Guiões das aulas práticas laboratoriais sobre o Sistema Coq

de

Elementos Lógicos da Programação II

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

Departamento de Informática
Universidade do Minho
1998

Introdução

O presente texto resulta da compilação dos guiões elaborados para as aulas laboratoriais da disciplina de Elementos Lógicos de Programação II, leccionada no segundo semestre do ano lectivo de 1997/98 ao 2º 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.

Em cada sessão foram 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.

Conteúdo

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. A forma como as expressões-λ são representadas em Coq pode ser resumidamente descrito pelo seguinte quadro:

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

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

1.1 Declarações locais e globais. O mecanismo de secções

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

```
Coq < Variable sigma: Set.
Coq < Variable tau : 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.

[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`.

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

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

Todo o restante trabalho desta aula será efectuado dentro do contexto `s1`. Note-se no entanto que isso não nos impede de efectuarmos declarações globais em qualquer momento, com o comando `Parameter`:

```
Coq < Parameter g : tau.
```

1.2 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:sigma] x.
Coq < Check [x:sigma] [y:tau] x.
```

Os dois termos cujo tipo pedimos ao sistema para inferir correspondem, na notação tradicional do λ-calculus, a $(\lambda x\colon \sigma.x)$ e $(\lambda x\colon \sigma.\lambda y\colon \tau.x)$. Note que se trata de termos do λ-calculus com tipos simples (designado $\lambda_{\rightarrow}$), 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.

1.3 Definições locais e globais

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

```
Coq < Definition idSigma := [x:sigma] x.
Coq < Local KSigmaTau := [x:sigma] [y:tau] 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 idSigma.
Coq < Print KSigmaTau.
```

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.
```

1.4 Secções revisitadas

Procedamos agora ao *fecho* da secção corrente:

```
Coq < End s1.
```

Qual o efeito do fecho da secção `s1`?

Para concluir o estudo das secções, avaliemos agora a seguinte sequência de comandos:

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

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

A interpretação do papel das secções como contextos de juízos do sistema 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

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

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.

1.5 Redução de termos

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

```
Coq < Variable A : Set.
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 : sigma.
Coq < Eval (idSigma a).
Coq < Compute (idSigma a).
```

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`.

1.6 Outros exemplos

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

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

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

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

$$double \doteq \lambda f\!:\sigma \to \sigma.\lambda x\!:\sigma.f(fx)$$

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

1.7 Mais alguns exemplos e questões ...

Para apagar todo o contexto, voltando ao contexto inicial, pode fazer-se

---

```
Coq < Reset Initial.
```

Experimente-o. É equivalente a sair e voltar a entrar no sistema Coq.

Vejamos então mais alguns exemplos ...

Considere os combinadores $F \doteq \lambda xy.y$ e $T \doteq \lambda xy.yx$ do λ-calculus sem tipos. Vamos agora converte-los ao sistema de tipos de Church, anotando as variáveis de abstracção com o respectivo tipo.

Assim, podemos definir F do seguinte modo:

```
Coq < Variable A, B : Set.
Coq < Definition F := [x:A][y:B] y.
```

---

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

Veja agora qual é o seu tipo (usando o comando `Check`). Repare que os tipos `A` e `B` foram previamente declarados no contexto.

No caso de T, dado o seu formato, precisamos de indicar que y é do tipo função (pois y está a ser aplicado a x) e o tipo de x deve coincidir com o domínio de y. Assim, podemos fazer

```
Coq < Definition T := [x:A][y:A->B] (y x).
```

Verifique o seu tipo e teste-o.

Questão 1A. Considere o combinador

$$S \doteq \lambda xya.xa(ya)$$

Qual terá de ser o tipo mais genérico das variáveis para que este combinador seja válido no λ-calculus com tipos. Defina o combinador S em Coq, e verifique o seu tipo.

Questão 1B. Considere o combinador

$$P \doteq \lambda xya.axy$$

Qual terá de ser o tipo mais genérico das variáveis para que este combinador seja válido no λ-calculus com tipos. Defina o combinador P em Coq, e verifique o seu tipo.

Abra agora a seguinte secção

