Haskell for Dummies

Luca Abeni

March 29, 2023

1 Introduzione

Mentre un linguaggio di programmazione imperativo rispecchia l'architettura di Von Neumann, descrivendo i programmi come sequenze di comandi (istruzioni) che modificano uno stato (per esempio, il contenuto di locazioni di memoria identificate da variabili), un linguaggio funzionale codifica i programmi come espressioni che vengono valutate, generando valori come risultati. Non esite più quindi un riferimento diretto all'architettura di Von Neumann e viene a mancare il concetto stesso di "stato" o variabili mutabili (variabili il cui contenuto può essere modificato).

Come detto, i programmi scritti in un linguaggio funzionale vengono eseguiti valutando espressioni. Informalmente parlando, esistono espressioni "complesse", che possono essere semplificate, ed espressioni "semplici", vale a dire non semplificabili ulteriormente. Un'espressione non semplificabile e' un *valore*, mentre un'espressione complessa può essere semplificata fino ad arrivare un valore (non ulteriormente semplificabile); l'operazione che calcola tale valore è detta *riduzione* (o valutazione) dell'espressione[1].

Un'espressione complessa è quindi composta da operazioni (o funzioni) applicate a valori e l'ordine nel quale tali funzioni ed operazioni vengono valutate dipende dal linguaggio. Per esempio, $4 * 3$ è un'espressione "complessa" composta dai valori 4 e 3 e dall'operazione di moltiplicazione. Valutando tale espressione, si riduce (semplifica) a 12, che è il valore del suo risultato. L'espressione "`if (n == 0) then (x + 1) else (x - 1)`", invece, è più "interessante", perché non è chiaro se e quando le sottoespressioni "`x + 1`" e "`x - 1`" vengono valutate: un linguaggio di programmazione potrebbe decidere di valutare sempre e comunque le due espressioni **prima** di valutare l'espressione `if`, oppure potrebbe decidere di valutare "`x + 1`" solo se "`n == 0`" e valutare "`x - 1`" solo se "`n != 0`".

In caso di "eager evaluation", la valutazione dell'espressione è effettuata valutando prima i parametri di ogni operazione e quindi applicando l'operazione ai valori ottenuti, mentre in caso di "lazy evaluation" le varie sottoespressioni sono valuatate solo quando il loro valore è effettivamente utilizzato: se quindi un'espressione è passata come parametro ad una funzione (o come argomento ad un'operazione) ma la funzione non usa tale parametro, il valore dell'espressione non è valutato.

In linguaggi come Standard ML che utilizzano un meccanismo di valutazione "eager", quindi, un'espressione composta da un operatore applicato ad uno o più argomenti è valutata riducendo prima i suoi argomenti e applicando poi l'operatore ai valori ottenuti valutando gli argomenti. Al contrario, in linguaggi come Haskell che utilizzano un meccanismo di valutazione "lazy", un'espressione composta da un operatore applicato ad uno o più argomenti è valutata riducendo gli argomenti solo quando l'operatore li utilizza realmente.

Riassumendo, un programma scritto in un linguaggio funzionale non è niente altro che un'espressione (o un insieme di espressioni), che viene valutata quando il programma esegue. Programmi complessi sono spesso scritti definendo funzioni (nel senso matematico del termine) che vengono poi invocate dall'espressione "principale" che descrive il programma stesso. In teoria, la valutazione di queste espressioni non dovrebbe avere effetti collaterali, ma alcuni tipi di effetti collaterali (per esempio, input ed output) sono spesso molto difficili da evitare. Da tutto questo si può se non altro intuire come un linguaggio di programmazione funzionale possa essere informalmente visto come una "versione di alto livello" del λ calcolo (che aggiunge se non altro il concetto di ambiente globale ed un po' di zucchero sintattico).

A questo proposito, è importante notare che in linguaggi come Haskell e Standard ML (a differenza di altri linguaggi funzionali come LISP o Scheme) le varie espressioni che compongono un programma sono scritte come operazioni o funzioni che agiscono su *valori appartenenti ad un tipo*. Le funzioni stesse sono

[1] Volendo essere precisi, va però notato che esistono espressioni per le quali il processo di riduzione non termina mai e quindi la semplificazione non arriva ad un valore. Tali espressioni possono essere viste come "l'equilalente funzionale" dei cicli infiniti.

caratterizzate da un tipo (una freccia fra il tipo dell'argmento ed il tipo del risultato)[2]. In questo senso, si può dire che linguaggi come LISP o Scheme derivino direttamente dal λ calcolo "semplice", mentre linguaggi come quelli della famiglia ML (Standard ML, ocaml, F#, ...) e Haskell derivino dal λ calcolo con tipi (anche se, per essere Turing-completi, devono usare un sistema di tipi ricorsivi).

In un linguaggio a tipizzazione statica, il tipo di un'espressione e dei suoi parametri è determinato a tempo di compilazione o comunque (in caso di interprete) prima di eseguire il codice: il compilatore/inteprete inferisce (estrapola) i tipi di parametri ed espressione analizzando il codice, senza eseguirlo. Alcuni fra i linguaggi con tipizzazione stretta e statica, come Haskell, complicano leggermente le cose perché usano un sistema di tipi che è anche *polimorfico*, in cui cioè una singola espressione può assumere tipi diversi dipendentemente dal contesto in cui è usata (ma, una volta fissato il contesto, ogni espressione viene valutata — anche se in modo lazy — ad un valore che appartiene ad un tipo, quindi la tipizzazione resta stretta). L'utilizzo di questo sistema di tipi polimorfico unito alla valutazione lazy usata da Haskell può talvolta trarre in inganno, facendo pensare che il linguaggio non utilizzi una tipizzazione stretta.

Nei casi in cui il tipo dei valori e delle espressioni non possa essere facilmente inferito dal compilatore (o dall'interprete) o non si voglia fare uso del polimorfismo, alcuni linguaggi (fra cui, per esempio, i linguaggi della famiglia ML ed Haskell) possono permettere di annotare le espressioni con il loro tipo, usando un'apposita sintassi (che si vedrà in seguito).

2 Tipi ed Espressioni in Haskell

I tipi di dato base forniti da Haskell sono: `()` (talvolta chiamato "unit", che è l'equivalente del tipo "`void`" di altri linguaggi), `Bool`, `Char`, `Int`, `Float`, `Double` e `String`.

Oltre a fornire questi tipi di base, Haskell permette di usare combinazioni di tipi sotto forma di *tuple*, di definire sinonimi per tipi di dato esistente e di definire nuovi tipi di dato (i cui valori sono generati da apposite funzioni chiamate *costruttori*).

Il tipo `()` è composto da un unico valore, chiamato anch'esso `()`[3] ed è utilizzato generalmente come parte del risultato di espressioni che non genererebbero alcun valore (e che sono importanti solo per i propri effetti collaterali). In teoria tali espressioni non dovrebbero esistere in un linguaggio di programmazione puramente funzionale, ma la necessità di effettuare Input/Output (che è un effetto collaterale) complica le cose. Haskell risolve il problema facendo si che le espressioni di I/O ritornino sia un effetto (codificato dal tipo "`IO` α") che un valore (che per le funzioni di output spesso non è rilevante, quindi si utilizza il tipo "`()`"... Il tipo delle funzioni di output sarà quindi "`IO ()`"). Un altro modo di vedere il tipo "`()`" è quello di considerarlo come un tipo "tupla di 0 elementi".

Il tipo `Bool`, invece, è composto da due valori (`True` e `False`).

Il tipo `Int` è composto (come il nome suggerisce) dai numeri interi, positivi e negativi. Su tali numeri sono definiti l'operatore unario `-`, e gli operatori binari standard che rappresentano le operazioni aritmetiche di base `*`, `+` e `-`[4]. L'operatore `/` di divisione non è definito sugli interi (mentre esiste la funzione `div` che calcola la divisione intera). Notare che a differenza di altri linguaggi Haskell accetta espressioni del tipo "`5 / 2`" (invece di segnalare un errore perchè `5` e `2` sono valori interi ma l'operazione "`/`" non è definita sugli interi). Questo avviene perché Haskell converte implicitamente `5` e `2` nei valori floating point `5.0` e `2.0`. Per finire, è da notare che "`div`" è una funzione, non un operatore, quindi la sintassi corretta per usarla è "`div 5 2`", non "`2 div 5`" (volendo, Haskell permette di usare funzioni binarie con la notazione infissa degli operatori a patto di quotarle fra virgolette singole: "`2 ‘div‘ 5`").

