Verificação de Programas

Nelma Moreira

Departamento de Ciência de Computadores da FCUP

Coq: demonstrador interactivo baseado no pCiC
Aula 22

COQ

COQ é um assistente de demonstração interactivo que permite:

- verificação de demonstrações de teoremas.
- a especificação e a obtenção de programas certificados (que satisfaçam essa especificação).
- desenvolvimento de demonstrações matemáticas numa lógica de ordem superior
- ambiente de desenvolvimento de lógicas diversas: temporais, modais, de descrição, etc.

COQ

É baseado num λ-calculus tipificado polimórfico com tipos dependentes e noção primitiva de tipos indutivos: o Cálculo de Construções Indutivas (CoC/pCiC).

Baseia-se no isomorfismo de Curry-Howard:

- as demonstrações são objectos
- as proposições (especificações) são tipos
- uma demonstração de uma proposição P é um objecto de tipo P

É constituído por:

- Um núcleo de verificação de tipos e construção de contextos bem tipificados
- Uma linguagem (funcional) de especificação (Gallina).
- Ferramentas de ajuda à construção interactiva de demonstrações: *Tácticas*

Lean

É outro demonstrador interactivo também baseado no CoC/CiC
`https://leanprover-community.github.io/index.html`

Como executar

- WWW: `coq.inria.fr`
- Linha de comando: `coqtop`
- Compilador: `coqc`
- Ambiente gráfico: `coqide`
- Emacs: `proofgeneral` módulo para vários demonstradores interactivos

Sintaxe: termos e tipos

Não há distinção sintáctica entre termos e tipos. Mas todos os termos têm tipo: mesmo os tipos.

A sintaxe do CiC é:

$$
\begin{array}{rcl}
s & := & \mathit{Set} \mid \mathit{Prop} \mid \mathit{Type} \\
t & := & s \mid x \mid c \mid C \mid I \mid \\
 & & \forall x : t.t \mid \lambda x : t.t \mid tt \mid \mathit{case}(t, \ldots, t) \\
 & & \mathtt{fix}\, x\, x : t := t, \ldots, x : t := t
\end{array}
$$

onde x são variáveis, c constantes, C construtores e I tipos indutivos.

Esta sintaxe tem muita coisa nova, basta para já considerar o subconjunto de $\lambda - \rightarrow$ (sistema de tipos simples).

Expressões e Tipos

Para ter acesso às bibliotecas (standard): (`init.v`)

`Require Import`

Por exemplo: `Arith`, `ArithRing`, `Lists`

Verificação de tipos:

```
Check t.
```

Verifica o tipo de t no ambiente actual.

```
Check 3.
3: nat
Check plus.
plus  : nat -> nat -> nat
Check (nat ->(nat ->nat)).
nat -> nat -> nat: Set
Check (3=4).
3=4: Prop
```

Ambientes e Contextos

São caracterizados por declarações e definições.

Declarações Associa um tipo a um identificador

`Variable x:nat.`

Definições Atribui um valor a um identificador

`Definition ex1:  fun x => x+ 3.`

`Theorem impdist:  (P->Q->R)->(P->Q)->(P->R).`

`Print e.` Retorna o valor associado

Contextos Âmbito local de definições e declarações (Γ)

`Section s.  ...  End s.` s um âmbito local

Ambientes Âmbito global de definições e declarações (E)

Notações

`Scope` Permite interpretar notações ($*$, $+$, etc)

`Open Scope nat.`

`Locate n.` indica os âmbitos dessa notação.