```
Coq < Section exemplo.
Coq < Variable C, D : Set.
Coq < Variable c : C.
Coq < Variable d : D.
Coq < Definition ex := (([x:C->D][y:C](x y))([y:C] d)(([y:C] y) c))
```

Indique a expressão-λ definida por `ex`. Quais as variáveis livres e ligadas de `ex` ?
Para saber qual a forma normal de `ex` podemos fazer

```
Coq < Compute ex.
```

Confira o resultado encontrado escrevendo no papel uma cadeia de redução maximal de raiz `ex`. Qual o tipo de `ex` ?

Faça agora

```
Coq < End exemplo.
```

Qual o efeito do fecho da secção `exemplo` ?

Questão 2A. Abra uma nova secção e declare localmente o seguinte contexto.

$$P : * \ , \ p : P$$

Calcule a forma normal da expressão $(\lambda x\colon P.(\lambda y\colon P.\lambda z\colon P \to P.zy)x)p(\lambda x\colon P.x)$ usando para isso a estratégia *lazy evaluation*. Use o Coq para conferir a forma por si encontrada.

Questão 2B. Abra uma nova secção e declare localmente o seguinte contexto.

$$T : * \ , \ t : T$$

Calcule a forma normal da expressão

$$(\lambda x\colon T \to T \to T.\lambda y\colon T \to T.\lambda z\colon T.xz(yz))(\lambda a\colon T.\lambda b\colon T.a)(\lambda x\colon T.x)t$$

usando para isso a estratégia *lazy evaluation*. Use o Coq para conferir a forma por si encontrada.

Questão 3A. Considere a seguinte função do λ-calculus com tipos simples, que efectua a composição de funções.

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

Efectue em Coq a definição de um λ-termo que represente esta função *comp*, verifique qual é o seu tipo, e teste-a.

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

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

Efectue em Coq a definição de um λ-termo que represente esta função fun, verifique qual é o seu tipo, e teste-a.

Para apagar todo o contexto e voltar ao contexto inicial, pode também fazer-se

```
Coq < Restore State Initial.
```

Experimente-o.

Vamos agora escrever, em Coq, as seguintes declarações e definições:

$A\colon *$
$B \doteq \lambda y\colon A \to A \to A.\lambda z\colon A \to A.\lambda x\colon A.y(zx)$
$K \doteq \lambda x\colon A \to A \to A.\lambda y\colon A.x$
$I \doteq \lambda x\colon A.x$

```
Coq < Variable A : Set.
Coq < Definition B := [y:A->A->A][z:A->A][x:A](y (z x)).
Coq < Definition K := [x:A->A][y:A] x.
Coq < Definition I := [x:A] x.
```

Vamos agora ver como são as cadeias de redução-β maximais de raiz em $B(KI)I$ que resultam da utilização das estratégias *Lasy Evaluation* e *Eager Evaluation*.

Lasy Evaluation

$$\underline{B(KI)I} \to_\beta \underline{(\lambda z\colon A \to A.\lambda x\colon A.(KI)(zx))I} \to_\beta \lambda x\colon A.\underline{(KI)}(Ix) \to_\beta \lambda x\colon A.\underline{(\lambda y\colon A.I)(Ix)} \to_\beta \lambda x\colon A.I$$

Eager Evaluation

$$B\underline{(KI)}I \to_\beta \underline{B(\lambda y\colon A.I)}I \to_\beta (\lambda z\colon A \to A\lambda x\colon A.\underline{(\lambda y\colon A.I)(zx)})I \to_\beta \underline{(\lambda z\colon A \to A.\lambda x\colon A.I)I} \to_\beta \lambda x\colon A.I$$

Como era de esperar, ambas as cadeias têm o mesmo objectivo. Esse objectivo é α-equivalente a $(\lambda x\colon A.\lambda y\colon A.y)$.

Podemos conferir estes resultados com o sistema Coq:

```
Coq < Compute (B (K I) I).
```

Se quiser sair do Coq, faça:

```
Coq < Quit.
```

---

2 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 $\{*, \square\}$ 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 $s, s_1, s_2 \in \{*, \square\}$.

Regras Gerais

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

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

A regra $\langle s_1, s_2 \rangle$ tem várias instâncias consoante s_1 e s_2 tomam o valor de $*$ ou $\square$. 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 ($\langle *, \Box \rangle, \langle \Box, * \rangle$ e $\langle \Box, \Box \rangle$) identificam-se oito *Sistemas Abstractos de Tipos* diferentes, a que se dão nomes específicos (ver quadro).

Dependências

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

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.

2.1 Termos generalizados

Vamos começar pela seguinte definição

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

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

```
Coq < Check id.
```

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

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

Vejamos como poderia ser feita a prova formal do juizo $\vdash (\lambda A{:}\, *. \lambda x{:}\, A.x) : \Pi A{:}\, *. A \rightarrow A$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash * : \Box}{A : * \vdash A : *}}{A : *, x : A \vdash x : A} \quad \dfrac{\dfrac{\vdash * : \Box}{A : * \vdash A : *} \quad \dfrac{\vdash * : \Box}{A : * \vdash A : *}}{A : * \vdash A \rightarrow A : *}}{A : * \vdash (\lambda x{:}\, A.x) : A \rightarrow A} \quad \dfrac{\vdash * : \Box \quad \dfrac{\dfrac{\vdash * : \Box}{A : * \vdash A : *} \quad \dfrac{\vdash * : \Box}{A : * \vdash A : *}}{A : * \vdash A \rightarrow A : *}}{\vdash (\Pi\, A{:}*.\, A \rightarrow A) : *}}{\vdash (\lambda A{:}\, *. \lambda x{:}\, A.x) : (\Pi\, A{:}*.\, A \rightarrow A)}$$

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

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[4]

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).

[4] 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 ($\langle *, \Box \rangle, \langle \Box, * \rangle$ e $\langle \Box, \Box \rangle$) identificam-se oito *Sistemas Abstractos de Tipos* diferentes, a que se dão nomes específicos (ver quadro).

Dependências

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

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: $id \doteq (\lambda A{:}\, *. \lambda x{:}\, A.x)$. Qual é o tipo de id ?

```
Coq < Check id.
```

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

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

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

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

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:

$$(id\,T\,t) \equiv (\lambda A\colon *.\lambda x\colon A.x)\,T\,t \to (\lambda x\colon T.x)\,t \to t$$

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 $(\lambda C\colon *.\lambda D\colon *.\lambda x\colon C.\lambda y\colon C \to D.yx)$. Qual o seu tipo ?

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

```
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 $(f\,T\,(P \to P)\,t)$

$$(f\,T\,(P \to P)\,t) \equiv (\lambda C\colon *.\lambda D\colon *.\lambda x\colon C.\lambda y\colon C \to D.yx)\,T\,(P \to P)\,t \to$$

$$\to (\lambda D\colon *.\lambda x\colon T.\lambda y\colon T \to D.yx)\,(P \to P)\,t \to (\lambda x\colon T.\lambda y\colon T \to P \to P.yx)\,t \to (\lambda y\colon T \to P \to P.yt)$$

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 4A. Abra uma secção e declare / defina em Coq

$$\begin{array}{l} A{:}\,* \\ B{:}\,* \\ exp \doteq \lambda p{:}\,(\Pi C{:}\,*.(A \to B \to C) \to C).p\,A\,(\lambda x{:}\,A.\lambda y{:}\,B.x) \end{array}$$

i) Quais as variáveis livres e ligadas de `exp` ?

ii) Deduza p tipo de `exp`. Use o Coq para verificar a sua resposta.

iii) Feche agora a secção que abriu. Qual o efeito do fecho desta secção.

iv) Qual o λ-termo definido agora em `exp` ?

v) Declare agora

$$\begin{array}{l} T{:}\,* \\ t{:}\,T \end{array}$$

Construa uma cadeia de redução maximal de raiz $(exp\;T\;T\;(\lambda C{:}\,*.\lambda x{:}\,T \to T \to C.x\,t\,t))$.

vi) Qual o tipo de $(exp\;T\;T\;(\lambda C{:}\,*.\lambda x{:}\,T \to T \to C.x\,t\,t))$?
Use o Coq para verificar a sua resposta.

