Azzolini Riccardo 2021-05-24

Correttezza

1 Correttezza di un programma funzionale

Formalmente, verificare la correttezza di un programma funzionale significa dimostrare che esso calcola una certa funzione matematica, la quale corrisponde alla specifica del programma. Per semplicità, tale problema verrà qui ridotto al problema di dimostrare che le definizioni delle funzioni soddisfano determinate proprietà. Adesso verranno presentate delle tecniche che permettono di effettuare tali dimostrazioni.

2 Principio di induzione sui numeri naturali

La tecnica principe (ma non l'unica) usata per le dimostrazioni di proprietà dei programmi funzionali è il **principio di induzione**, la cui forma di base è quella sui numeri naturali: per dimostrare che una certa proprietà P vale per tutti i numeri naturali $n \geq b$ (dove b è spesso 0) si dimostra che

- **caso base**: vale $P(b)$;
- **passo induttivo**: per ogni $n \geq b$, $P(n)$ implica $P(n+1)$.

2.1 Esempio: funzione fattoriale

Si consideri la funzione

```
def factorial(n: Int): Int =
  if (n == 0) 1              // (1)
  else n * factorial(n - 1)  // (2)
```

Si vuole dimostrare che per ogni $n \geq 4$ vale la proprietà

$$P(n) = (\texttt{factorial}(n) \geq 2^n)$$

Per svolgere questa dimostrazione, che viene fatta per induzione su n, bisogna ragionare su come il codice di `factorial` viene eseguito nel modello di sostituzione.

- *Caso base*: $n = 4$. Calcolando il valore di `factorial(4)` si ottiene

$$\texttt{factorial(4)} = 24 \geq 16 = 2^4$$

  dunque $P(4)$ è verificata.

- *Passo induttivo*: $n = h + 1$, con $h \geq 4$. Si deve dimostrare che

$$P(h) = (\texttt{factorial(}h\texttt{)} \geq 2^h)$$

  implica

$$P(h+1) = (\texttt{factorial(}h+1\texttt{)} \geq 2^{h+1})$$

  A tale scopo, si assume (**ipotesi induttiva**, IH) che la proprietà P valga per h

$$\texttt{factorial(}h\texttt{)} \geq 2^h \tag{IH}$$

  e si dimostra che allora essa vale per $h + 1$:

$$\begin{aligned}
\texttt{factorial(}h+1\texttt{)} &= \texttt{(}h+1\texttt{) * factorial(}h\texttt{)} && \text{[per (2)]} \\
&> 2 \texttt{ * factorial(}h\texttt{)} && [h \geq 4, \text{ quindi } h+1 > 2] \\
&\geq 2 \texttt{ * } 2^h && \text{[per (IH)]} \\
&= 2^{h+1}
\end{aligned}$$

In questa dimostrazione si sono applicati passi di riduzione ad alcuni dei termini coinvolti: ad esempio, nel passo induttivo il termine `factorial(`$h+1$`)` è stato sotituito con `(`$h+1$`) * factorial(`h`)`, poiché nel caso $n = h+1 \neq 0$ l'applicazione `factorial(`n`)` si riduce appunto al ramo (2) dell'`if-else` che costituisce il corpo della funzione. Un'osservazione importante è che tali passi di riduzione sono stati scritti come uguaglianze, con il simbolo $=$ anziché $\to$ o $\twoheadrightarrow$. Il fatto che ciò sia corretto non è scontato, va giustificato.

3 Trasparenza referenziale

Siccome i programmi funzionali non hanno effetti collaterali, se un termine t si riduce a un altro termine t', $t \twoheadrightarrow t'$, allora si possono considerare i due termini uguali, $t = t'$, cioè indistinguibili dal punto di vista del processo di valutazione: sostituire t con t' o viceversa nel contesto di una qualsiasi espressione non cambia il valore finale (il termine non ulteriormente riducibile) a cui l'espressione si riduce. Infatti, si può dimostrare formalmente che, dati t e t' tali che $t \twoheadrightarrow t'$, una qualunque espressione $E(\ldots t \ldots)$ si riduce a un valore V (non ulteriormente riducibile) se e solo se anche l'espressione $E(\ldots t' \ldots)$ ottenuta sostituendo t con t' si riduce allo stesso valore V,

$$E(\ldots t \ldots) \twoheadrightarrow V \iff E(\ldots t' \ldots) \twoheadrightarrow V$$

cioè le due espressioni sono in un certo senso rappresentazioni diverse dello stesso valore.