Inferência de tipos

$E, \Gamma \vdash t : A$ atribuição de tipos

Inferência de tipos para identificadores

$$\frac{(x : A) \in E, \Gamma}{E, \Gamma \vdash x : A}\ (Var)$$

```
Check plus.
plus: nat -> nat -> nat
```

Inferência de tipos para aplicação

$$\frac{E, \Gamma \vdash e_1 : A \to B \qquad E, \Gamma \vdash e_2 : A}{E, \Gamma \vdash e_1\, e_2 : B}\ (APP)$$

```
Check plus 2 3.
2 + 3: nat
```

Inferência de tipos

Uma abstração $\lambda x : A.e$ é representada por `fun x:A => e`

Inferência de tipos para abstração

$$\frac{E, \Gamma :: (x : A) \vdash e : B}{E, \Gamma \vdash \texttt{fun}\, x : A \Rightarrow e : A \to B}\ (ABS)$$

Inferência de tipos para `let`

`let` $v := t_1$ `in` t_2 corresponde a $t_2[v/t_1]$)

$$\frac{E, \Gamma \vdash t_1 : A \qquad E, \Gamma :: (v := t_1 : A) \vdash t_2 : B}{E, \Gamma \vdash \texttt{let}\, v := t_1\, \texttt{in}\, t_2 : B}\ (Let)$$

Computações –Eval

O COQ não é um ambiente para a execução de programas, mas é possível efectuar computações....que podem ser usadas nas tácticas...

Uma computação é obtida por reduções. Para além da redução β existem outras...

Estratégias Por valor (`cbv`) ou `lazy`

Regras de Redução

- Redução-δ substituir um identificador pela definição
- Redução-β $(\texttt{fun}\, x : A => e_1)e_2 \rightarrow e_1[x/e_2]$
- Redução-ζ aplica o `let`
- Redução-ι para tipos indutivos

```
Definition mul :=  fun n: nat -> n*n.

Eval cbv delta beta [mul] in (mul 5).
Eval compute  in (mul 5).
```

Tipos e Espécies (`sorts`)

O tipo de um tipo denomina-se **espécie** (sort).

`Set`

Os termos cujo tipo é `Set` dizem-se especificações.
Cada termo cujo tipo é uma especificação é um programa.
Se $A : Set$ e $B : Set$ então $A \to B : Set$

`Type`

O tipo do `Set` é `Type`. Embora na CiC exista uma hierarquia de tipos `Type` no COQ apenas se considera um.

`Prop`

O tipo para as proposições é `Prop`. Cada termo cujo tipo é uma proposição é uma demonstração. O tipo de `Prop` é `Type`
Se $A : Prop$ e $B : Prop$ então $A \to B : Prop$

A diferença entre um tipo `Set` e um `Prop` tem haver com a informação que é guardada ... a primeira é para programas e a segunda para proposições.

IPC($\rightarrow$) em COQ

Vamos considerar o fragmento implicacional da lógica proposicional intuicionista em COQ (`minimal.v`).

As variáveis proposicionais são de tipo `Prop`.

```
Section Minimal.
Variables P Q R T:Prop.
Check P.
```

Podia-se usar `Hypothesis` em vez de `Variable`.

No âmbito global pode-se usar `Axiom` ou `Parameter`.

`Theorem` Definição global de identificadores de tipo `Prop.` (ou `Lemma`).

```
Theorem imp_trans: (P->Q)-> (Q->R)->P->R.
```

Exemp.

Ver

Demonstrações dirigidas por objectivos

Objectivo P: $E, \Gamma \overset{?}{\vdash} P$

Tácticas: Comandos que aplicados a um objectivo, produzem uma lista possivelmente vazia de sub-objectivos.

Como se faz uma demonstração?

`Proof`

- Procura de habitantes em tipos.
- Técnica recursiva:
  - tomar um tipo no contexto
  - procurar a forma dum termo com buracos que terá esse tipo
  - recursivamente construir o termo preenchendo os buracos
- Para preencher os buracos usam-se *tácticas*: as de **introdução** criam um objectivo cuja estrutura é gerada por um construtor; de **eliminação** permitem usar factos cuja estrutura é dada por um construtor.
- Quando não há buracos, a demonstração está terminada.

Tácticas: `assumption` e `intro`

`exact x.` ou `assumption.` correspondem à regra (VAR) ou Axioma.