I tipi `Float` e `Double` sono composti da un insiemi di approssimazioni di numeri reali (in precisione singola per `Float` ed in precisione doppia per `Double`). I valori di tali tipi possono essere espressi tramite parte intera e parte frazionaria (per esempio, `3.14`) o usando la forma esponenziale (per esempio, `314e-2`). Ancora, il simbolo `-` può essere usato per negare un numero (invertirne il segno). Due valori speciali `NaN` (Not a Number) e `Infinity` possono essere usati per indicare valori non rappresentabili come numeri reali o valori infiniti (il risultato della divisione di un numero reale per 0).

Il tipo `Char` è composto dall'insieme dei caratteri. Un valore di tale tipo viene rappresentato racchiuso fra virgolette singole, come in C; per esempio, `'a'`.

[2] Formalmente, un tipo può essere definito come un insieme di valori ed il tipo di un valore indica l'insieme a cui tale valore appartiene

[3] Attenzione! Si noti che il tipo ed il suo unico valore hanno lo stesso nome, "`()`", e questo a volte può causare un po' di confusione

[4] notare che in Haskell l'operazione `-` (sottrazione) e l'operatore unario `-` che inverte il segno di un numero sono rappresentati dallo stesso simbolo.

Il tipo `String` è composto dall'insieme delle stringhe, rappresentate fra virgolette; per esempio `"test"`. Sulle stringhe, Haskell definisce l'operatore di concatenazione `++`: `"Ciao, " ++ "mondo" = "Ciao, Mondo"`.

Oltre ai "classici" operatori sui vari tipi di variabili Haskell fornisce un operatore di selezione `if`, che permette di valutare due diverse espressioni dipendentemente dal valore di un predicato. La sintassi di un'espressione `if` in Haskell è:

```
if <p> then <exp1> else <exp2>
```

dove `<p>` è un predicato (espressione di tipo booleano) ed `<exp1>` e `<exp2>` sono due espressioni aventi lo stesso tipo o tipi compatibili (notare che `<exp1>` e `<exp2>` devono avere tipi compatibili perché il valore dell'espressione `if` risultante ha lo stesso tipo di `<exp1>` e `<exp2>`). L'espressione `if` viene valutata come `<exp1>` se `<p>` è vero, mentre è valutata come `<exp2>` se `<p>` è falso.

Sebbene l'operatore `if` di Haskell sia spesso considerato l'equivalente a livello di espressione dell'operazione di selezione `if` fornita dai linguaggi imperativi come C, C++, Java o simili, è importante notare alcune differenze. Per esempio, i costrutti di selezione di un linguaggio imperativo permettono di eseguire un blocco di operazioni se il predicato è vero (ramo `then`) o un diverso blocco di operazioni se il predicato è falso (ramo `else`). In teoria, ciascuno dei due blocchi (`then` o `else`) può essere vuoto, a significare che non ci sono operazioni da eseguire per un determinato valore di verità del predicato. L'operatore `if` di Haskell, invece (come l'equivalente operatore di tutti i linguaggi funzionali), deve *sempre* essere valutabile ad un valore. Quindi, nessuna delle due espressioni `then` o `else` può essere vuota. In questo senso, un'espressione "`if` *predicato* `then` *espressione1* `else` *espressione2*" di Haskell è equivalente all'if aritmetico "*predicato* `?` *espressione1* : *espressione2*" del linguaggio C o C++.

Un esempio di utilizzo di `if` è

```
if a > b then a else b
```

che implementa un'espressione valutata al massimo fra `a` e `b`.

Per finire, è da notare una caratteristica importante del sistema di tipi di Haskell: l'esistenza delle *classi di tipi* (typeclass), che raggruppano vari tipi aventi determinate proprietà (vale a dire, per cui sono definite alcune operazioni specifiche).

Per esempio, la classe di tipi "`Eq`" racchiude tutti i tipi su cui è definito l'operatore di confronto "==", la classe "`Show`" contiene tutti i tipi visualizzabili a schermo tramite "`print`", la classe "`Ord`" contiene tutti i tipi su cui sono definiti gli operatori di confronto ("`<`" e "`>`"), la classe "`Num`" contiene tutti i tipi che esprimono valori numerici ("`Int`", "`Float`", "`Double`"), la classe "`Fractional`" contiene tutti i tipi che rappresentano numeri non interi ("`Float`" e "`Double`") e così via.

Il meccanismo di inferenza dei tipi di Haskell non assocerà un valore ad un tipo, ma ad una classe di tipi; quindi, per esempio, il valore "`5`" non è associato al tipo "`Int`", ma ad un generico tipo della classe "`Num`"; se si vuole specificare in modo preciso ed univoco il tipo di un valore, questo va fatto in modo esplicito.

3 Associare Nomi a Valori

Le espressioni che compongono un programma Haskell possono usare come operandi direttamente valori (espressioni irriducibili) o possono usare degli *identificatori* definiti in un *ambiente* per rappresentare dei valori (tecnicamente, si dice che l'ambiente contiene *legami* fra nomi e valori). Un ambiente può essere visto come un insieme di coppie (identificatore, valore) che associano nomi (o identificatori) a valori[5].

Mentre non esiste il concetto di variabile mutabile/modificabile, i vari "collegamenti" (binding) che legano nomi e valori nell'ambiente possono variare nel tempo. L'ambiente può essere infatti modificato (in realtà, esteso) associando un valore (di qualsiasi tipo) ad un nome (identificatore) tramite l'operatore "`=`":

```
<name> = <value>
<name> :: <type>; <name> = <value>
```

L'operatore "`=`" (chiamato *definizione* in Haskell) aggiunge all'ambiente un legame tra l'identificatore `<name>` ed il valore `<value>`, mentre "`::`" (chiamato *dichiarazione*) permette di specificare il tipo del valore. Il valore `<value>` può essere anche il risultato della valutazione di un'espressione; in questo caso, la definizione assume la forma

[5] Si noti che l'ambiente qui descritto è un ambiente globale ed ogni funzione avrà poi un suo ambiente locale.

```
<name> = <expression>
```

per esempio, `v = 10 / 2;` associa il nome "`v`" al valore 5.

Una particolarità interessante di Haskell è che il tipo del valore associato ad un nome non viene determinato nel momento della definizione (momento in cui si aggiunge all'ambiente un legame fra il nome ed il valore), ma quando il valore viene effettivamente utilizzato. Questo comporta che una definizione "a = 5" associa il nome "`a`" ad un generico valore numerico "5", non ad un intero, floating point o altro. Più precisamente, il tipo di "`a`" risulta essere "`Num p => p`", ad indicare un generico tipo "`p`" appartenente alla classe di tipi "`Num`". Tecnicamente, il simbolo "`p`" che rappresenta un tipo è detto *type variable* e l'espressione "`Num p =>`" rappresenta un vincolo su tale variabile (in questo caso, "`p`" deve essere un tipo numerico). Notare che l'operazione "`a/2.5`" è possibile (a differenza di quanto succede con altri linguaggi), perché l'operazione "`/`" è definita su argomenti il cui tipo appartiene alla classe "`Fractional`" ed "`a`" ha un tipo appartenente alla classe "`Num`"; poiché "`Fractional`" è un subset di "`Num`", il tipo di "`a`" può appartenere alla classe "`Fractional`" (come il tipo del valore "`2.5`") e quindi ed è quindi compatibile col tipo di "`/`". Il tipo del risultato è ovviamente "`Fractional a => a`".

Se si vuole che il tipo del valore associato ad un nome non sia polimorfico (ma sia un tipo ben specificato), bisogna dichiararlo esplicitamente come in

```
a = 5
a :: Int
```

In questo caso, l'operazione "`a/2.5`" non sarà possibile ed il compilatore genererà un errore. E' importante notare la differenza fra il meccanismo di inferenza polimorfica di Haskell ed il meccanismo di promozione dei tipi di dato (o conversione automatica dei valori) utilizzato da altri linguaggi: in Haskell, "`5 / 2.5`" è possibile perché il valore "`5`" può avere un tipo appartenente alla classe "`Fractional`", mentre in C, C++, Java o simili "`5`" è un valore intero che viene convertito automaticamente in floating point per eseguire la divisione.

