UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

APPUNTI DI "Ragionamento automatico"

Laurea Magistrale in *Ingegneria e scienze informatiche*

A.A. 2020 – 2021

Creato da *Davide Zampieri*

Introduzione.

Il modello di programmazione del sistema Coq è il modello funzionale, a cui appartengono linguaggi come: Lisp, Scheme, ML, Haskell, ecc. Il linguaggio di programmazione di Coq è fortemente tipato, ossia ogni espressione ha un tipo. Il legame fra un'espressione ed il suo tipo è rappresentato con il simbolo `:`, per esempio `O : T` indica che l'oggetto O ha il tipo T. Fra questi oggetti, ci sono gli oggetti di base: booleani, numeri naturali, stringhe.

```
true:  bool
   1:  nat
"hello": string
```

Il linguaggio di Coq è funzionale. Ciò vuol dire che le funzioni sono oggetti di prima classe, ovvero le funzioni si possono manipolare come un qualsiasi altro oggetto (ad esempio i numeri). Il simbolo `->` viene utilizzato per rappresentare il tipo delle funzioni. Per esempio, una funzione `f` che prende un numero naturale e restituisce un booleano si rappresenta come:

```
f: nat -> bool
```

Isomorfismo di Curry-Howard.

Programmare è la stessa cosa che provare. L'isomorfismo di Curry-Howard, infatti, dà una corrispondenza fra programma e prova e fra tipo e formula; permette cioè di dare un'equivalenza fra il fatto che esiste un programma *P* che ha il tipo *T* e il fatto che esiste una prova *P'* della formula *T'*.

Polimorfismo.

Una funzione può avere un tipo generico usando il cosiddetto polimorfismo. Ciò significa che la funzione si potrà applicare a tutti i tipi di oggetto. La presenza del polimorfismo viene resa esplicita mettendo come prefisso al tipo il carattere `'` (in ML) o il carattere `?` (in Coq).

Esempio.

Secondo l'isomorfismo di Curry-Howard, dare una prova di

```
A ⇒ ((A ⇒ B) ⇒ B)
```

è equivalente a trovare un programma che ha tipo

```
'a -> (('a -> 'b) -> 'b)
```

Definizioni.

Spiegheremo come si possono definire nuove costanti e nuove funzioni nel mondo della programmazione e, simmetricamente, nuovi predicati e nuove relazioni nel mondo della prova.

Proposizioni e insiemi.

Coq è un dimostratore basato sull'isomorfismo di Curry-Howard che usa un linguaggio funzionale tipato chiamato Calcolo delle Costruzioni Induttive per definire calcolo e prova. Il fatto che dentro Coq si possa calcolare e provare si nota dalla presenza dei due tipi `Prop` e `Set`. Il primo corrisponde al tipo delle proposizioni, dunque appartiene al mondo delle prove. Il secondo corrisponde agli oggetti sui quali possiamo calcolare. Per esempio abbiamo:

```
False: Prop
  nat: Set
```

Essere in un linguaggio tipato implica che anche i tipi `Prop` e `Set` devono avere un tipo (o meglio una sorte), ovvero `Type`. Il motivo della scelta di `Type` si deve al fatto che quest'ultimo ha come tipo sé stesso. Per sapere il tipo di un oggetto dentro Coq si usa il comando `Check`. Ogni comando in Coq comincia con una lettera maiuscola e termina con un punto.

Funzioni.

Il costruttore di base di Coq è la funzione; si scrive come `fun x:T => P`. Il suo tipo è `forall x: T, C` dove C è il tipo di P. Se il tipo non è dipendente, ovvero se x non compare dentro C, una variante del tipo è `T -> C`. La notazione matematica usuale di `fun x: T => P: forall x: T, C` è il lambda termine `λx. P: ∀x: T. C`.

Parametri.