A táctica `intro` introduz novas hipóteses no contexto. Corresponde à aplicação da regra (ABS) (ou introdução da implicação $\rightarrow I$).

Sendo o objectivo $E, \Gamma \stackrel{?}{\vdash} P \rightarrow Q$

`intro H.`

Gera um sub-objectivo da forma $E, \Gamma :: (H : P) \stackrel{?}{\vdash} Q$.

Se t é um termo de tipo Q então `fun H : P` $\Rightarrow$ `t` é o resultado do objectivo inicial.

Variantes: `intro` v. ou `intros` $v_1 \ldots v_n$ ou `intros.` ou `intro.`

Tácticas: `apply`

Corresponde à aplicação da regra (APP) (ou eliminação da implicação $\rightarrow E$ (*Modus Ponens*)).

Tipos cabeça e final

Se t tem tipo $P_1 \rightarrow \cdots \rightarrow P_n \rightarrow Q$ então $P_k \rightarrow \cdots \rightarrow P_n \rightarrow Q$ e Q são tipos da cabeça de nível k de t. Se Q não é uma implicação então é o tipo final.

Se t é um termo de tipo $P_1 \rightarrow \cdots \rightarrow P_n \rightarrow Q$ e o objectivo é $P_k \rightarrow \cdots \rightarrow P_n \rightarrow Q$

então `apply` t. gera k-1 objectivos $P_1, \ldots, P_{k-1}$.

Se esses objectivos sucedem com termos t_1,...,t_{k-1}, então o objectivo inicial sucede com o termo $t\ t_1\ t_2 \cdots t_{k-1}$.

```
Theorem apply_ex: (Q-> R -> T) -> (P-> Q) -> P -> R -> T.
```

Estrutura das demonstrações

`Theorem x:P.` ou `Goal P.`

Mostrar os objectivos correntes

`Show.` ou `Show i.`

`Show Proof.` (termo gerado)

Refazer (*n*) passos

`Undo.` ou `Undo n.`

Interromper ou Reiniciar

`Abort.` ou `Restart.`

Fim

`Qed.` ou `Save id.` (se `Goal`).

Tentar demonstrar `((P -> Q) -> P) -> P`.

Teoremas *versus* Definições

Os Teoremas são de tipo `Prop` e as definições de tipo `Set`. As mesmas tácticas podem ser usadas em ambos!.

`Theorem nome:T. Proof t.`

`Definition nome:T. := t.`

A diferença essencial é a `Opacidade` e `Transparencia`:

- nas demonstrações de teoremas $x := t : T$ há certos pormenores da construção que não são relevantes (pode ser `opaca`).
- numa definição como corresponde um programa, todos os pormenores interessam para se poder extrair o programa...

Do mesmo modo podemos fazer demonstrações só usando `Section.` e `Hypothesis.` em vez da táctica `intro`.

Composição de Tácticas

Composição $tac_1; tac_2$ aplica tac_1 ao objectivo g e tac_2 a **todos** os subjectivos que a primera gera.

Se alguma falhar, falha todo o processo.

Generatização $tac_1; [tac_2 | \ldots | tac_n]$ supõe que a primeira táctica gera n subobjectivos e ao i-ésimo é aplicada tac_i

Orelse $tac || tac'$ Se tac falha, aplica tac'. Se esta falha, falha tudo.

`idtac` é uma táctica que sucede sempre

`fail` é uma táctica que falha sempre

`try try` tac é equivalente a $tac || $`idtac`

Mais tácticas

`cut` se o objectivo é $\Gamma \stackrel{?}{\vdash} P$ e é possível obter uma demonstração de $Q \to P$ e Q então podemos usar a táctica `cut`Q

`assert` Se achamos que Q ajuda a demonstrar o objectivo P podemos fazer `assert Q`, demonstrar Q e depois usar como lema para demonstrar Q.

`auto` Se o COQ sabe como demonstrar o objectivo (isto é se há um algoritmo de decisão)

`trivial` Como o anterior mas para casos mais simples.