Tornando alla creazione di legami nell'ambiente, è interessante notare che in Haskell l'associazione di un nome ad un valore può essere vista come una *dichiarazione di variabile*: per esempio, si può dire che `pi = 3.14` crea una variabile identificata dal nome `pi` e la lega al valore reale 3.14. Ma va comunque notato che queste variabili sono semplicemente nomi per dei valori, non sono contenitori di valori modificabili. In altre parole, una variabile è immutabile ed ha sempre un valore costante, non modificabile. Una successiva dichiarazione `pi = 3.1415` non modifica il valore della variabile `pi` ma crea un nuovo valore 3.1415 di tipo `Float` o `Double` e lo associa al nome `pi`, "mascherando" l'associazione precedente. Haskell utilizza sempre l'ultimo valore che è stato associato ad un nome. Questo significa che la keyword "`=`" **modifica l'ambiente, non il valore di variabili**: "`=`" definisce sempre una nuova variabile (inizializzata col valore specificato) e crea un nuovo legame (fra il nome specificato e la variabile creata) nell'ambiente.

Per finire, è interessante ricordare che in un linguaggio di programmazione funzionale le funzioni sono valori esprimibili, denotabili e memorizzabili; quindi, una variabile Haskell può rappresentare anche valori di tipo funzione (come vedremo nella prossima sezione). Per esempio, una funzione "sommaquadrati" è definibile come:

```
sommaquadrati x y = x * x + y * y
```

4 Funzioni

Un particolare tipo di dato che non è stato precedentemente citato ma costituisce una caratteristica fondamentale dei linguaggi funzionali è il tipo di dato *funzione*. Come suggerito dal nome, un valore di questo tipo è una funzione, intesa nel senso matematico del termine: una relazione che mappa ogni elemento di insieme dominio in uno ed un solo elemento di un insieme codominio. In un linguaggio funzionale, gli insiemi dominio e codominio sono definiti dai tipi del parametro e del valore generato. Vedendo le cose da un altro punto di vista, si potrebbe dire che una funzione può essere considerata come una *espressione parametrizzata*, vale a dire un'espressione il cui valore dipende dal valore di un parametro.

Si noti che considerare funzioni con un solo parametro non è riduttivo: il parametro può essere una n-upla di valori (quindi, l'insieme dominio è il prodotto cartesiano di n insiemi), oppure si può usare il meccanismo del *currying* (vedi Sezione 7) per ridurre funzioni con più parametri a funzioni con un solo parametro. Un'altra cosa importante da notare è che in base alla definizione data qui sopra una funzione ha l'unico effetto di calcolare un valore (risultato, o valore di ritorno) in base al valore del parametro.

Non può quindi avere *effetti collaterali* di qualsiasi tipo (vale a dire, non può avere effetti che non siano nel valore di ritorno della funzione).

Come in tutti i linguaggi funzionali, i valori di tipo funzione sono *esprimibili*, vale a dire possono essere generati come risultati di espressioni. In particolare, Haskell utilizza il simbolo "\" per rappresentare le astrazioni del λ calcolo ("\" è un'approssimazione della lettera greca λ) e generare quindi valori di tipo funzione. La sintassi è:

```
\ <param> -> <expression>
```

dove `<param>` è il nome del parametro formale mentre `<expression>` è un'espressione valida, che può utilizzare i nomi presenti nell'ambiente globale più il nome `<param>`. L'espressione \x -> **exp** quando valutata ha quindi come risultato un valore di tipo funzione. Ogni volta che la funzione verrà applicata ad un valore (parametro attuale), tale valore verrà legato al nome `x` (parametro formale) nell'ambiente locale in cui è valutata l'espressione `exp` (il costrutto "\" crea quindi un ambiente locale per la funzione!).

Per esempio,

```
\n -> n + 1
```

è una funzione che incrementa un numero (in questo caso, il parametro formale è `n` e l'espressione da valutare quando viene applicata la funzione è `n + 1`). Una funzione può essere applicata ad un valore facendo seguire il valore del parametro attuale alla funzione. Per esempio,

```
(\n -> n + 1) 5
```

applica la funzione `\n -> n + 1` al valore 5 (le parentesi sono necessarie per indicare l'ordine di precedenza delle operazioni: prima si definisce la funzione e poi si applica al valore 5). Questo significa che il valore 5 (parametro attuale) viene legato al nome `n` (parametro formale) e poi viene valutata l'espressione `n + 1`, che fornisce il valore di ritorno della funzione. Il risultato di questa espressione è ovviamente 6.

Come tutti gli altri valori, anche un valore di tipo funzione può essere associato ad un nome usando il meccanismo di definizione "`=`" di Haskell. Per esempio, il seguente codice definirà una variabile (non modificabile, ricordate!) `incrementa` avente come valore una funzione che somma 1 al valore passato come parametro (in altre parole, assocerà il nome `incrementa` a tale funzione):

```
incrementa = \n -> n + 1
```

Il tipo di questa variabile è `Num a => a -> a` (equivalente alla dizione matematica $f : \mathcal{X} \to \mathcal{X}$, dove $\mathcal{X}$ è un insieme di numeri — $\mathcal{N}$, $\mathcal{Z}$, $\mathcal{R}$, ...). A questo punto è ovviamente possibile applicare la funzione ad un parametro attuale usandone il nome: il risultato di `incrementa 5` è ovviamente 6. Associare nomi simbolici a funzioni (vale a dire: definire variabili di tipo funzione) è particolarmente utile quando la funzione viene usata più volte... Si consideri per esempio la differenza fra

```
(\x -> x+1) ((\x -> x+1) 2)
```

e

```
incrementa (incrementa 2)
```

Per finire, in Haskell esiste una sintassi semplificata per associare nomi a funzioni (questo si può anche vedere come definire variabili di tipo funzione o, in breve, definire funzioni):

```
<name> <param> = <expression>
```

è (circa) equivalente a

```
<name> = \ <param> -> <expression>
```

Per esempio, la funzione `incrementa` è definibile come

```
incrementa n = n + 1;
```

che appare più leggibile rispetto alla definizione basata su "`=`" e "\" (la quale è più simile al λ-calcolo).

Notare infine che questa sintassi non introduce nuove funzionalità ma è soltanto *zucchero sintattico* per semplificare la scrittura di definizioni `... = \...`[6].

[6] In realtà le cose sono leggermente più complicate di questo, in caso di funzioni a più parametri... Ma per il momento non consideriamo questi dettagli.

5 Definizione per Casi

Oltre a poter essere definita tramite un'espressione aritmetica che permette di calcolarne il valore (come appena visto), una funzione può anche essere definita "per casi", specificando esplicitamente il valore del risultato corrispondente ad ogni valore del parametro (o a specifici valori, usando poi un'espressione generica per i casi rimanenti). Questo è generalmente utile per le funzioni definite per induzione (o per le funzioni ricorsive), in cui si distinguono i valori per le basi indittive e per i passi induttivi.

La definizione per casi può essere facilmente implementata usando l'operatore `case`:

```
\x -> case x of
  <pattern_1> -> <expression_1>
  <pattern_2> -> <expression_2>
 ...
 ...
  <pattern_n> -> <expression_n>
```

dove l'espressione **case** x **of** p1 -> e1; p2 -> e2; ... prima valuta l'espressione `x`, poi confronta il valore ottenuto con i *pattern* specificati (`p1`, `p2`, etc...). Non appena il confronto ha successo (si verifica un *match*), viene valutata l'espressione corrispondente assegnando all'espressione `case` il valore risultante.

Un modo semplice per definire "per casi" una funzione è quindi quello di usare valori costanti come pattern per enumerare i possibili valori del parametro. Un esempio di funzione definita per casi (o per enumerazione) è:

```
giorno = \n -> case n of
                 1 -> "Lunedi'"
                 2 -> "Martedi'"
                 3 -> "Mercoledi'"
                 4 -> "Giovedi'"
                 5 -> "Venerdi'"
                 6 -> "Sabato"
                 7 -> "Domenica"
                 _ => "Giorno non valido"
```

notare che lo strano pattern "_" è utilizzato per "catturare" tutti i casi non precedentemente enumerati (numeri interi minori di 1 o maggiori di 7).