Il comando per introdurre un parametro è `Parameter` <NAME>: <TYPE>. Esempi:

`Parameter A: Set.`	A è un insieme di oggetti.
`Parameter B: Prop.`	B è una proposizione.
`Parameter a: A.`	a è un elemento di A.
`Parameter b: B.`	b è una prova di B.
`Parameter f: A -> A.`	f è una funzione da A in A.
`Parameter g: B -> B.`	g è una prova di B ⇒ B.
`Parameter P: A -> Prop.`	P è un predicato, ossia una proprietà degli elementi di A.
`Parameter Q: A -> A -> Prop.`	Q è un predicato, ossia una relazione fra due elementi di A.
`Parameter id: forall A: Set, A -> A.`	`id` è una funzione che prende un insieme A e un elemento di questo insieme e restituisce un elemento di A.
`Parameter Id: forall A: Prop, A -> A.`	`Id` è una prova della formula `∀A: Prop.` A ⇒ A.

Connettivi logici.

- La congiunzione è definita in Coq con la funzione `and`. Ma `and A B` si può scrivere come `A /\ B`.
- La disgiunzione è definita in Coq con la funzione `or`. Ma `or A B` si può scrivere come `A \/ B`.
- La negazione è definita in Coq con la funzione `not`. Ma `not A` si può scrivere come ~A.
- La costante vera è definita in Coq con l'oggetto `True`.
- La costante falsa è definita in Coq con l'oggetto `False`.
- Il quantificatore esistenziale è definito in Coq con la funzione `ex`. Ma `ex T P` si può scrivere come `exists x: T, (P T)`.

Definizione di nuovi oggetti.

I parametri permettono di assumere l'esistenza di oggetti. In Coq si possono ovviamente definire nuovi oggetti in termini di altri oggetti con il comando `Definition` <NAME>: <TYPE> := <EXP>.

Costruzioni induttive.

Spiegheremo come definire nuove strutture dati e come manipolarle. Una struttura dati è definita tramite un elenco degli oggetti che la compongono. Ogni elemento in questo elenco rappresenta un costruttore, ovvero un modo di costruire un oggetto. Questo è importante perché ci permetterà di fare delle discussioni (analisi) sulla natura di un oggetto con la costruzione `match ... with ... end`. Per esempio, questo è il meccanismo di base che permette di far cambiare il comportamento di una definizione in funzione dei valori dei suoi argomenti.

Oggetti induttivi.

In Coq i nuovi oggetti induttivi si definiscono usando il comando:

```
Inductive <NAME>: <TYPE> :=
  <NAME>: <TYPE>
| <NAME>: <TYPE>
...
| <NAME>: <TYPE>
.
```

L'esempio tipico di oggetti induttivi è dato dai numeri naturali. Una rappresentazione possibile è la seguente:

```
Inductive nat: Set :=
  O: nat
| S: nat -> nat.
```

Con tale rappresentazione, 0 è rappresentato come `O`, 1 è rappresentato come `S O`, 2 è rappresentato come `S (S O)` e così via. Un altro esempio è rappresentato dai booleani:

```
Inductive bool: Set :=
  true : bool
| false: bool.
```

Il comando `Inductive` permette non solo di definire nuovi tipi ma anche predicati, dando le sue proprietà caratteristiche. Per esempio, la definizione della parità di un numero può essere data induttivamente come:

```
Inductive Even: nat -> Prop :=
  Even0 : (Even O)
| EvenSS: forall n: nat, (Even n) -> (Even (S (S n))).
```

Costruzioni per casi.

Il comando `match` permette di fare una discussione sul valore di un oggetto induttivo:

```
match <EXP> with
  <FILTER> => <EXP>
| <FILTER> => <EXP>
...
| <FILTER> => <EXP>
end
```

Ad esempio, il seguente predicato indica se un numero è nullo:

```
Definition is_zero: nat -> Prop :=
  fun a: nat =>
    match a with
      O    => True
    | (S _) => False
    end.
```

Ancora, la seguente funzione calcola il complementare di un booleano:

```
Definition notb: bool -> bool :=
  fun a: nat =>
    match a with
      true  => false
    | false => true
    end.
```

Funzioni ricorsive.

