5 Introdução aos Tipos Indutivos – os números naturais

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

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

5.1 Motivação – Os números naturais e os axiomas de Peano

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Algumas notas sobre a *igualdade* em Coq

A existência de um predicado universal[13] de igualdade em **Coq** leva-nos a reflectir um pouco sobre quais devem ser as propriedades de tal predicado. Além de ter de se tratar obrigatoriamente de uma relação de equivalência (reflexiva, simétrica e transitiva), tem de verificar ainda duas propriedades:

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

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

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

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

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

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

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

Como ilustração deste facto, demonstremos a propriedade de transitividade:

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

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

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

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

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

5.3 Algumas provas sobre os números naturais, utilizando reescrita

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

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

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

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

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

```
Coq < Print Hint.
```

Continuemos então a nossa prova:

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

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

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

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

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

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

Provemos agora o simétrico do segundo axioma da soma:

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

Questão 8. ...

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

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

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

5.4 Tipos Indutivos em Coq

A existência, em algumas linguagens de programação, de primitivas específicas para a definição de novos tipos indutivos generaliza e automatiza os princípios referidos relativamente aos números naturais. Assim, quando numa linguagem de programação se cria um novo tipo indutivo, associando-se-lhe um nome e um conjunto de construtores,[14] tem-se em geral as seguintes garantias:

1. Os construtores são ortogonais;

2. Os construtores são injectivos;

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

Vejamos então como declarar o tipo indutivo dos números naturais:

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

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

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

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

```
Coq < Check nati_ind.
```

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

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

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

De novo foi necessário ajudar na unificação de ordem superior. O **Coq** disponibiliza uma táctica própria que facilita um pouco esta tarefa: a indução pode ser invocada a partir da variável de tipo `nati` que se encontra declarada no contexto actual e sobre a qual se pretende efectuar o raciocínio indutivo. A dita táctica chama-se `Elim`[15]:

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

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

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

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

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

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

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

5.5 Recursão Primitiva

Segue-se a definição das funções de soma e produto de números naturais. Em tipos indutivos, a definição de tais funções é feita por utilização de um construtor especial, dito o **operador de recursão primitiva** do tipo, cuja criação em **Coq** é automática aquando da declaração de cada tipo, sendo o seu nome igual ao do tipo, acrescentado do sufixo `_rec`:[16]

```
Coq < Check nati_rec.
```

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

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

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

Observe os seguintes resultados.

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

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

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

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

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

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

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

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

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

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

Questão 9. ...

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

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

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

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

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

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

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

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

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

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