Questo principio prende il nome di **trasparenza referenziale** (**referential transparency**), ed è importante perché permette di applicare il **ragionamento equazionale**, il modo di operare sulle equazioni/uguaglianze usato tipicamente in matematica. In particolare, la possibilità di applicare un'uguaglianza in entrambi i versi, sostituendo uno qualsiasi dei due lati con l'altro, dà una maggiore "agilità" nelle dimostrazioni rispetto a quella che si otterrebbe lavorando direttamente con la relazione di riduzione, la quale consente la sostituzione solo nel verso da sinistra a destra.

Data un'uguaglianza $t = t'$ ricavata da una relazione di riscrittura $t \twoheadrightarrow t'$, si chiamano:

- *unfold* l'operazione di sostituzione del lato sinistro t con il lato destro t' (cioè la sostituzione nel verso della riduzione $t \twoheadrightarrow t'$) in una qualche espressione;
- *fold* l'operazione di sostituzione del lato destro t' con il lato sinistro t (cioè la sostituzione nel verso opposto alla riduzione $t \twoheadrightarrow t'$) in una qualche espressione.

4 Esempio: associatività di ++

Si consideri l'operatore `++` di concatenazione sulle liste. Si vuole dimostrare che esso è associativo:

$$\forall xs, ys, zs((xs \texttt{ ++ } ys) \texttt{ ++ } zs = xs \texttt{ ++ } (ys \texttt{ ++ } zs))$$

In questa dimostrazione si sfrutta il **principio di induzione strutturale**, una generalizzazione del principio di induzione applicabile a tutte le strutture definite in modo ricorsivo. Nel caso delle liste, tale principio assume la forma enunciata in seguito (ed è equivalente al principio di induzione sui numeri naturali applicato alla lunghezza di una lista): per dimostrare che una certa proprietà P vale per tutte le liste xs, si dimostra che

- *caso base*: vale $P(\texttt{Nil})$;
- *passo induttivo*: per ogni lista xs e per ogni elemento x, $P(xs)$ implica $P(x \texttt{ :: } xs)$.

Per procedere con la dimostrazione è necessario conoscere l'implementazione dell'operatore `++`. Per semplicità, invece del metodo `++` della classe `List` si considera la funzione

```
def concat[T](xs: List[T], ys: List[T]): List[T] = xs match {
  case Nil => ys
  case x :: xs1 => x :: concat(xs1, ys)
}
```

e si interpreta `xs ++ ys` come un'abbreviazione di `concat(xs, ys)`.

Il corpo di `concat` è un'espressione `match`, che definisce come la funzione viene riscritta a seconda dei valori degli argomenti (di fatto, il simbolo `=>` di ciascun `case` rappresenta la relazione di riscrittura). Allora, per il principio di trasparenza referenziale si identificano dai casi del `match` (cioè dalle possibili riscritture) i seguenti **assiomi**, un insieme di equazioni che caratterizzano completamente il comportamento della funzione `concat`:[1]

$$\forall ys(\texttt{concat(Nil, } ys\texttt{)} = ys) \tag{C1}$$
$$\forall x, xs1, ys(\texttt{concat(}x\texttt{ :: } xs1\texttt{, } ys\texttt{)} = x\texttt{ :: concat(}xs1\texttt{, } ys\texttt{)}) \tag{C2}$$

Ora questi assiomi possono essere riscritti usando `++` come abbreviazione di `concat`:

$$\forall ys(\texttt{Nil ++ } ys = ys) \tag{C1}$$
$$\forall x, xs1, ys(\texttt{(}x\texttt{ :: } xs1\texttt{) ++ } ys = x\texttt{ :: (}xs1\texttt{ ++ } ys\texttt{)}) \tag{C2}$$

