.1 Extracção de Programas

Comecemos por escrever uma especificação da função, através de um predicado definido indutivamente. Este predicado pode ser visto como um programa lógico que realiza a função:

```
Coq < Inductive catP : List -> List -> List -> Prop :=
Coq <       catP_nil : (l2:List) (catP nil l2 l2)
Coq <    | catP_cons : (l,l2,lt:List)(x:A) (catP l l2 lt) ->
Coq <                  (catP (cons x l) l2 (cons x lt)).
Coq <
Coq < Hint catP_nil catP_cons.
```

A definição deste tipo indutivo constitui de facto uma especificação do comportamento de uma função de concatenação de listas. É demonstrável que para cada par de listas `l1` e `l2`, existe uma outra que é a sua concatenação, como passamos a verificar:

```
Coq < Theorem proofCat : (l1,l2:List) (Ex [l:List] (catP l1 l2 l)).
proofCat <
proofCat < Induction l1; Clear  l1; Intros.
proofCat < Exists  l2.
proofCat < Auto.
proofCat < Elim (H l2); Clear  H.
proofCat < Intros x H.
proofCat < Exists  (cons a x).
proofCat < Elim H; Clear  H.
proofCat < Auto.
proofCat < Auto.
proofCat < Save.
```

A prova envolve indução na lista `l1` e na própria definição de `catP`, e garante que qualquer processo de se calcular `l3` a partir de `l1` e `l2` de acordo com `catP` define uma função total. Observe-se a estrutura indutiva da prova:

```
Coq < Print Natural proofCat.
```

Esta prova contém, ela própria, informação de carácter computacional, descrevendo *como* é obtida a concatenação de `l1` com `l2`. De facto, tratando-se de uma prova intuicionista (como em toda a teoria abstracta de tipos) de uma quantificação existencial, essa prova apresenta obrigatoriamente um *testemunho* dessa quantificação (um elemento que satisfaz o predicado quantificado). Esse testemunho é `l3`.

Quer isto dizer que, ao construir-se interactivamente uma prova de que, para qualquer par de listas, existe uma outra que é a sua concatenação, essa prova contém obrigatoriamente o *processo* de construção dessa lista. De acordo com a própria definição indutiva de `catP`, esse processo avança naturalmente por análise de casos no primeiro argumento da concatenação, fornecendo um testemunho diferente para cado caso.

Desejamos então extraír desta prova a sua componente informativa (em `Set`), esquecendo a componente lógica (em `Prop`). Para isso comecemos por repetir a prova acima, mas desta feita considerando uma interpretação como `Set` para `*`, em vez de `Prop`. Note-se que em termos formais encontrar uma prova para um teorema (de tipo `Prop`) corresponde a encontrar um termo do tipo (de tipo `Set`) correspondente a esse teorema.

Observe-se a seguinte sintaxe, correspondente à quantificação existencial em `Set`, e repita-se então a construção da prova, por forma a construir um termo do tipo indicado:[21]

[21] Um estudo aprofundado das propriedades e utilizações possíveis destes "tipos existenciais" sai fora do âmbito desta sessão de trabalho.

```
Coq < Check (l1,l2:List) { l:List |  (catP l1 l2 l) }.
Coq < Theorem progCat : (l1,l2:List) { l:List |  (catP l1 l2 l) }.
progCat <  ............
progCat < Save.
```

O **Coq** fornece um processo pelo qual o conteúdo computacional de uma prova pode ser extraído e traduzido para uma de várias linguagens de programação possíveis.[22] O termo extraído de uma prova com conteúdo informativo (computacional) é obtido pelo **Coq** 'esquecendo' a parte lógica da prova, e pode ser visualizado (bem como o seu tipo) com o comando `Extraction`. O termo extraído é sempre um termo de λ_ω com definições indutivas, ou seja, um termo sem a presença de tipos dependentes de termos.

Observemos então o termo extraído, e veja-se como o seu tipo é, como seria de esperar, $List \to List \to List$, correspondente ao tipo da função de concatenação de listas:

```
Coq < Extraction progCat.
```

Para traduzir o termo extraído para, por exemplo, **Caml**, basta fazer:

```
Coq < Write Caml File "cat" [progCat].
```

A possibilidade de ocorrência do erro obtido (impossibilidade de atribuir um tipo em ML à função `cat`) é compreensível se tivermos em conta que o sistema de tipos desta linguagem é mais restrito do que o de λ_ω. No entanto, trata-se aqui de um caso simples de resolver: basta fechar a secção corrente para tornar realmente polimórficas as funções que temos escrito, e tentar de novo:

```
Coq < Save State s1.
Coq < End SetA.
Coq < Write Caml File "cat" [progCat].
Coq < Restore State s1.
```

Este comando terá como efeito a criação de um ficheiro `cat.ml`, contendo uma função `progCat` em Caml, que realiza a concatenação de duas listas. O ficheiro contém ainda uma tradução do tipo indutivo das listas polimórficas, necessário para a definição da função.[23]

A utilização do estado `s1` acima justifica-se apenas pelo facto de desejarmos continuar a trabalhar com listas de um tipo `A`, sem toda a sintaxe adicional associada às listas polimórficas.

[22] Nomeadamente, vários dialectos de **ML** ou **Gofer**.

[23] Não são portanto utilizados por este *package* os tipos pré-definidos do Caml, como sejam as listas polimórficas.

