7 Introdução à verificação e síntese de programas em Coq

Nesta sessão de trabalho introduzimos algumas facilidades do sistema **Coq** de apoio ao desenvolvimento de *programas certificados*. Em particular, efectuaremos dois tipos de exercício com interesse: a extracção de programas a partir de provas das suas especificações, e a construção automática de provas de correcção de programas face às suas especificações.

Como caso de estudo, continuaremos a utilizar o tipo indutivo das Listas Polimórficas, na sequência da sessão de trabalho anterior. Vamos estudar uma função muito simples, a *concatenação* de listas, para a qual obtivemos uma possível realização na sessão anterior.

O seguinte prelúdio é suficiente para nos permitir trabalhar nesta sessão:

```
Coq < Require Natural.
Coq < Require Extraction.
Coq <
Coq < Section SetA.
Coq < Variable A:Set.
Coq < Inductive List : Set :=
Coq <           nil : List
Coq <         | cons : A -> List -> List.
Coq < Token "::".
Coq < Infix 1 "::" cons.
```

7.1 Extracção de Programas

Comecemos por escrever uma especificação da função, através de um predicado definido indutivamente. Este predicado pode ser visto como um programa lógico que realiza a função:

```
Coq < Inductive catP : List -> List -> List -> Prop :=
Coq <       catP_nil : (l2:List) (catP nil l2 l2)
Coq <    | catP_cons : (l,l2,lt:List)(x:A) (catP l l2 lt) ->
Coq <                  (catP (cons x l) l2 (cons x lt)).
Coq <
Coq < Hint catP_nil catP_cons.
```

A definição deste tipo indutivo constitui de facto uma especificação do comportamento de uma função de concatenação de listas. É demonstrável que para cada par de listas `l1` e `l2`, existe uma outra que é a sua concatenação, como passamos a verificar:

```
Coq < Theorem proofCat : (l1,l2:List) (Ex [l:List] (catP l1 l2 l)).
proofCat <
proofCat < Induction l1; Clear  l1; Intros.
proofCat < Exists  l2.
proofCat < Auto.
proofCat < Elim (H l2); Clear  H.
proofCat < Intros x H.
proofCat < Exists  (cons a x).
proofCat < Elim H; Clear  H.
proofCat < Auto.
proofCat < Auto.
proofCat < Save.
```

A prova envolve indução na lista `l1` e na própria definição de `catP`, e garante que qualquer processo de se calcular `l3` a partir de `l1` e `l2` de acordo com `catP` define uma função total. Observe-se a estrutura indutiva da prova:

```
Coq < Print Natural proofCat.
```

Esta prova contém, ela própria, informação de carácter computacional, descrevendo *como* é obtida a concatenação de `l1` com `l2`. De facto, tratando-se de uma prova intuicionista (como em toda a teoria abstracta de tipos) de uma quantificação existencial, essa prova apresenta obrigatoriamente um *testemunho* dessa quantificação (um elemento que satisfaz o predicado quantificado). Esse testemunho é `l3`.

Quer isto dizer que, ao construir-se interactivamente uma prova de que, para qualquer par de listas, existe uma outra que é a sua concatenação, essa prova contém obrigatoriamente o *processo* de construção dessa lista. De acordo com a própria definição indutiva de `catP`, esse processo avança naturalmente por análise de casos no primeiro argumento da concatenação, fornecendo um testemunho diferente para cado caso.

Desejamos então extraír desta prova a sua componente informativa (em `Set`), esquecendo a componente lógica (em `Prop`). Para isso comecemos por repetir a prova acima, mas desta feita considerando uma interpretação como `Set` para `*`, em vez de `Prop`. Note-se que em termos formais encontrar uma prova para um teorema (de tipo `Prop`) corresponde a encontrar um termo do tipo (de tipo `Set`) correspondente a esse teorema.

Observe-se a seguinte sintaxe, correspondente à quantificação existencial em `Set`, e repita-se então a construção da prova, por forma a construir um termo do tipo indicado:[20]

[20] Um estudo aprofundado das propriedades e utilizações possíveis destes "tipos existenciais" sai fora do âmbito desta sessão de trabalho.

```
Coq < Check (l1,l2:List) { l:List |  (catP l1 l2 l) }.
Coq < Theorem progCat : (l1,l2:List) { l:List |  (catP l1 l2 l) }.
progCat <  ............
progCat < Save.
```

O **Coq** fornece um processo pelo qual o conteúdo computacional de uma prova pode ser extraído e traduzido para uma de várias linguagens de programação possíveis.[21] O termo extraído de uma prova com conteúdo informativo (computacional) é obtido pelo **Coq** 'esquecendo' a parte lógica da prova, e pode ser visualizado (bem como o seu tipo) com o comando `Extraction`. O termo extraído é sempre um termo de λ_ω com definições indutivas, ou seja, um termo sem a presença de tipos dependentes de termos.