Un'altra cosa che conviene fare è rinominare le variabili che compaiono negli assiomi, in modo da evitare confusione con le (diverse) variabili trattate al momento dell'applicazione degli assiomi (le quali altrimenti potrebbero facilmente avere gli stessi nomi, dato che per le liste si tendono a usare sempre nomi come `x`, `xs1`, `ys`, ecc.):

$$\forall list(\texttt{Nil ++ } list = list) \tag{C1}$$
$$\forall head, tail, list(\texttt{(}head\texttt{ :: } tail\texttt{) ++ } list = head\texttt{ :: (}tail\texttt{ ++ } list\texttt{)}) \tag{C2}$$

Infine, per brevità si lascia implicita la quantificazione universale delle variabili che compaiono negli assiomi:

$$\texttt{Nil ++ } list = list \tag{C1}$$
$$\texttt{(}head\texttt{ :: } tail\texttt{) ++ } list = head\texttt{ :: (}tail\texttt{ ++ } list\texttt{)} \tag{C2}$$

Adesso si può procedere con la dimostrazione vera e propria dell'associatività di `++`, cioè della proprietà P che afferma che, per ogni `xs`, `ys` e `zs`:

$$\texttt{(}xs\texttt{ ++ } ys\texttt{) ++ } zs = xs\texttt{ ++ (}ys\texttt{ ++ } zs\texttt{)} \tag{P}$$

La dimostrazione si svolge per induzione strutturale su `xs`.

- *Caso base*: `xs` = `Nil`. Per dimostrare il caso base si parte dal lato sinistro della proprietà (P), si applica l'assioma (C1) nel verso da sinistra a destra per "togliere" il `Nil` dentro le parentesi, e infine si applica di nuovo (C1), questa volta nel verso da destra a sinistra, per "rimettere" `Nil` fuori dalle parentesi in modo da ottenere il lato destro della proprietà (P):

$$\begin{aligned} \texttt{(Nil ++ } ys\texttt{) ++ } zs &= ys\texttt{ ++ } zs && \text{[per (C1)]} \\ &= \texttt{Nil ++ (}ys\texttt{ ++ } zs\texttt{)} && \text{[per (C1)]} \end{aligned}$$

[1] Siccome gli assiomi relativi a una funzione ne caratterizzano *completamente* il comportamento, essi sono sufficienti per dimostrare qualunque proprietà dimostrabile della funzione: conoscere il codice è necessario solo al fine di estrarre gli assiomi, quando questi non sono già noti.

- *Passo induttivo*: $xs = x$ `::` $xs1$. Bisogna dimostrare

$$((x :: xs1) ++ ys) ++ zs = (x :: xs1) ++ (ys ++ zs) \qquad \text{(P)}$$

assumendo per ipotesi induttiva che:

$$(xs1 ++ ys) ++ zs = xs1 ++ (ys ++ zs) \qquad \text{(IH)}$$

Anche in questo caso si potrebbe partire da un lato della proprietà (P) e cercare di trasformarlo nell'altro lato, ma è forse più intuitivo trattare separatamente i due lati, "semplificandoli" con gli assiomi e applicando l'ipotesi induttiva per ottenere la stessa espressione da entrambi i lati. A sinistra si ottiene

$$\begin{aligned} &((x :: xs1) ++ ys) ++ zs \\ &\quad = (x :: (xs1 ++ ys)) ++ zs && \text{[per (C2)]} \\ &\quad = x :: ((xs1 ++ ys) ++ zs) && \text{[per (C2)]} \\ &\quad = x :: (xs1 ++ (ys ++ zs)) && \text{[per (IH)]} \end{aligned}$$

mentre a destra si ottiene

$$\begin{aligned} &(x :: xs1) ++ (ys ++ zs) && \text{[per (C2)]} \\ &\quad = x :: (xs1 ++ (ys ++ zs) \end{aligned}$$

che è appunto la stessa espressione, quindi la (P) è dimostrata anche nel caso induttivo.

L'associatività di `++` è così dimostrata. Usando le stesse tecniche si potrebbero dimostrare varie altre proprietà di `++`, tra cui ad esempio il fatto che `Nil` sia l'elemento neutro, espresso dalle seguenti due proprietà:

$$\forall xs(xs ++ \texttt{Nil} = xs) \qquad \text{(P1)}$$
$$\forall xs(\texttt{Nil} ++ xs = xs) \qquad \text{(P2)}$$

la (P1) si dimostra per induzione strutturale su xs, mentre la (P2) è immediatamente dimostrata perché coincide con l'assioma (C1).