In Coq si possono definire funzioni ricorsive, ma bisogna fare attenzione nel definire tali funzioni. Infatti, aggiungere funzioni ricorsive significa introdurre computazioni possibilmente infinite. Dunque, in Coq si possono definire funzioni ricorsive, ma il sistema deve assicurarsi la terminazione di tutte le chiamate possibili della funzione. Per ottenere tale assicurazione si usa un criterio sintattico, ossia quando si definisce una funzione si specifica un argomento particolare della funzione tale che, ad ogni chiamata ricorsiva, il valore di questo argomento diminuisce strutturalmente. Il comando per definire una funzione ricorsiva sarà quindi:

```
Fixpoint <NAME> (<NAME>: <T>) ... (<NAME>: <T>) {struct <NAME>}: <T> :=
   <EXP>.
```

La convenzione prevede che il nome dopo `struct` indichi l'argomento che assicura la terminazione. Infine, per avere la computazione di una funzione dentro Coq, si può usare il comando `Compute`.

Confronto tra definizioni induttive e funzioni ricorsive.

Precedentemente abbiamo definito il predicato di parità usando una costruzione induttiva. Poiché un predicato è in realtà una funzione, per definire la parità si può anche usare la funzione:

```
Fixpoint Even (n: nat) {struct n}: Prop :=
  match n with
     O         => True
  | (S O)      => False
  | (S (S n1)) => (Even n1)
  end.
```

La terminazione è assicurata, poiché `n1` ha due `S` meno di `n`. Definire qualcosa tramite una definizione induttiva vuol dire definire una nuova nozione e dare le sue proprietà fondamentali. Usare una funzione invece è più diretto. Questo concetto si vede sulle prove: provare `Even (S (S O))` è più semplice perché `Even (S (S O))` vale `True`. Dunque, una prova di `True` è sufficiente. Se si vuole usare il fatto di essere pari per definire altre funzioni, occorre però definire la funzione `Even` nel mondo del calcolo (`Set`) e non in quello delle prove (`Prop`).

Connettivi logici come tipi induttivi.

In realtà i connettivi logici non sono primitivi ma possono essere definiti come tipi induttivi.

- Congiunzione: una prova di A ∧ B è una coppia di prove data da una prova di A e da una prova di B.
- Disgiunzione: una prova di A ∨ B è o una prova di A o una prova di B.
- Verità: è definita come un oggetto che ha una prova I.
- Falsità: è definita come un oggetto che non ha una prova.
- Negazione: ¬A è definita come A ⇒ False.
- Quantificazione esistenziale: è definita come una coppia data da un elemento che verifica la proprietà e la prova che tale elemento verifica la proprietà.

```
Inductive and (A: Prop) (B: Prop): Prop :=
  conj: A ->  B ->  (and A B).
```

```
Inductive or (A: Prop)(B: Prop): Prop :=
  or_introl: A -> (or A B)
| or_intror: B ->  (or A B).
```

```
Inductive True: Prop :=
  I: True.
```

```
Inductive False: Prop := .
```

```
Definition not: Prop -> Prop := fun A: Prop => A -> False.
```

```
Inductive ex (A: Set) (P: (A -> Prop)): Prop :=
  ex_intro: forall x: A, (P x) -> (ex A P).
```

Prova per ricorsione.

Una tecnica di prova usuale in matematica è la cosiddetta prova per ricorsione. Per i numeri naturali tale prova si comporta così: se possiamo provare una proprietà per 0 e se per un numero arbitrario n sappiamo che la proprietà è vera per n e la si può dedurre per n+1, allora sappiamo che tale proprietà è vera per tutti i numeri naturali. Non solo si può esprimere questo principio come una formula Coq, ma si può anche provarlo. Inoltre, un tale principio può essere costruito nello stesso modo per tutte le definizioni induttive. Infatti, ogni volta che si definisce un costruttore induttivo, Coq definisce automaticamente anche il principio di induzione, il cui nome è formato da quello del tipo più il suffisso `_ind`.

```
Check Even_ind.
Even_ind
     : forall P: nat -> Prop,
       (P O)
       -> (forall n: nat, (Even n) -> (P n) -> (P (S (S n))))
       -> forall n: nat, (Even n) -> (P n)
```

Prove interattive.