8.2 Verificação de Programas

Relembremos agora a versão da função `cat` que havíamos definido na aula anterior:

```
Coq < Fixpoint cat [l1:List] : List -> List :=
Coq <     [l2:List]
Coq <         Cases l1 of
Coq <                nil => l2
Coq <      | (cons h t) => (cons  h (cat t l2))
Coq < end.
```

Se esquecermos o que fizemos nesta sessão até este ponto – a síntese do programa **ML** que implementa esta função – podemo-nos questionar agora sobre a *correcção* da função `cat` em relação à especificação inicial, expressa na definição indutiva `catP`. Este problema, da *Verificação de Programas*, é formalmente distinto do anterior – não temos agora o esforço criativo de realizar um programa que implemente uma especificação; o programa é dado e queremos apenas verificar que ele implementa correctamente a especificação.

Tratando-se de um problema mais simples, espera-se que a sua resolução seja mais automatizável em **Coq**. De facto é-o, devido à existência de tácticas especiais que automatizam as provas de correcção. Por prova de correcção entende-se, aqui, uma prova de que o termo em questão é uma função total que obedece à especificação dada pela definição indutiva de `catP`.

Retomemos então a prova de que existe sempre a concatenação de duas listas, de acordo com a especificação expressa pelo tipo `catP`:

```
Coq < Theorem proofCatImpl : (l1,l2:List) { l:List |  (catP l1 l2 l) }.
```

Associemos agora a esta prova a função que supostamente implementa a especificação:

```
proofCatImpl < Realizer cat.
```

e finalmente recorramos à táctica especial `Program` para efectuar esta prova.

```
Coq < Program.
Coq < Auto.
```

O efeito da táctica `Program` é o de introduzir no contexto a declaração de um termo `cat` construído a partir da função `cat`, e cujo conteúdo informativo é uma função com o tipo de `cat`.

No caso corrente, esta declaração introduzida por `Program` tem exactamente o tipo correspondente ao objectivo da prova, pelo que esta se conclui trivialmente. Noutras circunstâncias, a táctica `Program_all` pode-se revelar mais útil, uma vez que repete a aplicação de `Program` e `Auto` enquanto possível.

A conclusão com sucesso desta prova pela táctica `Program` indica que o termo `cat` realiza correctamente a especificação `catP`, pelo que se pode afirmar que aquele termo constitui um programa *certificado*.

Exercício: Aplique as metodologias estudadas neste guião (Extracção e Verificação) à função que calcula o comprimento de uma lista. Apresente de forma devidamente comentada todos os passos que efectuar.

Referências

[1] José Manuel Valença, *Sebenta Teórica de Elementos Lógicos da Programação III* (Manuscrito). Universidade do Minho, Departamento de Informática, 1996/97.

[2] José Manuel Valença, *Introdução à Lógica de Ordem Superior e Sistemas de Prova Assistida.* Universidade do Minho, Departamento de Informática, 1996/97.

[3] Henk Barendregt. *Lambda calculi with types.* In Samson Abramsky, D. M. Gabbai, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1991.

[4] Projet Coq. *The Coq Proof Assistant - A Tuturial.* Technical Report, INRIA-Rocquencourt - CNRS-ENS Lyon, 1996

[5] Projet Coq. *The Coq Proof Assistant - Reference Manual.* Technical Report, INRIA-Rocquencourt - CNRS-ENS Lyon, 1996.

[6] Projet Coq. *The Coq Proof Assistant - Standard Library.* Technical Report, INRIA-Rocquencourt - CNRS-ENS Lyon, 1996.

[7] Eduardo Giménez. *A Tuturial on Recursive Types in Coq.* Technical Report, INRIA-Rocquencourt - CNRS-ENS Lyon, 1996.

Elementos Lógicos da Programação III
Exame Prático (LMCC 3. Ano) 1997/98

Utilize o Sistema Coq para responder às questões que se seguem. Indique na sua folha de resposta todas as declarações, definições e comandos Coq que utilizou para resolver os problemas apresentados.

1. Abra uma secção de nome, `questao1`, e construa um contexto adquado à definição das expressões K e M.

$$K \doteq \lambda X{:}\ast.\lambda Y{:}\ast.\lambda x{:}X.\lambda y{:}Y.\, x$$
$$M \doteq \lambda h{:}(\Pi C{:}\ast.(A \to B \to C) \to C).\, h\, A\, (K\, A\, B)$$

1.1 Defina as expressões-λ K e M.

1.2 Apresente os contextos mínimos Γ_1 e Γ_2 e os tipos T_1 e T_2 para os quais é possível construir os juizos $\Gamma_1 \vdash K{:}T_1$ e $\Gamma_2 \vdash M{:}T_2$.

1.3 Que dependências são necessárias para garantir a boa formação do termo K ?

1.4 Calcule a forma normal da seguinte expressão-λ

$$\lambda a{:}A.\lambda b{:}B.\,(M\,(\lambda D{:}\ast.\lambda z{:}A \to B \to D.\, z\, a\, b))$$

Use o Coq para verificar a sua resposta.

1.5 Feche agora a secção `questao1`. Qual o efeito do fecho desta secção ?

2. Construa em Coq uma prova para o seguinte teorema de segunda ordem:

$$\forall A, B.\ A \to \neg A \to B$$

Qual o termo que codifica a prova ? Comente-o.

3. Prove o seguinte teorema da lógica de predicados:

$$(\exists x.\, P(x) \wedge Q(x)) \to (\forall z.\exists y.\, Q(y) \vee R(z))$$

4. Relembre a definição, do Coq, dos números naturais e da soma de naturais:

```
Coq < Print nat.
Coq < Print plus.
```

4.1 Defina o tipo indutivo, `natList`, das listas de números naturais.

4.2 Defina a função, `soma`, que calcula o somatório dos elementos de uma lista de naturais.

4.3 Prove que: `soma [2, 1, 0] = 3`

5. Defina indutivamente o predicado *maior1*, sendo *maior1(x)* verdade, se x for um número natural maior do que 1. Prove que

$$\forall x, y.\ maior1(x) \wedge maior1(y) \to maior1(x+y)$$