5 Esempio: `reverse`

Si consideri la funzione `reverse`,

```
def reverse[T](xs: List[T]): List[T] = xs match {
  case Nil => Nil
  case y :: ys => reverse(ys) ++ List(y)
}
```

dalla cui definizione si possono estrarre i seguenti assiomi (qui riportati con le variabili già rinominate):

$$\texttt{reverse(Nil) = Nil} \tag{R1}$$

$$\forall \textit{head}, \textit{tail}(\texttt{reverse(}\textit{head}\texttt{ :: }\textit{tail}\texttt{)} = \texttt{reverse(}\textit{tail}\texttt{) ++ List(}\textit{head}\texttt{)}) \tag{R2}$$

(si noti che (R1) non ha alcuna quantificazione, neanche implicita, poiché al suo interno non compaiono variabili — c'è solo la costante `Nil`).

Come esempio di caso un po' più complicato rispetto ai precedenti si vuole dimostrare la seguente proprietà:

$$\forall \textit{xs}(\texttt{reverse(reverse(}\textit{xs}\texttt{))} = \textit{xs}) \tag{P}$$

La dimostrazione procede per induzione strutturale su $\textit{xs}$.

- *Caso base*: $\textit{xs} = \texttt{Nil}$.

$$\begin{aligned} \texttt{reverse(reverse(Nil))} &= \texttt{reverse(Nil)} && \text{[per (R1)]} \\ &= \texttt{Nil} && \text{[per (R1)]} \end{aligned}$$

- *Passo induttivo*: $\textit{xs} = \textit{y}\texttt{ :: }\textit{ys}$. Per ipotesi induttiva:

$$\texttt{reverse(reverse(}\textit{ys}\texttt{))} = \textit{ys} \tag{IH}$$

  Come nell'esempio precedente si sceglie di ragionare separatamente sui due lati della proprietà (P), cercando di renderli uguali.

  Sul lato sinistro di (P) si applica l'assioma (R2) nel verso da sinistra a destra,

$$\begin{aligned} &\texttt{reverse(reverse(}\textit{y}\texttt{ :: }\textit{ys}\texttt{))} \\ &\quad = \texttt{reverse(reverse(}\textit{ys}\texttt{) ++ List(}\textit{y}\texttt{))} && \text{[per (R2)]} \end{aligned}$$

  ma a questo punto non si hanno più uguaglianze (assiomi di `reverse` o di `++`, ipotesi induttiva, ecc.) applicabili per effettuare sostituzioni (si potrebbe applicare di nuovo (R2) nel verso opposto, ma così si tornerebbe inutilmente al punto di partenza).

  Sul lato destro si applica l'ipotesi induttiva,

$$\textit{y}\texttt{ :: }\textit{ys} = \textit{y}\texttt{ :: reverse(reverse(}\textit{ys}\texttt{))} \qquad \text{[per (IH)]}$$

  e anche in questo caso non c'è altro da fare.

  La dimostrazione del passo induttivo rimane dunque bloccata sull'uguaglianza

$$\texttt{reverse(reverse(}\textit{ys}\texttt{) ++ List(}\textit{y}\texttt{))} = \textit{y}\texttt{ :: reverse(reverse(}\textit{ys}\texttt{))}$$

Il modo "corretto" di procedere a questo punto non è scontato: costruire le dimostrazioni è un processo creativo, non meccanico, e non è detto che la prima "strada" provata in una certa situazione sia quella giusta. Una possibilità è cercare un altro modo (diverso dall'applicazione diretta di assiomi e ipotesi induttiva, che in questo caso non è bastato) di dimostrare l'uguaglianza a cui ci si è attualmente fermati.