Per il momento abbiamo visto che, seguendo l'isomorfismo di Curry-Howard, per costruire una prova di una formula il procedimento consiste nel dare un oggetto il cui tipo è la formula. Costruire tale oggetto da capo può essere difficile, specialmente per prove un po' complicate. In Coq è comunque possibile costruire tale oggetto in modo interattivo. La costruzione interattiva è basata sulla nozione di tattica. Una tattica è un'operazione che prende un obiettivo (goal) e restituisce una lista di sotto-obiettivi (subgoals). L'idea è che se i sotto-obiettivi possono essere provati, allora vuol dire che l'obiettivo iniziale è valido. Un obiettivo è composto dalla formula da provare (conclusione) e dall'ambiente (contesto) dentro il quale si deve provare la formula. L'ambiente è costituito dalla lista delle ipotesi che si possono usare per provare la formula.

Comandi dell'ambiente interattivo.

Il comando `Theorem` permette di iniziare a fare una prova in modo interattivo:

`Theorem <NAME>: <TYPE>.`

A questo punto si possono usare le tattiche oppure i comandi `Undo` e `Abort` per cancellare l'ultima tattica e uscire dall'ambiente interattivo, rispettivamente. Quando la prova è finita (non ci sono più obiettivi), si può registrarla usando il comando `Qed`.

Tattiche di base.

- La tattica `exact` permette di dare direttamente l'espressione (λ-termine) che ha come tipo l'obiettivo che vogliamo provare.
- Se un oggetto `n` ha un tipo induttivo `T`, applicando la tattica `elim n`, si inizia una prova per ricorsione su `T`.

Tattiche di base per i connettivi logici.

$A \to B$	$\dfrac{\Gamma, h : A \vdash ? : B}{\Gamma \vdash ? : A \to B}$	`intro` h
	$\dfrac{\Gamma \vdash h : A \to B \quad \Gamma \vdash ? : A}{\Gamma \vdash ? : B}$	`apply` h
$\forall x : A, B$	$\dfrac{\Gamma, x : A \vdash ? : B}{\Gamma \vdash ? : \forall x : A, B}$	`intro` x
	$\dfrac{\Gamma \vdash h : \forall x : A, B \quad \Gamma \vdash t : A}{\Gamma \vdash ? : B[x \leftarrow t]}$	`apply` h `with` $(x := t)$
$A \wedge B$	$\dfrac{\Gamma \vdash ? : A \quad \Gamma \vdash ? : B}{\Gamma \vdash ? : A \wedge B}$	`split`
	$\dfrac{\Gamma \vdash h : A \wedge B \quad \Gamma, h_A : A, h_B : B \vdash ? : C}{\Gamma \vdash ? : C}$	`destruct` h `as` (h_A, h_B)
$\exists x : A, B$	$\dfrac{\Gamma \vdash t : A \quad \Gamma \vdash ? : B[x \leftarrow t]}{\Gamma \vdash ? : \exists x : A, B}$	`exists` t
	$\dfrac{\Gamma \vdash h : \exists x : A, B \quad \Gamma, x : A, h_B : B \vdash ? : C}{\Gamma \vdash ? : C}$	`destruct` h `as` $[x\ h_b]$

$A \vee B$	$\dfrac{\Gamma \vdash ? : A}{\Gamma \vdash ? : A \vee B}$
	
	$\dfrac{\Gamma \vdash ? : B}{\Gamma \vdash ? : A \vee B}$
	
	$\dfrac{\Gamma \vdash h : A \vee B \quad \Gamma, h_A : A \vdash ? : C \quad \Gamma, h_B : B \vdash ? : C}{\Gamma \vdash ? : C}$
		`left`
		
		`right`
		
		`destruct` h `as` $[h_A \mid h_B]$

`False`	$\dfrac{\Gamma \vdash h : \texttt{False}}{\Gamma \vdash ? : C}$	`destruct` h
$\neg A$ $(\equiv A \to \texttt{False})$	$\dfrac{\Gamma, h : A \vdash \texttt{False}}{\Gamma \vdash ? : \neg A}$ $\dfrac{\Gamma \vdash h : \neg A \quad \Gamma \vdash A}{\Gamma \vdash ? : C}$	`intro` h `destruct` h

Deduzione naturale.