Questão 4B. Abra uma secção e declare / defina em Coq

$$\begin{array}{l} A{:}\,* \\ B{:}\,* \\ ex \doteq \lambda p{:}\,B \to A.\lambda q{:}\,(\Pi C{:}\,*.A \to C).\lambda b{:}\,B.q\,A\,(p\,b) \end{array}$$

i) Quais as variáveis livres e ligadas de `exp` ?

ii) Deduza p tipo de `exp`. Use o Coq para verificar a sua resposta.

iii) Feche agora a secção que abriu. Qual o efeito do fecho desta secção.

iv) Qual o λ-termo definido agora em `exp` ?

v) Declare agora

$$\begin{array}{l} U{:}\,* \\ V{:}\,* \\ x{:}\,U \\ y{:}\,V \end{array}$$

Construa uma cadeia de redução maximal de raiz $(exp\;U\;V\;(\lambda y{:}\,V.x))$.

vi) Qual o tipo de $(exp\;U\;V\;(\lambda y{:}\,V.x))$?
Use o Coq para verificar a sua resposta.

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).
```

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 $\boldsymbol{\lambda\omega}$

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 $\lambda\omega$.

Exercício: Como definiria, recorrendo aos termos de que dispõe neste momento, a função polimórfica que recebe um par ordenado com ambas as componentes do mesmo tipo, e devolve esse par com as componentes invertidas? Qual o tipo de tal função?

(ex: $\langle 2, 4\rangle \mapsto \langle 4, 2\rangle$)

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,

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

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 $A_1 \rightarrow$

$A_2 \ldots \to A_i \ldots \to *$, com A_i tipos proposicionais. A seguinte instância da regra do produto, juntamente com o axioma $\Gamma \vdash * : \Box$, gera todos estes géneros:

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

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.[5]

A regra de formação de tipos-produto, que na sua instância de $\lambda_{\to}$ dava origem à formação dos tipos funcionais da forma $A \to B$, deve em λP ser devidamente reinterpretada. De facto, a razão pela qual em $\lambda_{\to}$ se podia escrever o tipo $\Pi x\colon A.B$ simplesmente como $A \to B$, 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 $P : A \to *$, o tipo-produto $\Pi x\colon A.(Px)$.

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.[6]

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:

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

[6] 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.

```
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)).
```

---

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.[7]

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`.[8]

```
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.
```

[7] 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.

[8] 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 $\lambda x\colon U.y$ 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 $\Pi A{:}*.\ \Pi B{:}*.\ A \to B \to A$, constitui uma prova da proposição $\forall A, B.\ A \to B \to A$, 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 $A \to B \to C$, 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 5A. Construa uma prova do seguinte teorema de segunda ordem:

$$\forall S, Q, R.\ S \to (Q \to S \to R) \to Q \to R$$

Indique o termo que codifica a sua prova.

Questão 5B. Construa uma prova do seguinte teorema de segunda ordem:

$$\forall S, Q, R.\ (S \to Q \to R) \to Q \to S \to R$$

Indique o termo que codifica a sua prova.

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.
```

Exercício: 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[9]) podem ser definidas da seguinte forma:

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

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.[10]

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

[10] 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.

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 *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.[11]

[11] Esta forma de se trabalhar com novas conectivas insere-se naquilo a que se costuma chamar Princípio dos **Juízos**

```
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 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.

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.

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.[12] 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

[12] 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 $S \to \bot$. 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 teorema

$$\neg(Q \vee R) \rightarrow (\neg Q \wedge \neg R)$$

Questão 6B. Construa em **Coq** uma prova para o teorema

$$(\neg Q \wedge \neg R) \rightarrow \neg(Q \vee R)$$

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

```
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 $A(y) \rightarrow B(y)$, 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 10A. Prove o teorema

$$(\exists x.\neg A(x)) \rightarrow \neg(\forall x.A(x))$$

Questão 10B. Prove o teorema

$$\neg(\exists x.A(x)) \rightarrow (\forall x.\neg A(x))$$

Exercício: Averigue se a frase $\forall x.\exists y.P(x,y) \rightarrow \exists y.\forall x.P(x,y)$ 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,[13] 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"