Per quanto detto, l'operatore `case` sembra essere l'equivalente a livello di espressioni del comando `case` o `switch` (comando di selezione multipla) esistente in molti linguaggi imperativi. Esiste però un'importante differenza: l'espressione `case` di Haskell (come l'equivalente di molti linguaggi funzionali) permette di utilizzare non solo pattern costanti (come visto nell'esempio precedente), ma anche pattern più complessi contenenti nomi o costrutti come tuple o simili. Nel confrontare il valore di un'espressione `x` con questi pattern "più complessi", Haskell utilizza un meccanismo di *pattern matching*. Se un pattern per esempio contiene dei nomi, quando Haskell confronta l'espressione con tale pattern può creare legami fra nomi e valori in modo che il confronto abbia successo. Per esempio, nel seguente codice

```
f = \a -> case a of
            0 -> 1000.0
            x -> 1.0 / x
```

se il parametro della funzione non è `0` (il primo pattern non matcha) si crea un binding fra `x` ed il valore associato ad `a` (in modo che il secondo pattern possa matchare).

Pattern più complessi basati su coppie o tuple di valori possono essere invece usati (anche senza usare il costrutto "`case`") per definire funzioni in più variabili:

```
somma = \ (a, b) -> a + b
```

In questo caso, quando `somma` viene invocata il meccanismo di pattern matching viene usato per creare un binding fra il nome `a` ed il primo valore della coppia passata come parametro attuale e fra il nome `b` ed il secondo valore.

Tornando alla definizione di funzioni "per casi", è interessante notare come l'espressione

```
f = \x -> case x of
  <pattern_1> => <expression_1>
  <pattern_2> => <expression_2>
```

```
    ...
    ...
    <pattern_n> => <expression_n>
```

possa essere riscritta senza usare il costrutto "`case`". Per fare questo, bisogna prima riscrivere la definizione precedente come

```
f x = case x of
    <pattern_1> => <expression_1>
    <pattern_2> => <expression_2>
    ...
    ...
    <pattern_n> => <expression_n>
```

che può essere semplificata in

```
f <pattern_1> -> <expression_1>
f <pattern_2> -> <expression_2>
 ...
 ...
f <pattern_n> -> <expression_n>;
```

Quest'ultima sintassi è talvolta più chiara, in quanto permette di specificare in modo ancora più esplicito il valore risultato della funzione per ogni valore dell'argomento. Più formalmente, Haskell confronta il valore del parametro attuale con cui la funzione viene invocata con i vari pattern specificati nella definizione `<pattern_1>...<pattern_n>`. Se il parametro attuale matcha `<pattern_1>`, si considera la prima definizione e la funzione viene valutata come `<expression_1>`, se il parametro attuale matcha `<pattern_2>` la funzione viene valutata come `<expression_2>` e così via. In altre parole, *la creazione del legame fra parametro formale (nome) e parametro attuale (valore) nell'ambiente locale della funzione avviene per pattern matching*. Si noti che la definizione "standard" f x = <expression> è un caso particolare di questa forma.

Usando questa sintassi, la definizione della funzione `giorno` presentata in precedenza diventa:

```
giorno 1 = "Lunedi'"
giorno 2 = "Martedi'"
giorno 3 = "Mercoledi'"
giorno 4 = "Giovedi'"
giorno 5 = "Venerdi'"
giorno 6 = "Sabato"
giorno 7 = "Domenica"
giorno _ = "Giorno non valido"
```

Ancora, è importante che i pattern `<expression_1>`, ... `<expression_n>` coprano tutti i possibili valori che la funzione può ricevere come parametri attuali. Per questo il simbolo speciale (_) permette di matchare tutti i valori non coperti dalle clausole precendenti.

Sebbene il meccanismo di pattern matching sia stato introdotto in questa sede per spiegare la definizione di funzioni per casi, è importante capire che questo è un meccanismo estremamente generico, utilizzato in molti altri contesti, che di fare molto più che definire funzioni per casi. In generale, si può dire che il meccanismo di pattern matching è utilizzato ogni qual volta si crei un legame fra un nome ed un valore (anche indipendentemente dalla definizione di funzioni): per esempio, l'espressione

```
x = 2.5
```

crea un legame fra il valore floating point 2.5 ed il nome `x` percreare un match fra il pattern "`x`" con il valore 2.5. Una volta capito questo concetto, è facile intuire come pattern più complessi possano essere utilizzati per creare più legami contemporaneamente; per esempio

```
(x, y) = (4, 5)
```

andrà a legare il nome `x` al valore `4` ed il nome `y` al valore `5` in modo da creare un match fra il pattern `(x, y)` ed il valore coppia $(4, 5)$.

Per definire il concetto di pattern in modo più preciso, si può dire che un pattern può essere:

- un valore costante, che matcha solo con quello specifico valore;

- un *variable pattern* `<var>`, che matcha un qualsiasi valore, dopo aver creato un binding fra il nome `<var>` ed il valore;
- un *tuple pattern* `(<pattern_1>, <pattern_2>, ..., <pattern_n>)`, che compone n pattern più semplici in una tupla. In questo caso si ha match con una tupla di n valori se e solo se ognuno dei valori della tupla matcha col corrispondente pattern del tuple pattern;
- il wildcard pattern _, che matcha qualsiasi valore.

Notare che il valore `()` può essere visto come un esempio di tuple pattern (tupla nulla).

Come noto, un programma scritto secondo il praradigma di programmazione funzionale utilizza il meccanismo della ricorsione per implementare algoritmi che verrebbero implementati tramite iterazione usando un paradigma imperativo. È quindi importante capire come implementare la ricorsione usando Haskell.

Una funzione ricorsiva non è una funzione simile alle altre ha alcun bisogno di essere definita in modo particolare: semplicemnte, l'espressione che implementa il corpo della funzione richiama la funzione stessa. Per esempio,

```
fact = \ n -> if n == 0 then 1 else n * fact (n - 1)
```

che potrebbe essere definita anche come

```
fact n = if n == 0 then 1 else n * fact (n - 1)
```

o addirittura

```
fact 0 = 1
fact n = n * fact (n - 1)
```

6 Controllare l'Ambiente

Come molti dei moderni linguaggi, Haskell utilizza scoping statico: in una funzione, i simboli *non locali* sono risolti (vale a dire: associati ad un valore) facendo riferimento all'ambiente del blocco di codice in cui la funzione è definita (e non all'ambiente del chiamante).

Notare che il costrutto lambda (o la "sintassi abbellita" per definire funzioni) crea un blocco di annidamento statico (come i simboli { e } in C). All'interno di questo blocco viene aggiunto un nuovo binding fra il simbolo che identifica il parametro formale ed il valore del parametro attuale con cui la funzione verrà invocata. Questo nuovo binding può introdurre un nuovo simbolo o mascherare un binding esistente. Per esempio, una definizione "v = 1" crea un legame fra il simbolo "`v`" ed il valore "`1`" nell'ambiente globale. Una definizione "f = \v -> 2 * v" crea un blocco di annidamento (contenente l'espressione "2 * v") dentro il quale il simbolo "`v`" non è più associato al valore "`1`", ma al valore del parametro attuale con cui "`f`" sarà invocata. Quindi, "`f 3`" ritornerà 6, non 2.

I simboli non locali (notare che per quanto detto fino ad ora i simboli locali sono solo i parametri della funzione) vengono cercati nell'ambiente attivo *quando la definizione della funzione viene valutata* (e non quando la funzione viene invocata) e possono essere risolti in tale momento. Per esempio, una dichiarazionef = \ x =>x + y; risulterà in un errore se quando tale dichiarazione viene processata il simbolo `y` non è legato ad alcun valore.

Haskell mette anche a disposizione due meccanismi per creare blocchi di annidamento statici e modificare l'ambiente al loro interno (senza che l'ambiente esterno al blocco risulti modificato): uno per creare blocchi di annidamento contenenti espressioni (**let** <definizioni> **in** <espressione>) ed uno per modificare l'ambiente all'interno di una definizione (<definizione> **where** <definizione>).