Una tabella di verità non dà spiegazioni sul perché una formula è una tautologia. C'è quindi bisogno della deduzione naturale, ovvero di un modo per costruire tautologie. Secondo tale costruzione, per ogni operatore ci sono delle regole per creare l'operatore (introduzione) e delle regole per usare l'operatore (eliminazione).

$$(\wedge I)\ \frac{\varphi \quad \psi}{\varphi \wedge \psi}\wedge I$$

$$(\wedge E)\ \frac{\varphi \wedge \psi}{\varphi}\wedge E \qquad \frac{\varphi \wedge \psi}{\psi}\wedge E$$

$$(\to I)\ \frac{\begin{array}{c}[\varphi]\\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\to I$$

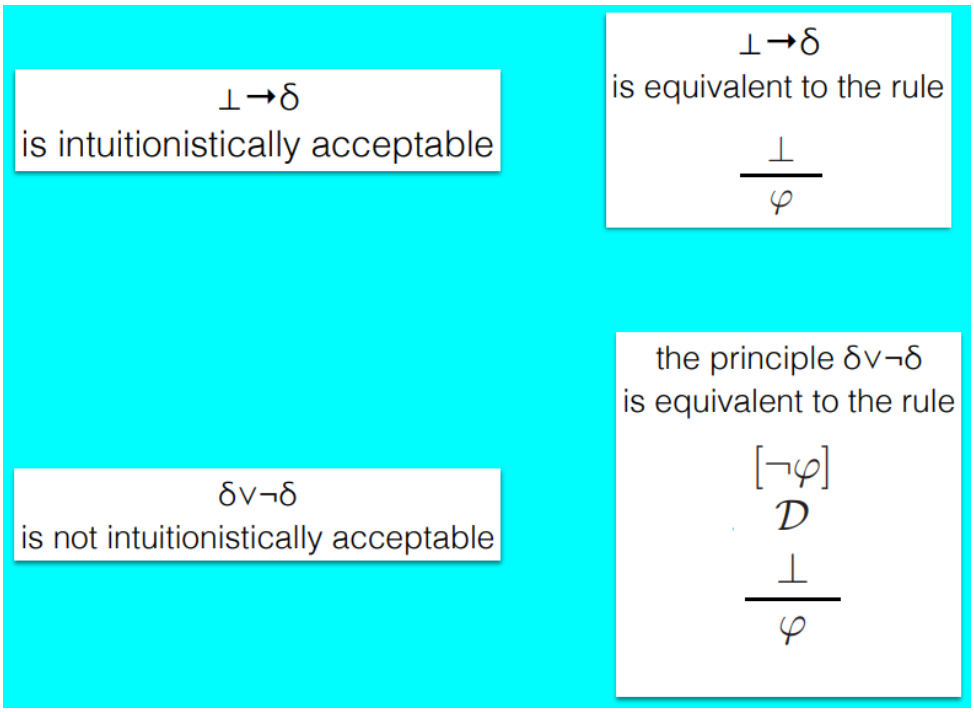
$$(\to E)\ \frac{\varphi \quad \varphi \to \psi}{\psi}\to E$$

$$\frac{\varphi}{\varphi \vee \psi}\vee I \qquad \frac{\psi}{\varphi \vee \psi}\vee I$$

$$\frac{\varphi \vee \psi \qquad \begin{array}{c}[\varphi]\\ \vdots \\ \sigma\end{array} \qquad \begin{array}{c}[\psi]\\ \vdots \\ \sigma\end{array}}{\sigma}\vee E$$

Intuizionismo e logica classica.