Observemos então o termo extraído, e veja-se como o seu tipo é, como seria de esperar, $List \to List \to List$, correspondente ao tipo da função de concatenação de listas:

```
Coq < Extraction progCat.
```

Para traduzir o termo extraído para, por exemplo, **Caml**, basta fazer:

```
Coq < Write Caml File "cat" [progCat].
```

A possibilidade de ocorrência do erro obtido (impossibilidade de atribuir um tipo em ML à função `cat`) é compreensível se tivermos em conta que o sistema de tipos desta linguagem é mais restrito do que o de λ_ω. No entanto, trata-se aqui de um caso simples de resolver: basta fechar a secção corrente para tornar realmente polimórficas as funções que temos escrito, e tentar de novo:

```
Coq < Save State s1.
Coq < End SetA.
Coq < Write Caml File "cat" [progCat].
Coq < Restore State s1.
```

Este comando terá como efeito a criação de um ficheiro `cat.ml`, contendo uma função `progCat` em Caml, que realiza a concatenação de duas listas. O ficheiro contém ainda uma tradução do tipo indutivo das listas polimórficas, necessário para a definição da função.[22]

A utilização do estado `s1` acima justifica-se apenas pelo facto de desejarmos continuar a trabalhar com listas de um tipo `A`, sem toda a sintaxe adicional associada às listas polimórficas.

[21] Nomeadamente, vários dialectos de **ML** ou **Gofer**.

[22] Não são portanto utilizados por este *package* os tipos pré-definidos do Caml, como sejam as listas polimórficas.

7.2 Verificação de Programas

Relembremos agora a versão da função `cat` que havíamos definido na aula anterior:

```
Coq < Fixpoint cat [l1:List] : List -> List :=
Coq <     [l2:List]
Coq <         Cases l1 of
Coq <                nil => l2
Coq <       | (cons h t) => (cons  h (cat t l2))
Coq < end.
```

Se esquecermos o que fizemos nesta sessão até este ponto – a síntese do programa **ML** que implementa esta função – podemo-nos questionar agora sobre a *correcção* da função `cat` em relação à especificação inicial, expressa na definição indutiva `catP`. Este problema, da *Verificação de Programas*, é formalmente distinto do anterior – não temos agora o esforço criativo de realizar um programa que implemente uma especificação; o programa é dado e queremos apenas verificar que ele implementa correctamente a especificação.

Tratando-se de um problema mais simples, espera-se que a sua resolução seja mais automatizável em **Coq**. De facto é-o, devido à existência de tácticas especiais que automatizam as provas de correcção. Por prova de correcção entende-se, aqui, uma prova de que o termo em questão é uma função total que obedece à especificação dada pela definição indutiva de `catP`.

Retomemos então a prova de que existe sempre a concatenação de duas listas, de acordo com a especificação expressa pelo tipo `catP`:

```
Coq < Theorem proofCatImpl : (l1,l2:List) { l:List |  (catP l1 l2 l) }.
```

Associemos agora a esta prova a função que supostamente implementa a especificação:

```
proofCatImpl < Realizer cat.
```

e finalmente recorramos à táctica especial `Program` para efectuar esta prova.

```
Coq < Program.
Coq < Auto.
```

O efeito da táctica `Program` é o de introduzir no contexto a declaração de um termo `cat` construído a partir da função `cat`, e cujo conteúdo informativo é uma função com o tipo de `cat`.

No caso corrente, esta declaração introduzida por `Program` tem exactamente o tipo correspondente ao objectivo da prova, pelo que esta se conclui trivialmente. Noutras circunstâncias, a táctica `Program_all` pode-se revelar mais útil, uma vez que repete a aplicação de `Program` e `Auto` enquanto possível.

A conclusão com sucesso desta prova pela táctica `Program` indica que o termo `cat` realiza correctamente a especificação `catP`, pelo que se pode afirmar que aquele termo constitui um programa *certificado*.

Exercício: Aplique as metodologias estudadas neste guião (Extracção e Verificação) à função que calcula o comprimento de uma lista. Apresente de forma devidamente comentada todos os passos que efectuar.