In altre parole, "**let** <definizioni> **in** <espressione>" è un'espressione valutata al valore di `<espressione>` dopo che l'ambiente è stato modificato aggiungendo i binding definiti in `<definizioni>`. Tali binding sono utilizzati per valutare l'espressione e sono poi rimossi dall'ambiente immediatamente dopo la valutazione. Per esempio, il costrutto `let` è utilizzabile per implementare una versione *tail recursive* della funzione fattoriale. Si ricordi che una funzione è tail recursive se utilizza solo chiamate ricorsive *in coda*: la tradizionale funzione fact = \n -> **if** n == 0 **then** 1 **else** fact (n − 1) ∗ n non è tail recursive perché il risultato di `fact(n - 1)` non è immediatamente ritornato, ma deve essere moltiplicato per `n`. Una versione tail recursive della funzione fattoriale utilizza un secondo parametro per memorizzare il risultato parziale: fact_tr = \n =>\res =>**if** n == 0 **then** res **else** fact_tr (n − 1) (n ∗ res). Questa funzione riceve quindi due argomenti, a differenza della funzione `fact` originale. E' quindi necessario un wrapper

```
fact = \ n ->
        let
            fact_tr = \ n -> \ res ->
                if n == 0 then
                    res
                else
                    fact_tr (n - 1) (n * res)
        in
            fact_tr n 1
```

Figure 1: Implementazione della funzione fattoriale con ricorsione in coda in Haskell, nascondendo la funzione `fact_tr` tramite costrutto `let`.

```
fact = \n -> fact_tr n 1
    where fact_tr = \n -> \res -> if n == 0
            then
                res
            else
                fact_tr (n - 1) (n * res)
```

Figure 2: Implementazione della funzione fattoriale con ricorsione in coda in Haskell, nascondendo la funzione `fact_tr` tramite il costrutto `where`.

che invochi `fact_tr` con i giusti parametri: fact = \n =>fact_tr n 1. Una soluzione di questo genere ha però il problema che `fact_tr` è visibile non solo a `fact` (come dovrebbe essere), ma nell'intero ambiente globale. Ll costrutto `let` permette di risolvere questo problema, come mostrato in Figura 1.

Il costrutto "<definizione1> **where** <definizione2>" permette invece di utilizzare i binding definiti da `<definizioe2>` in `<definizione1>`, ripristinando poi l'ambiente originario.

L'utilità di `where` può essere capita meglio considerando il seguente esempio: si supponga di voler implementare in Haskell una funzione $f : \mathcal{N} \to \mathcal{N}$, anche se Haskell non supporta il tipo `unsigned int` (corrispondente ad $\mathcal{N}$) ma solo il tipo `Int` (corrispondente a $\mathcal{Z}$). Per sopperire a questa limitazione, si può definire una funzione `integer_f` che implementa $f()$ usando `Int` come dominio e codominio, richiamandola da una funzione `f` che controlla se il valore del parametro è positivo o no, ritornando -1 in caso di argomento negativo:

```
    f = \n -> if n < 0 then -1 else integer_f n
  where
    integer_f = \n -> ...
```

Questa soluzione permette di evitare di esportare a tutti la funzione `integer_n`, che accetta anche argomenti negativi senza fare controlli.

Analogamente, `where` può essere usato per "nascondere" la funzione `fact_tr` a due argomenti nella definizione tail recursive del fattoriale (vedere esempio precedente in Figura 1), come mostrato in Figura 2.

Confrontando i due esempi di Figura 1 e 2, è facile capire come esista una stretta relazione fra il costrutto `let` ed il costrutto `where` e come possa essere sempre possibile usare `let` al posto di `where` (spostando le definizioni dal blocco `in` all'esterno del costrutto).

7 Funzioni che Lavorano su Funzioni

Poiché un linguaggio funzionale fornisce il tipo di dato funzione e la possibilità di vedere funzioni come valori denotabili (gestiti in modo analogo a valori di altri tipi più "tradizionali" come interi e floating point), è possibile definire funzioni che accettano valori di tipo funzione come parametro e quindi lavorano su funzioni. In modo analogo, il valore di ritorno di una funzione può essere a sua volta di tipo funzione. Da questo punto di vista, le funzioni che accettano funzioni come parametri e ritornano valori di tipo funzioni non sono dissimili da funzioni che accettano e ritornano (per esempio) valori di tipo intero. Ci sono però alcuni dettagli che meritano di essere considerati con più attenzione e che motivano l'esistenza di questa sezione.

Per analizzare alcune peculiarità interessanti delle funzioni che lavorano su funzioni (spesso citate come "high-order functions" in letteratura) consideriamo un semplice esempio basato sul calcolo della derivata (o meglio, di una sua approssimazione, tramite rapporto incrementale sinistro) di una funzione $f : \mathcal{R} \to \mathcal{R}$. Cominciando definendo una funzione `calcoladerivata` che accetta come parametro la funzione f di cui calcolare la derivata ed un numero $x \in \mathcal{R}$. La funzione `calcoladerivata` ritorna un'approssimazione della derivata di f calcolata nel punto x. Tale funzione, che ha un parametro di tipo funzione ed uno di tipo floating point e ritorna un valore di tipo floating point, può essere implementata, per esempio, come:

```
calcoladerivata = \ (f, x) -> (f(x) - f(x - 0.001)) / 0.001
```

(si tralasci il fatto che la derivata è stata approssimata col rapporto incrementale sinistro con passo $\epsilon = 0.001$).

È da notare come Haskell sia in grado di inferire il tipo dei parametri `x` (che risulta essere un tipo "`a2`" appartenente alla classe "`Fractional`", quindi un numero floating point, a causa dell'espressione `x - 0.001`) ed `f` (che risulta essere una funzione da un tipo "`a1`" appartenente alla classe "`Fractional`" al tipo "`a2`" — sempre appartenente alla classe "`Fractional`" — poiché `f` viene applicata ad `x` ed il suo valore di ritorno viene diviso per `0.001`). Il tipo di `calcoladerivata` sarà quindi "`(Fractional a1, Fractional a2) => (a2 -> a1, a2) -> a1`", dove appunto `a1` ed `a2` sono type variable col vincolo di rappresentare tipi della classe "`Fractional`" (numeri floating point).

L'esempio presentato non è comunque sorprendente, perché qualcosa di simile alla funzione `calcoladerivata` presentata qui sopra si può facilmente implementare anche usando un linguaggio prevalentemente imperativo (per esempio, usando il linguaggio C si può usare un puntatore a funzione come primo parametro, invece di f). Una cosa invece notevolmente più difficile da implementare con linguaggi non funzionali è una funzione `derivata` che riceva in ingresso solo la funzione $f()$ (e non il punto `x` in cui calcolare la derivata) e ritorni una funzione (e non un numero reale) che approssima la derivata di f. Questa funzione `derivata` ha un unico parametro, di tipo `(Fractional a1, Fractional a2) => a1 -> a2` (ad indicare che `a1` ed `a2` sono tipi floating point) e ritorna un valore di tipo `a1 -> a2`. Il tipo di questa funzione sarà quindi `(Fractional a1, Fractional a2) => (a2 -> a1) -> a2 -> a1` ed una sua possibile implementazione in Haskell è la seguente:

```
derivata = \f -> (\x -> (f x - f (x - 0.001)) / 0.001)
```

Cerchiamo di capire meglio questa definizione di `derivata`: derivata = \f -> sostanzialmente dice che il nome "`derivata`" è associato ad una funzione che ha come parametro "`f`". L'espressione che definisce come calcolare tale funzione è \x -> (f x − f (x − 0.001)) / 0.001 (alcune parentesi sono state aggiunte nell'esempio di qui sopra per aumentarne la leggibilità), che indica una funzione della variabile `x` calcolata come $\frac{fx - f(x-0.001)}{0.001}$. Quindi, il valore di ritorno della funzione `derivata` è una funzione (del parametro `x`, che Haskell può identificare come floating point a causa dell'espressione $x - 0.001$). Tale funzione è calcolata in base alla funzione `f`, che è parametro di `derivata`. Valutando la definizione, Haskell può inferire il tipo di `f` (funzione da floating point a floating point).

Come ulteriore considerazione, si può notare come la funzione `derivata` possa essere vista come una versione "curryficata" della funzione `derivata`. Sostanzialmente, invece di invocare la funzione passandole 2 argomenti (la funzione `f` di cui calcolare la derivata ed il punto `x` in cui calcolare la derivata), si invoca passando solo il primo argomento `f`... Se invocata con 2 argomenti, la funzione ritorna un numero reale (il valore della derivata di `f` nel punto `x`); quindi, se applicata ad un unico parametro `f` la funzione non può ritornare un reale, ma ritornerà un "oggetto" che può diverntare un numero reale quando applicato ad un ulteriore parametro `x` (di tipo reale)... Questo "oggetto" è quindi una funzione $f' : \mathcal{R} \to \mathcal{R}$.