Con le regole precedenti abbiamo la logica intuizionista. Per avere la logica usuale, invece, occorre che per ogni formula A sia vero A ∨ ¬A.



Predicati, proposizioni e funzioni.

Un predicato è una proprietà o relazione tra oggetti. Quando un predicato è applicato su tutti i suoi argomenti diventa una proposizione. Le funzioni, invece, sono diverse dai predicati in quanto prendono oggetti e restituiscono oggetti. Per esempio:

- `Prime 5` è un predicato che rappresenta il fatto che 5 è primo.
- `Greater 5 3` è un predicato che rappresenta il fatto che 5 è maggiore di 3.

Quantificatori.

Il quantificatore universale permette di costruire formule che sono vere per tutti gli oggetti. Il quantificatore esistenziale, invece, permette di costruire formule che sono vere per almeno un oggetto. Per esempio:

- `∀x. (P x x)` rappresenta il fatto che per ogni `x`, `x` è in relazione con sé stesso tramite `P` (ovvero `P` è una relazione riflessiva).
- `∀n. ∃p. (Greater p n) ∧ (Prime p)` esprime il fatto che per ogni numero `n` esiste un numero `p` maggiore di `n` tale che `p` sia un numero primo (ovvero l'insieme dei numeri primi è infinito).

Variabile libera e variabile legata.

Una variabile si dice legata se si trova sotto un quantificatore con lo stesso nome; altrimenti, la variabile si dice libera. Inoltre, una variabile è sempre legata con il quantificatore più prossimo. Per esempio:

- Nella formula `∃x. (P x y)`, `x` è legata mentre `y` è libera.
- Nella formula `∃x. ∀x. (P x)` la variabile `x` dentro `(P x)` è legata al quantificatore universale.

Il nome delle variabili legate non è significativo: `∀x. ∃y. (P x y)` è equivalente a `∀z. ∃t. (P z t)`. Con le variabili è associata la nozione di sostituzione, la quale permette di rimpiazzare una variabile con una espressione: `F[E/x]` significa che dentro la formula `F`, `x` è stata rimpiazzata con `E`. La sostituzione rispetta i quantificatori: `((P x) ∧ (∀x. (Q x x)))[a/x]` vale `((P a) ∧ (∀x. (Q x x)))`.

Deduzione naturale con i quantificatori.

Aggiungiamo le regole per l'introduzione e l'eliminazione dei quantificatori.

$$\forall I\ \frac{\varphi}{\forall x \varphi} \qquad \forall E\ \frac{\forall x \varphi}{\varphi[t/x]}$$

$$\exists I\ \frac{\varphi[t/x]}{\exists x \varphi} \qquad \exists E\ \frac{\exists x \varphi \qquad \begin{array}{c}[\varphi]\\ \vdots \\ \psi\end{array}}{\psi}$$

Tattiche in Coq.

Tattiche di calcolo proposizionale:

- `apply thm`, applica la risorsa `thm` (può essere un teorema o un'ipotesi).
- `intro H`, introduce un'ipotesi chiamandola H.
- `intros`, introduce tutte le ipotesi.
- `unfold nome`, sostituisce la costante `nome` con la sua definizione.
- `left` e `right`, distruggono la tesi della forma A ∨ B.
- `split`, distrugge la tesi della forma A ∧ B oppure A ⇔ B.
- `destruct thm`, distrugge un'ipotesi (formata da ∧, ∨, ⇔, ∃, ...).
- `exists term`, distrugge ∃ nella tesi.

Tattiche di uguaglianza:

- `reflexivity`, dimostra x = x.
- `symmetry`, riduce x = y ad y = x.
- `transitivity x`, riduce a = b ad a = x e x = b.
- `rewrite thm`, riscrive l'equazione `thm` (ad esempio se `e` : a = b, `rewrite e` sostituisce a con b oppure `rewrite <- e` sostituisce b con a)

Tattiche di dimostrazione automatica:

- `auto`, usa fatti già noti.
- `tauto`, dimostra una tautologia.
- `trivial`, dimostra una semplice proposizione.
- `ring`, risolutore automatico di equazioni.