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

quando aplicar a regra RAA, e como obter o absurdo, justifica-se a apresentação da árvore de prova em Dedução Natural (mp e $\rightarrow$ são as regras de eliminação e introdução da implicação, e não se faz 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 $\neg A \rightarrow \bot$. A táctica `Cut` permite em geral substituir o objectivo V por $U \rightarrow V$ e U, mediante o comando `Cut U`:[14]

```
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

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

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 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.`

5.5 Algumas notas sobre a *igualdade* em Coq

A existência de um predicado universal[15] 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.

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

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

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

Este axioma 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`. Esta igualdade, conhecida por **Igualdade de Leibniz** é definid em Coq com recurso a um tipo indutivo[16]

O sistema Coq quando arranca carrega, por defeito, uma serie de definições, entre as quais a dos números naturais. Observe, atentamente o resultado dos seguintes comandos:

```
Coq < Check nat.
Coq < Check O.
Coq < Check S.

Coq < Check (S (S O)).
```

Vejamos mais uns exemplos. Seja `f` uma função sobre naturais que aplicada a zero dá como resultado zero:

[16] De facto, toda a concepção do sistema Coq se encontra intimamente ligada aos tipos indutivos.

```
Coq < Variable f : nat -> nat.
Coq < Hypothesis foo : (f O) = O.
```

Provemos então que $\forall n.\ n = O \rightarrow f(n) = n$

```
Coq < Lemma L1 : (n:nat) n=O -> (f n)=n.
L1 < Intros.
L1 < Rewrite H.
L1 < Apply foo.
L1 < Qed.
```

Provemos ainda que $f(f(O)) = O$

```
Coq < Lemma L2 : (f (f O))=O.
L2 < Rewrite foo.
L2 < Rewrite foo.
L2 < Reflexivity.
```

Em seguida, apresentamos mais duas maneiras alternativas de de provar este mesmo lema:

```
L2 < Restart.
L2 < Rewrite foo.
L2 < Apply foo.

L2 < Restart.
L2 < Replace (f O) with O.
L2 < Apply foo.
L2 < Symmetry.
L2 < Apply foo.

L2 < Save.
```

Referências

[1] José Manuel Valença, *Sebenta Teórica de Elementos Lógicos da Programação II* (Manuscrito). Universidade do Minho, Departamento de Informática, 1997/98.

[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.

Elementos Lógicos da Programação II

Exame Prático (LMCC 2. 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. Considere o combinador W do λ-calculus sem tipos:

$$W \doteq \lambda x\, y\, z\, .\, x\, z\, (z\, y)\, y$$

1.1 Qual deverá ser o tipo mais genérico das variáveis de W para que este combinador deja válido no λ-calculus com tipos. Defina o combinador W em Coq e indique o seu tipo.

1.2 O combinador W que acabou de definir é polimórfico ? Justifique a sua resposta.

2. Abra uma secção de nome, `questao2`, e construa um contexto adquado à definição das expressões K e M.

$$K \doteq \lambda Y\colon *.\lambda X\colon *.\lambda x\colon X.\lambda y\colon Y.\, x$$
$$M \doteq \lambda h\colon (\Pi C\colon *.C \to U \to C).\lambda y\colon U.\lambda x\colon U \to V \to U.\, x\; y\; (h\; V\; a\; y)$$

2.1 Defina as expressões-λ K e M.

2.2 Quais a s variáveis livres e ligadas de K e de M ?

2.3 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 $\Gamma_2 \vdash M : T_2$.

2.4 Calcule a forma normal da seguinte expressão-λ

$$(\lambda u\colon U.\, M\; (K\; U)\; u)$$

Qual o seu tipo ? Use o Coq para verificar as suas respostas.

2.5 Feche agora a secção `questao2`. Qual o efeito do fecho desta secção ?

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

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

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

4. Sejam A, B e C proposições. Partindo das hipóteses de que $A \to B$ e $\neg(A \to C)$ são fórmulas válidas, prove que $A \to (B \land \neg C)$.

5. Prove o seguinte teorema da lógica de predicados:

$$R \to (\exists x.\forall y.\; (P(x,y) \land R) \to Q(x,y)) \to (\forall y.\exists x.P(x,y) \to Q(x,y))$$