Ricordiamo che l'idea fondamentale del currying è (esprimendosi in modo informale) che una funzione di due parametri x e y è equivalente ad una funzione del parametro x che ritorna una funzione del parametro y. Quindi, il meccanismo del currying permette di esprimere una funzione in più variabili come una funzione in una variabile che ritorna una funzione delle altre variabili. Per esempio, una funzione $f : \mathcal{R}x\mathcal{R} \to \mathcal{R}$ che riceve due parametri reali e ritorna un numero reale può essere riscritta come $f_c : \mathcal{R} \to (\mathcal{R} \to \mathcal{R})$.

Per esempio, la funzione `somma` che somma due numeri

```
somma = \ (x, y) -> x + y
```

può essere scritta tramite currying come

```
sommac = \x -> (\y -> x + y)
```

Per capire meglio tutto questo si può confrontare la funzione $f(x,y) = x^2 + y^2$ con la sua versione ottenuta attraverso il currying $f_c(x) = f_x(y) = x^2 + y^2$. Si noti che $f()$ ha dominio $\mathcal{R}^2$ e codominio $\mathcal{R}$, mentre $f_c()$ ha dominio $\mathcal{R}$ e codominio $\mathcal{R} \to \mathcal{R}$. In Haskell queste funzioni sono definite come

```
f  = \ (x, y)  -> x * x + y * y
fc = \x -> (\y -> x * x + y * y)
```

`fc` permette di fare applicazioni parziali, tipo `val g = f 5`, che definisce una funzione $g(y) = 25 + y^2$, mentre `f` non permette nulla di simile.

Il currying è direttamente supportato a livello sintattico da Haskell, che fornisce una sintassi semplificata per definire funzioni "curryficate":

```
f a b = exp;
```

è equivalente a

```
f = \a -> \b -> exp;
```

Usando questa sintassi semplificata, le definizioni di `f` ed `fc` (esempio precedente) diventano quindi

```
f (x, y) = x * x + y * y;
fc x y   = x * x + y * y;
```

mentre la definizione di `derivata` diventa

```
derivata f x = (f(x) - f(x - 0.001)) / 0.001;
```

8 Ancora sui Tipi di Dato

Fino ad ora abbiamo lavorato coi tipi di dato predefiniti di Haskell: `()`, `Bool`, `Int`, `Float`, `Double`, `Char` e `String` (anche se il tipo `()` non è stato molto utilizzato... In pratica, è utile prevalentemente per modellare funzioni "strane" che non accettano argomenti o che ritornano solo effetti e non valori).

Tutti i tipi riconosciuti da Haskell (siano essi "tipi semplici" come i tipi primitivi appena citati, o tipi più complessi definiti dall'utente) possono essere aggregati in vario modo per generare tipi più complessi. Il modo più semplice per comporre tipi è formando delle tuple (definite sul prodotto cartesiano dei tipi che le compongono). Per esempio, abbiamo già visto come una funzione a più argomenti fosse definibile come una funzione che ha come unico argomento una tupla composta dai vari argomenti:

```
sommaquadrati = \ (a,b) -> a * a + b * b
```

Questa funzione ha tipo "**Num** a =>(a, a) −> a" che significa "funzione da una coppia di elementi del tipo `a` ad un elemento del tipo `a`, dove `a` è un tipo che rappresenta un numero (`Int`, `Float`, `Double`)"; in altre parole, l'unico parametro della funzione è di tipo "(a,a)", che rappresenta il prodotto cartesiano di un insieme numerico per se stesso (per esempio, $\mathcal{N} \times \mathcal{N}$).

Più formalmente parlando, una tupla è un insieme ordinato di più valori, ognuno dei quali avente un tipo riconosciuto da Haskell. L'insieme di definizione di una tupla è dato dal prodotto cartesiano degli insieme di definizione dei valori che la compongono. Ancora, notare come il tipo `()` (o unit, avente un unico valore `()`) possa essere visto come una tupla di 0 elementi e come le tuple di 1 elemento corrispondano con l'elemento stesso. Per esempio, `(6)` è equivalente a `6` ed ha tipo `Num p => p`; quindi, l'espressione "(6)" viene valutata come "6" e l'espressione "(6) == 6" viene valutata come `True`.

Come già visto, il meccanismo del pattern matching applicato alle tuple permette di creare binding fra più nomi e simboli contemporaneamente: "coppia=("pi_greco", 3.14)" crea un legame fra il simbolo "`coppia`" e la coppia "("pi_greco", 3.14)" (avente tipo "`Fractional b => ([Char], b)`"). Invece, "(pigreco, **pi**) = coppia" crea un legame fra il simbolo "`pigreco`" e la stringa "`"pi greco"`" e fra il simbolo "`pi`" ed il numero "`3.14`" (di un tipo Fractional). Questo stesso meccanismo può essere usato per passare parametri multipli ad una funzione usando una tupla come unico argomento.

Haskell permette anche di definire nuovi tipi di dato, usando la keyword "`data`" che permette di far riconoscere al compilatore dati i cui valori non sono precedentemente conosciuti. Il modo più semplice per usare `data` è la definizione dell'equivalente di un tipo enumerativo; per esempio

```
data Currency = Eur | Usd | Ounce_gold
```

```
data Currency = Eur | Usd | Ounce_gold
data Money = Euros Float | Usdollars Float | Ounces_gold Float

convert (amount, to) =
    let toeur (Euros x)        = x
        toeur (Usdollars x)    = x / 1.05
        toeur (Ounces_gold x) = x * 1113.0
    in
        (case to of
            Eur         -> toeur amount
            Usd         -> toeur amount * 1.05
            Ounce_gold -> toeur amount / 1113.0
        , to)
```

Figure 3: Conversione di denaro fra valute, usando un nuovo tipo di dato per rappresentare le varie valute. Questo esempio mostra come fare pattern matching sui tipi di dato definiti dall'utente.

```
data Currency = Eur | Usd | Ounce_gold
data Money = Euros Float | Usdollars Float | Ounces_gold Float

convert (amount, to) =
    let toeur (Euros x)        = x
        toeur (Usdollars x)    = x / 1.05
        toeur (Ounces_gold) x = x * 1113.0
    in
        case to of
            Eur         -> Euros        (toeur amount)
            Usd         -> Usdollars    (toeur amount * 1.05)
            Ounce_gold -> Ounces_gold (toeur amount / 1113.0)
```

Figure 4: Conversione di denaro fra valute, versione finale.

definisce tre nuovi valori ("`Eur`", "`Usd`" ed "`Ounce_gold`") che non erano precedentemente riconosciuti: prima di questa definizione, provando ad associare il valore "`Eur`" ad un nome (con "c = Eur") si ottiene un errore. Ma dopo la definizione, "c = Eur" ha successo ed a "`c`" viene associato un valore di tipo "`Currency`".

In una definizione di questo tipo, il simbolo "`|`" rappresenta una "o" a significare che una variabile di tipo "`Currency`" può avere valore "`Eur`" oppure "`Usd`" oppure "`Ounce_gold`". Si noti che nomi separati da "`|`", che rappresentano valori costanti che stiamo definendo, devono iniziare con una lettera maiuscola, come il nome del tipo. Tecnicamente, essi sono dei *costruttori di valore*, vale a dire funzioni che ritornano (costruiscono) valori del nuovo tipo che stiamo definendo. In questo caso (definizione di un tipo equivalente ad un tipo enumerativo), i costruttori non hanno alcun argomento e sono quindi dei costruttori costanti; esiste però la possibilità di definire costruttri che generano un valore a partire da un argomento.

Per esempio, si potrebbe pensare di definire un tipo `Money` come segue:

```
data Money = Euros Float | Usdollars Float | Ounces_gold Float
```

(notare che qui si sono usati "`Euros`, "`Usdollars`" ed "`Ounces_gold`" per fare si che i nomi di costruttori del tipo "`Money`" fossero diversi dai nomi di costruttori del tipo "`Currency`"...).

Come tutte le funzioni di Haskell, anche questi costruttori di valore hanno un solo argomento (e come al solito si può ovviare a questa limitazione usando una tupla come argomento).