Tattiche sui tipi induttivi:

- `pattern term`, serve per far capire qual è il predicato legato al goal e ad un suo termine `term` (ad esempio per far emergere un principio di induzione).
- `case`, esegue un'analisi per casi senza ricorsione (si comporta come `elim` ma non utilizza un principio di ricorsione).

Altre tattiche:

- `idtac`, non esegue alcuna operazione.
- `replace term1 with term2`, sostituisce `term1` con `term2`.
- `simpl`, semplifica la tesi.
- `assumption`, serve a terminare una dimostrazione quando il goal si trova tra le ipotesi.
- `cut a`, esegue una dimostrazione per lemmi (se il goal è P passo a dimostrare che vale `a -> P` e che vale `a`).
- `assert a`, variante di `cut` (se il goal è P assumo che `a -> P` e quindi dovrò fornire una dimostrazione per `a` e una dimostrazione per P con ipotesi `a`).
- `induction var`, inizia una prova per induzione su `var`.
- `now tac`, esegue la tattica `tac` seguita da `easy`.
- `easy`, cerca di risolvere il goal usando le tattiche `trivial`, `reflexivity`, `symmetry` e `inversion`.
- `inversion H`, guarda come è fatto il tipo induttivo di H e dice da quale costruttore può essere prodotto H (in pratica inverte la freccia, in quanto i predicati induttivi sono i più piccoli predicati che godono di una certa proprietà).

Approfondimento su apply.

Il funzionamento di base di `apply` prevede che se devo dimostrare Q e ho un'ipotesi `H : P -> Q`, allora se dimostro P riesco a dimostrare anche Q. Tuttavia, è anche possibile applicare un'ipotesi in un'altra ipotesi: ad esempio, se devo dimostrare che `A -> (A -> B) -> B` allora posso derivare B applicando l'ipotesi `H0 : A -> B` direttamente nell'ipotesi `H : A` scrivendo `apply H0 in H`. Inoltre, esiste anche la tattica `apply H with v := t`, che può essere utilizzata quando una variabile dipendente `v` non riesce ad essere dedotta dal goal. Infine, la tattica `eapply` si comporta come `apply` ma non fallisce quando non ci sono istanze deducibili per alcune variabili nelle premesse. Piuttosto, applica un quantificatore esistenziale alle variabili ancora da istanziare. Tali istanze dovranno essere trovate in seguito nella prova.

Approfondimento su inversion.

La tattica `inversion H` funziona quando il predicato principale dell'ipotesi H è un predicato induttivo. Intuitivamente, `inversion H` cerca di aiutarti a ragionare su quale costruttore possa essere stato utilizzato per aver prodotto H, e questo ragionamento è fatto aggiungendo ipotesi che corrispondono ai vincoli che devono essere soddisfatti quando questo costruttore viene utilizzato.

Separatori tra tattiche.

- `.` – le tattiche vengono eseguite una alla volta.
- `;` – composizione sequenziale, se una tattica apre più subgoal quella successiva opera in parallelo su tutti i subgoal.
- `[ … | … ]` – composizione parallela, sapendo che ho un certo numero di subgoal applico per ognuno le tattiche separate da `|` in parallelo.
- `( … || … )` – or-else, se la prima tattica ha successo si va avanti mentre se fallisce prova ad applicare la seconda.

Sommario.

I dimostratori di teoremi automatici (DTA) sono programmi creati per dimostrare che un'asserzione è conseguenza logica (in un determinato sistema formale, ovvero in una logica) di un insieme di asserzioni (assiomi ed ipotesi). Il limite dei DTA risiede nel riuscire a dare una adeguata formulazione dell'asserzione da dimostrare e degli assiomi/ipotesi che devono descrivere il problema da risolvere.

