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

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

1 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 $(\lambda x\colon \sigma.xx)$ e $(\lambda x\colon \sigma.x)(\lambda x\colon \sigma.x)$.[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:

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

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 $(\lambda x{:}\, P.(\lambda y{:}\, P.\lambda z{:}\, P \to P.zy)x)p(\lambda x{:}\, P.x)$ 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

$$(\lambda x{:}\, T \to T \to T.\lambda y{:}\, T \to T.\lambda z{:}\, T.xz(yz))(\lambda a{:}\, T.\lambda b{:}\, T.a)(\lambda x{:}\, T.x)t$$

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.

$$comp \doteq \lambda f{:}\, B \to C.\lambda g{:}\, A \to B.\lambda x{:}\, A.f(gx)$$

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.

$$fun \doteq \lambda g{:}\, A \to B \to C.\lambda h{:}\, (B \to C) \to A \to B.\lambda x{:}\, A.h(gx)x.$$

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:

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

```
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

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

Eager Evaluation

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

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

Podemos conferir estes resultados com o sistema Coq:

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

Se quiser sair do Coq, faça:

```
Coq < Quit.
```