6 Tipos Indutivos – exemplos

6.1 Pares Ordenados

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

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

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

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

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

O tipo indutivo proposicional `Pair` tem portanto um único construtor. Esta definição assegura por um lado a impossibilidade de existência de elementos de tipo `Pair` que não sejam construídos pelo construtor `mkPair`, e por outro a injectividade deste. Permite ainda a definição de funções por análise de casos[17] / concordância de padrões, como se exemplifica com as funções destrutoras:

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

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

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

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

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

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

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

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

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

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

São agora demonstráveis trivialmente as igualdades que de outra forma teriam de ser impostas artificialmente sobre este tipo. Vejamos por exemplo como provar $(mkPair\ (proj1\ p)\ (proj2\ p)) = p$

:

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

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

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

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

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

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

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

Questão 10. ...

6.2 Listas Polimórficas

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

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

Comecemos por provar um lema simples mas útil:

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

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

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

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

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

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

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

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

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

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

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

Questão 11. ...

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.3 Predicados como definições indutivas

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

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

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

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

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

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

Vejamos então como construir a mesma prova de há pouco:

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

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

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

```
membPcat1 < Check memberP_ind.
```

Justifica-se a sua reprodução em notação tradicional:

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

Observando atentamente a última linha desta definição, conclui-se que este princípio de indução pode ser utilizado para mostrar, a partir de uma hipótese da forma $(memberP\ l\ a)$, qualquer frase $(P\ l\ a)$, com P um predicado de tipo $P : List \to A \to Prop$. Quando o objectivo for desta forma[18] e existir uma hipótese como a referida, a táctica `Elim`, invocada com a dita hipótese, terá como efeito a substituição do objectivo por dois outros, correspondentes às segunda e terceira linhas acima reproduzidas.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
Coq < End SetA.
```

6.4 Ainda as conectivas lógicas e a igualdade em Coq

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

Analisamos atentamente os seguintes resultados:

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

Cada definição possui tantos construtores quantas as formas possíveis de se provar um teorema construído com a respectiva conectiva lógica:[19] duas para a disjunção, uma para a conjunção, e nenhuma para o absurdo. O tipo de cada construtor não suscita quasquer dúvidas.

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

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

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

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

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

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

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

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

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

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

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