Un'osservazione chiave è che l'uguaglianza

$$\texttt{reverse(reverse(}ys\texttt{) ++ List(}y\texttt{))} = y \texttt{ :: reverse(reverse(}ys\texttt{))}$$

non può essere ulteriormente semplificata perché tutte le possibili semplificazioni (sull'operatore `++` a sinistra, su `reverse(reverse(`ys`))` a destra, ecc.) dipendono dalla struttura di `reverse(`ys`)`, che è una qualche lista (dipendente da ys) sulla quale non si ha modo di ragionare induttivamente poiché non si sa esattamente che forma abbia. Una cosa che si può invece fare (ed è una tecnica usata spesso nelle dimostrazioni) è provare a **generalizzare** l'uguaglianza, chiedendosi se essa sia una proprietà che vale anche quando si sostituisce a `reverse(`ys`)` una generica lista zs:

$$\forall zs(\texttt{reverse(}zs\texttt{ ++ List(}y\texttt{))} = y \texttt{ :: reverse(}zs\texttt{)} \qquad \text{(Q)}$$

Infatti, se si dimostrasse (Q) per ogni lista zs si otterrebbe in particolare che essa vale per la lista `reverse(`ys`)`, e ciò completerebbe la dimostrazione della proprietà (P).

Siccome zs è una generica lista, senza alcun vincolo sulla sua struttura (a differenza di `reverse(`ys`)`, che ha una qualche relazione con ys), si può provare a dimostrare (Q) per induzione strutturale su zs. Nello svolgimento, oltre agli assiomi di `reverse` e `++`

$$\texttt{reverse(Nil) = Nil} \qquad \text{(R1)}$$
$$\texttt{reverse(}head\texttt{ :: }tail\texttt{)} = \texttt{reverse(}tail\texttt{) ++ List(}head\texttt{)} \qquad \text{(R2)}$$

$$\texttt{Nil ++ }list = list \qquad \text{(C1)}$$
$$\texttt{(}head\texttt{ :: }tail\texttt{) ++ }list = head\texttt{ :: (}tail\texttt{ ++ }list\texttt{)} \qquad \text{(C2)}$$

sarà utile anche il seguente assioma relativo al costruttore `List` (ovvero al metodo `List.apply`):

$$\texttt{List(}head\texttt{)} = head\texttt{ :: Nil} \qquad \text{(L)}$$

- *Caso base*: zs `= Nil`.

$$\begin{aligned}
\texttt{reverse(Nil ++ List(}y\texttt{))} &= \texttt{reverse(List(}y\texttt{))} && \text{[per (C1)]}\\
&= \texttt{reverse(}y\texttt{ :: Nil)} && \text{[per (L)]}\\
&= \texttt{reverse(Nil) ++ List(}y\texttt{)} && \text{[per (R2)]}\\
&= \texttt{Nil ++ List(}y\texttt{)} && \text{[per (R1)]}\\
&= \texttt{List(}y\texttt{)} && \text{[per (C1)]}\\
&= y\texttt{ :: Nil} && \text{[per (L)]}\\
&= y\texttt{ :: reverse(Nil)} && \text{[per (R1)]}
\end{aligned}$$

  Quindi (Q) è dimostrata nel caso base.

- *Passo induttivo*: `zs` $=$ `w :: ws`. Per ipotesi induttiva:

$$\texttt{reverse(}ws\texttt{ ++ List(}y\texttt{))} = y\texttt{ :: reverse(}ws\texttt{)} \qquad \text{(IH)}$$

  Allora:

$$\begin{aligned}
&\texttt{reverse((}w\texttt{ :: }ws\texttt{) ++ List(}y\texttt{))} \\
&\quad = \texttt{reverse(}w\texttt{ :: (}ws\texttt{ ++ List(}y\texttt{)))} && \text{[per (C2)]} \\
&\quad = \texttt{reverse(}ws\texttt{ ++ List(}y\texttt{)) ++ List(}w\texttt{)} && \text{[per (R2)]} \\
&\quad = \texttt{(}y\texttt{ :: reverse(}ws\texttt{)) ++ List(}w\texttt{)} && \text{[per (IH)]} \\
&\quad = y\texttt{ :: (reverse(}ws\texttt{) ++ List(}w\texttt{))} && \text{[per (C2)]} \\
&\quad = y\texttt{ :: reverse(}w\texttt{ :: }ws\texttt{)} && \text{[per (R2)]}
\end{aligned}$$

  Dunque (Q) è dimostrata anche nel caso induttivo.

Ciò conclude la dimostrazione di (Q), e di conseguenza quella di (P).