Il linguaggio in cui l'asserzione, gli assiomi e le ipotesi vengono comunemente codificate è la logica classica del primo ordine, anche se molti DTA si basano su logiche non classiche (come la logica intuizionista) o su logiche di ordine superiore. Nella comune definizione di DTA sono comprese due diverse tipologie di programma:

- Proof Checker, i quali ricevono in input dall'utente una dimostrazione e ne decidono la correttezza.
- Proof Assistant, i quali pur presentando alcune tecniche di deduzione automatica hanno il compito di mantenere uno stato della dimostrazione e di fornire tattiche per modificare tale stato fino a pervenire ad una dimostrazione completa (Coq appartiene a questa tipologia).

Proof Assistant Coq.

Coq è un sistema di dimostrazione interattivo (Proof Assistant) progettato principalmente per scrivere specifiche formali e per verificare che determinati programmi soddisfino tali specifiche. Viene associato a Coq un linguaggio di specifica dei problemi, chiamato Gallina, i cui termini possono rappresentare programmi, loro proprietà o prove di quest'ultime. Grazie all'isomorfismo di Curry-Howard (fra i termini di un sistema di tipi come il λ-calcolo tipato e una logica basata sulla deduzione naturale) programmi, proprietà e prove sono formalizzati in uno stesso linguaggio, chiamato Calcolo delle Costruzioni Induttive (CIC), che è un'estensione del λ-calcolo caratterizzata da un ricco sistema di tipi funzionali e dalla presenza di tipi induttivi e co-induttivi primitivi. Naturalmente una teoria con alto potere espressivo come il CIC permette di utilizzare Coq anche come ambiente logico per la codifica di teorie matematiche e per la dimostrazione di quest'ultime. Infatti, il CIC è il prodotto della sintesi di due sistemi di tipi molto potenti quali la teoria intuizionista dei tipi (da cui eredita una grande espressività nella definizione di tipi induttivi) e il sistema-F (o λ-calcolo del secondo ordine). Nella teoria intuizionista dei tipi, i tipi dipendenti sono usati per codificare i quantificatori.

$$\mathbf{Prod}(s,s',s'')\ \frac{E,\Gamma \vdash A : s \quad E,\Gamma :: (a : A) \vdash B : s'}{E,\Gamma \vdash \forall\, a{:}A,\, B : s''}$$

Triplet (s, s', s'')	constraints	role
(s, s', s')	$s, s' \in \{\mathtt{Set}, \mathtt{Prop}\}$	simple types
$(\mathtt{Type}(i), \mathtt{Prop}, \mathtt{Prop})$		Impredicativity in Prop
$(s, \mathtt{Type}(i), \mathtt{Type}(i))$	$s \in \{\mathtt{Set}, \mathtt{Prop}\}$	dependence
$(\mathtt{Type}(i), \mathtt{Type}(j), \mathtt{Type}(k))$	$(i \leq k,\ j \leq k)$	higher-order

Fig. 4.4. The triplets for the Calculus of Constructions

Stile di ragionamento.

In generale Coq si fonda su uno stile di ragionamento chiamato deduzione naturale, ossia lo stile che decompone il ragionare in regole di introduzione, che indicano come provare un'asserzione di un determinato tipo, e di eliminazione, che indicano come usare asserzioni di un determinato tipo. Oltre alle tattiche base di introduzione ed eliminazione per i vari tipi (proposizioni), Coq rende disponibili alcune tecniche di dimostrazione automatica; le principali sono:

- `auto`, la quale implementa una procedura di risoluzione del goal tramite l'applicazione delle tattiche di base e facendo riferimento ad un database di teoremi conosciuti (funziona solo per determinate tipologie di asserzioni).
- `tauto`, la quale presenta una procedura di risoluzione del goal più potente di `auto`.

Credits

Basato sulle note del corso di Metodi Formali dell'Informatica, Laurent Théry (DISIM – UnivAQ)

Basato sulle slide fornite dal *prof. Andrea Masini*

Repository GitHub personale: https://github.com/zampierida98/UniVR-informatica

Indirizzo e-mail personale: zampieri.davide@outlook.com