3 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]

3.1 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 $\Pi A{:}\,*.\ \Pi B{:}\,*.\ A \to B \to A$, 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

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

3.2 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:

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

3.3 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:

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

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]

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

[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.

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.[10]

[10] 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).