In questo caso, "`Euros`" non rappresenta un valore (costante) del tipo "`Money`" (come accadeva invece con "`Eur`" per il tipo "`Currency`"), ma è una funzione da numeri `Float` a valori di tipo `Money` (in Haskell, "**Float** −> Money"). I valori di tipo `Money` sono quindi "`Euros 0.5`" o simili.

La Figura 3 mosta come una funzione "`convert`" che converte valori fra diverse valute sia implementabile usando i due nuovi tipi di dato appena descritti, facendo pattern matching su valori di tipo "`Currency`" in "`toeur`" e facendo pattern matching su valori di tipo "`Money`" nel corpo di "`convert`". Notare però che l'utilizzo di questi nuovi tipi di dato è ancora parziale, perché questa versione di "`convert`"

riceve una coppia in cui il primo elemento è di tipo "`money`" ed il secondo è di tipo "`currency`", ma ritorna una coppia (reale, currency) invece che un valore di tipo "`money`". L'implementazione di Figura 4 risolve questo ultimo problema.

Quando si usa "`data`" per definire un nuovo tipo di dato, è importante ricordare che definire un tipo semplicemente specificandone l'insieme di definizione (insieme dei valori di tale tipo) non è abbastanza. Bisogna anche definire come operare su valori di tale tipo, definendo funzioni che operano sul tipo che si va a definire (tornando all'esempio precedente, definire i tipi "`Currency`" e "`Money`" senza definire la funzione "`convert`" non è molto utile...).

9 Lavorare con Haskell

Una macchina astratta per il linguaggio Haskell (vale a dire, una macchina astratta in grado di capire ed eseguire programmi scritti in Haskell) può essere implementata tramite interpreti o compilatori (meglio: può essere implementata in modo prevalentemente interpretato o prevalentemente compilato). Sebbene esistano diversi compilatori o interpreti per Haskell, in questa sede ci concetreremo sul Glasgow Haskell Compiler (ghc) ed in particolare sulla sua versione interattiva, ghci, che implementa un ciclo read-evaluate-print (Real Evaluate Print Loop — REPL) per Haskell. Questo perché un programma interattivo come `ghci` è inizialmente più intuitivo e semplice da usare (evitandoci, almeno per un primo momento, di dover considerare concetti più complessi come I/O Monad o simili). Questo significa che operativamente un utente può interagire direttamente con il REPL tramite un prompt e testare in modo veloce ed intuitivo i primi comandi che vedremo. Il REPL pu'o essere invocato digitando il comando "`ghci`", che risponde come segue:

```
luca@nowhere   $ ghci
GHCi, version 8.6.5:  http://www.haskell.org/ghc/ :?  for help
Prelude>
```

Notare che `ghci` presenta il suo prompt (caratterizzato dal simbolo "`Prelude>`") e rimane in attesa che vengano immesse espressioni da valutare.

Come già spiegato, eseguire un programma funzionale significa valutare le espressioni che lo compongono (vedi Sezione 1); quindi, un REPL Haskell può essere inizialmente visto come un valutatore interattivo di espressioni.

Le espressioni immesse tramite il prompt vengono valutate da `ghci` man mano che l'utente le immette; ogni volta che `ghci` valuta un'espressione, ne visualizza a schermo il valore risultante. Per esempio, digitando

```
5
```

si ottiene risposta

```
5
```

indicante che l'espressione è stata valutata al valore 5. Chiaramente, si possono valutare anche espressioni più complesse, usando gli operatori aritmetici che conosciamo e la notazione infissa (con cui siamo familiari):

```
Prelude> 5 + 2
7
Prelude> 2 * 3 * 4
24
Prelude> (2 + 3) * 4
20
```

e così via.

Notare che il tasto "Return" (o "Enter") serve da *terminatore* indicando a ghci la fine di un'espressione. Quindi, un'espressione non viene compilata/valutata fino a che non si incontra un ritorno a capo; d'altra parte, un ritorno a capo comporta una valutazione immediata dell'espressione e questo potrebbe comportare problemi con espressioni multi-linea. Per esempio, il seguente codice Haskell dichiara e definisce una variabile intera chiamata "`a`" inizializzandola col valore 5:

```
a :: Int
a = 5
```

Ma se si prova ad immettere queste linee nel prompt del REPL di `ghci` si ottiene:

```
Prelude> a :: Int

<interactive>:4:1: error: Variable not in scope: a :: Int
```

perché `ghci` prova a valutare "`a :: Int`" prima che "`a = 5`" sia stato immesso. Una possibile soluzione è quella di immettere dichiarazione e definizione nella stessa linea separandole con un "`;`":

```
Prelude> a :: Int; a = 5
Prelude> a
5
Prelude> :t a
a :: Int
```

da questo esempio si vede anche che il comando `ghci` "`:t`" permette di vedere il tipo del valore associato ad un simbolo. Un altro modo per risolvere il problema potrebbe essere stato quello di separare dichiarazione e definizione su due righe diverse, ma racchiuderle fra i simboli "`:{`" e "`:}`":

```
ghci> :{
ghci| a :: Int
ghci| a = 5
ghci| :}
ghci> a
5
ghci> :t a
a :: Int
```

In pratica, quando un blocco di codice è racchiuso fra "`:{`" e "`:}`" `ghci` non lo compila/valuta ad ogni Return, ma attende la chiusura del blocco prima di compilare il codice.

Se invece si fosse evitata la dichiarazione del tipo di `a`, il risultato sarebbe stato:

```
a = 5
Prelude> a
5
Prelude> :t a
a :: Num p => p
```

ad indicare che il valore associato ad `a` è un numero, vale a dire il suo tipo appartiene alla classe `Num` (`Int`, `Float`, `Double`). Da questo esempio si comincia a vedere come data un'espressione, il compilatore possa essere in grado di inferire il tipo delle variabili (ma anche degli argomenti e del valore di ritorno delle funzioni). A differenza di altri linguaggi come Standard ML, Haskell utilizza il suo sistema di tipi polimorfico per dare maggiore flessibilità senza fare conversioni automatiche di tipo. Per questo motivo, espressioni come

```
5 + 2.0
```

sono valide:

```
Prelude> :t 5
5 :: Num p => p
Prelude> :t 2.0
2.0 :: Fractional p => p
Prelude> 5 + 2.0
7.0
Prelude> :t 5 + 2.0
5 + 2.0 :: Fractional a => a
```

Questo indica che `5` è un generico numero, mentre `2.0` è un numero floating point. Il risultato della somma sarà quindi un numero floating point (tipo appartenente alla classe `Fractional`). L'operatore `+` (somma) accetta come parametri due valori dello stesso tipo, purcé questo tipo appartenga alla classe `Num`. Poiché la classe `Fractional` è un sottoinsieme della classe `Num`, la somma è possibile ed il suo risultato ha un tipo che appartiene alla classe `Fractional`.

Come precedentemente detto, Haskell fornisce vari tipi di base:

```
Prelude> :t 2
2 :: Num p => p
```

```
Prelude> :t 2.0
2.0 :: Fractional p => p
Prelude> :t 2 > 1
2 > 1 :: Bool
Prelude> :t "abc"
"abc" :: [Char]
Prelude> :t 'a'
'a' :: Char
```

Siamo ora pronti a fare la prima cosa che viene fatta quando si incontra un nuovo linguaggio di programmazione:

```
> "Hello, " ++ "world"
"Hello, world"
```

notare che `"Hello, "` e `"world"` sono due valori di tipo `String` (che è una lista di caratteri "`[Char]`"), mentre "++" è l'operatore di concatenazione fra stringhe, che riceve in ingresso due parametri di tipo stringa e genera una stringa che rappresenta la loro concatenazione.

Fino ad ora abbiamo visto come usare `ghci` per valutare espressioni in cui tutti i valori sono espressi esplicitamente. Ricordiamo però che uno dei primi tipi di *astrazione* che abbiamo visto consiste nell'assegnare **nomi** ad "entità" (termine informale) riconosciute da un linguaggio. Vediamo quindi come sia possibile associare nomi ad entità in haskell e quali siano le entità denotabili in haskell (in un linguaggio imperativo, tali entità denotabili erano variabili, funzioni, tipi di dato, ...). Come in tutti i linguaggi funzionali di alto livello, anche in haskell esiste il concetto di *ambiente/environment* (una funzione che associa nomi a valori denotabili) ma non esiste il concetto di memoria (funzione che associa ad ogni variabile il valore che essa contiene)[7]; quindi, è possibile associare nomi a valori, ma non è possibile creare variabili modificabili:

```
Prelude> n = 5
Prelude> n
5
Prelude> :t n
n :: Num p => p
```

Questi comandi associano il valore `5` al nome `n`, stampano il valore associato ad `n` e stampano il tipo di `n` (meglio, la classe di tipi a cui appartiene). Chiaramente, è possibile usare qualsiasi tipo di espressione per calcolare il valore a cui associare un nome:

```
Prelude> x = 5.0 + 2.0
Prelude> n = 2 * 3 * 4
```

Dopo aver associato un nome ad un valore, è possibile usare tale nome (invece del valore) nelle successive espressioni:

```
Prelude> x = 5.0 + 2.0
Prelude> y = x * 2.0
Prelude> x > y
False
```

notare che "y = x * 2" non avrebbe generato un errori... Perché?

Un secondo livello di astrazione consiste nel definire *funzioni*, associando nomi a blocchi di codice. Mentre nei linguaggi imperativi questo è un concetto diverso rispetto alla definizione di variabili, nei linguaggi funzionali (e non solo) esiste un tipo di dato "funzione", generato da espressioni del tipo $\lambda x.e$ (astrazione del λ calcolo). In particolare, haskell utilizza il simbolo "\" al posto del simbolo "λ" e "`->`" invece di ".":

```
Prelude>:t \x -> x + 1
\x -> x + 1 :: Num a => a -> a
```

in questo caso il valore risultante dalla valutazione dell'espressione è una funzione (nel senso matematico del termine) da un tipo numerico "`a`" a se stesso. Poiché non si è usata alcuna definizione, a tale funzione non è stato dato un nome (si parla di *funzione anonima*). Una funzione può però essere applicata a dei dati anche senza darle un nome:

[7] Teoricamente, il concetto di memoria / variabile modificabile esiste anche in haskell, ma noi non ce ne occuperemo.

```
Prelude> (\x -> x + 1) 5
6
```

Anche per le funzioni, `ghci` è generalmente in grado di inferire correttamente i tipi di dato:

```
Prelude> :t \x -> x + 1
\x -> x + 1 :: Num a => a -> a
Prelude> :t \x -> x + 1.0
\x -> x + 1.0 :: Fractional a => a -> a
```

A questo punto, dovrebbe essere chiaro come associare un nome ad una funzione, tramite quella che in altri linguaggi verrebbe chiamata definizione di funzione e che in haskell corrisponde ad una semplice definizione di variabile. Tecnicamente, il seguente codice definisce una variabile "`doppio`" di tipo funzione da numeri a numeri:

```
Prelude> doppio = \n -> 2 * n
Prelude> :t doppio
doppio :: Num a => a -> a
Prelude> doppio 9
18
Prelude> doppio 4.0
8.0
```

Haskell fornisce anche una sintassi semplificata per definire le funzioni:

```
Prelude> doppio n = 2 * n
Prelude> doppio 9
18
```

La sintassi semplificata `name a = ...` appare più intuitiva della definizione esplicita `name = \a -> ...`, ma è equivalente ad essa.

Componendo quanto visto fin'ora con l'espressione `if`, è possibile definire funzioni anche complesse (equivalenti ad algoritmi iterativi implementati con un linguaggio imperativo) tramite ricorsione. Per esempio,

```
Prelude> fact = \n -> if n == 0 then 1 else n * fact (n - 1)
Prelude> fact 5;
120
```

Usando la sintassi semplificata si possono invece definire funzioni ricorsive anche per casi, distinguendo la base induttiva dal passo induttivo:

```
Prelude> :{
Prelude| fact 0 = 1
Prelude| fact n = n * fact (n - 1)
Prelude| :}
Prelude> fact 4
24
```

Con quanto visto fino ad ora, dovrebbe a questo punto essere semplice implementare in haskell funzioni ricorsive che effettuano le seguenti operazioni:

- Calcolo del fattoriale di un numero *usando ricorsione in coda* (Figura 5)
- Calcolo del massimo comun divisore fra due numeri (Figura 6)
- Soluzione del problema della torre di Hanoi (Figure 7 e 8)

Come spiegato precedentemente in questo documento, oltre a permettere di definire e valutare espressioni (associando eventualmente espressioni o valori a nomi, tramite il concetto di ambiente), Haskell permette di definire ed utilizzare nuovi *tipi di dato*. In particolare, il costrutto `data` permette di definire nuovi tipi di dato specificandone i costruttori che ne generano le varianti. Per esempio, si può definire un tipo `Colore`, che rappresenta una componente di rosso, verde o blu (con l'intensità espressa come numero reale):

```
fact_tr = \n -> \res -> if n == 0 then res else fact_tr (n - 1) (n * res)
fact = \n -> fact_tr n 1

fact_tr1 0 res = res
fact_tr1 n res = fact_tr1 (n - 1) (res * n)
fact1 n = fact_tr1 n 1
```

Figure 5: Fattoriale con ricorsione in coda.

```
gcd = \a -> \b -> if b == 0 then a else gcd b (a `mod` b)

gcd1 a b = if b == 0 then a else gcd1 b (a `mod` b)

gcd2 a 0 = a
gcd2 a b = gcd2 b (a `mod` b)
```

Figure 6: Massimo Comun Divisore.

```
Prelude>:t Rosso

<interactive>:1:1: error: Data constructor not in scope: Rosso
Prelude> :t Rosso 0.5

<interactive>:1:1: error:
    Data constructor not in scope: Rosso :: Double -> t
Prelude> data Colore = Rosso Float | Blu Float | Verde Float
Prelude> :t Rosso
Rosso :: Float -> Colore
Prelude> :t Rosso 0.5
Rosso 0.5 :: Colore
```

Prima della definizione "**data** Colore = Rosso **Float** Blu Float — Verde Float—" la parola "`Rosso`" non viene riconosciuta da `ghci`, che lamenta l'utilizzo di un costruttore (`Rosso`) che non è stato definito (vedere i primi due errori); dopo la definizione la parola "`Rosso`" viene correttamente riconosciuta come un costruttore del tipo "`Colore`" (funzione da `Float` a `Colore`).

Per finire, va notato come `ghci` possa sembrare poco pratico da usare, perché immettere programmi multi-linea non è semplicissimo, anche usando "`:{`" e "`:}`". Questo problema può essere risolto editando programmi Haskell complessi in file di testo (generalmente con estensione `.hs`), in modo da poter utilizzare le funzionalità di editor avanzati come `emacs`, `vi`, `gedit` o simili. Un programma contenuto in un file di testo può essere poi caricato da `ghci` usando la direttiva "`:l`": "**Prelude**> :l <file>.hs"

Come esempio di utilizzo di `:l`, si può inserire il programma di Figura 8 nel file `hanoi.hs`, usando il poprio editor di testo preferito (per esempio `vi`). Il programma può poi essere caricato in `ghci` con

```
Prelude> :l hanoi.hs
[1 of 1] Compiling Main             ( hanoi.hs, interpreted )
Ok, one module loaded.
*Main> putStrLn  (move 3 "Left" "Right" "Center")
Move disk from Left to Right
Move disk from Left to Center
Move disk from Right to Center
Move disk from Left to Right
Move disk from Center to Left
Move disk from Center to Right
Move disk from Left to Right
```

Come mostrato nell'esempio precedente, la funzione `move` è adesso definita come se fosse stata inserita direttamente da tastiera (a proposito dell'esempio, si noti come la funzione "`putStrLn`" sia stata usata per visualizzare la stringa, in modo da gestire correttamente i caratteri di ritorno a capo).

```
move n from to via =
    if n == 0
      then
        "\n"
      else (move (n - 1) from via to) ++
        "Move disk from " ++
        from ++ " to " ++ to ++
        (move (n - 1) via to from)
```

Figure 7: Torre di Hanoi.

```
move n from to via =
    if n == 1
      then
        "Move disk from " ++ from ++ " to " ++ to ++ "\n"
      else
        (move (n - 1) from via to) ++
        (move 1 from to via) ++
        (move (n - 1) via to from)
```

Figure 8: Torre di Hanoi, versione alternativa.

L'utilizzo di `:l` è anche utile per il debugging, perché segnala con più precisione gli errori sintattici, indicando la linea del programma in cui si è verificato l'errore.