Piccola Introduzione alla Programmazione Funzionale

Luca Abeni

30 dicembre 2022

Indice

1 Per Programmatori Imperativi **5**
1.1 Stili e Paradigmi di Programmazione . . . 5
1.2 Ricorsione ed Iterazione . . . 7
1.3 Ricorsione e Stack . . . 12
1.4 Funzioni e Spezie . . . 17
1.5 Linguaggi di Programmazione Funzionale . . . 21
1.6 Computazione per Riduzione . . . 23
1.7 Un Linuguaggio Funzionale Minimale . . . 24

2 Primo Incontro col Lambda Calcolo **27**
2.1 Introduzione al Lambda-Calcolo . . . 27
2.2 Semantica del Lambda Calcolo . . . 29
2.3 Codifica di Linguaggi di Alto Livello . . . 31
2.4 Rimozione delle Astrazioni . . . 35
2.5 Lambda Calcolo con Tipi . . . 37

3 Linguaggi di Programmazione Funzionale for Dummies **41**
3.1 Introduzione . . . 41
3.2 Tipi ed Espressioni in Standard ML . . . 42
3.3 Associare Nomi a Valori . . . 43
3.4 Funzioni . . . 44
3.5 Definizione per Casi . . . 45
3.6 Ricorsione . . . 47
3.7 Controllare l'Ambiente . . . 48
3.8 Funzioni che Lavorano su Funzioni . . . 49
3.9 Ancora sui Tipi di Dato . . . 51
3.10 Cenni di Programmazione Modulare . . . 56
3.11 Lavorare con Standard ML . . . 58
3.12 Tipi ed Espressioni in Haskell . . . 65
3.13 Associare Nomi a Valori . . . 67
3.14 Funzioni . . . 68
3.15 Definizione per Casi . . . 69
3.16 Controllare l'Ambiente . . . 72
3.17 Funzioni che Lavorano su Funzioni . . . 73
3.18 Ancora sui Tipi di Dato . . . 75
3.19 Lavorare con Haskell . . . 76
3.20 Qualche Nota sul C++ . . . 82
3.21 Variabili in C++ . . . 82
3.22 Funzioni come Valori in C++ . . . 83
3.23 Algebraic Data Types in C++ . . . 86

4 Tipi di Dato Ricorsivi **89**
4.1 Tipi di Dato . . . 89
4.2 Ricorsione . . . 90
4.3 Tipi di Dati Ricorsivi . . . 91
4.4 Liste Immutabili e Non . . . 93

5 Fixpoint ed Altre Amenità **99**
5.1 Fixed Point Combinator 99
5.2 Implementazione in SML 100
5.3 Implementazione in Haskell 102
5.4 Come Derivare l'Y Combinator 105

A Un po' di Definizioni **109**
A.1 Identificatori, Legami ed Ambiente 109
A.2 Variabili Modificabili 110
A.3 Entità Denotabili, Esprimibili e Memorizzabili 110
A.4 Funzioni 111
A.5 Chiusure 113
A.6 Chiusure e Classi 114

Capitolo 1

Per Programmatori Imperativi

1.1 Stili e Paradigmi di Programmazione

La programmazione funzionale è uno stile (o un *paradigma*) di programmazione; sebbene esistano linguaggi che facilitano la scrittura di programmi secondo il paradigma funzionale (o addirittura impongono l'utilizzo di tale paradigma di programmazione), è possibile scrivere programmi secondo lo stile funzionale indipendentemente dal linguaggio di programmazione che si sta utilizzando (e quindi anche usando un linguaggio tradizionalmente considerato "imperativo" come il C).

Per capire meglio cosa si intenda per programmazione funzionale (e come utilizzare uno "stile di programmazione funzionale"), consideriamo il modo in cui siamo abituati a sviluppare un programma. Siamo tradizionalmente abituati a considerare un algoritmo come una sequenza di azioni che vanno ad "operare" su qualcosa: per esempio, una ricetta di cucina può essere vista come una sequenza di azioni sugli ingredienti, che li trasformano nel piatto che vogliamo cucinare... O, più informaticamente parlando, un programma può essere visto come una sequenza di istruzioni che agiscono su uno stato condiviso (memoria/variabili, dispositivi di I/O, ...).

Vedere un programma come una sequenza di istruzioni che agiscono sulla memoria (o sui dispositivi di I/O) è consistente con l'architettura dei moderni computer (una o più CPU che eseguono istruzioni Assembly, le quali operano su una memoria condivisa) che deriva dall'architettura di Von Neumann. Esistono però anche altri modi di pensare ad un programma (o ad un algoritmo).

Consideriamo per esempio l'algoritmo di Euclide per il calcolo del massimo comun divisore (mcm, gcd in inglese). Questo algoritmo, che conosciamo fin dalle scuole medie, dice circa: "*dati due numeri naturali a e b, se $b = 0$ allora a è il massimo comun divisore. Altrimenti, si assegna ad a il valore di b ed a b il resto della divisione fra a e b, poi si ricomincia dall'inizio.*"

Una semplice implementazione di questo algoritmo usando un approccio imperativo si ottiene "traducendolo" in un linguaggio di programmazione imperativo. Per esempio, usando la sintassi del linguaggio C si ottiene il codice mostrato in Figura 1.1: L'unica piccola complicazione rispetto alla descrizione dell'algoritmo è l'introduzione della variabile temporanea `tmp`, che serve a non perdere il valore di b quando si assegna `b = a % b`. Il discorso diventa più complicato quando ci chiediamo *perché l'algoritmo di Euclide funziona* (vale a dire: possiamo provare che l'algoritmo implementato qui sopra calcola correttamente il massimo comun divisore fra due numeri?). La spiegazione del funzionamento dell'algoritmo è circa la seguente:

- Il massimo comun divisore fra a e 0 è chiaramente a (poiché 0 è divisibile per qualsiasi numero, con resto 0)
- Il massimo comun divisore fra a e $b \neq 0$ è uguale al massimo comun divisore fra b ed $a\%b$ (dimostrabile per induzione)

Dal punto di vista matematico, questo significa che

$$gcd(a, b) = \begin{cases} \text{a} & \text{se } \text{b} = 0 \\ \text{gcd(b, a \% b)} & \text{altrimenti} \end{cases}$$

e questa definizione matematica ci porta ad una nuova implementazione dell'algoritmo di Euclide mostrata in Figura 1.2. I puristi obietteranno che questa implementazione non è strutturata (ha un unico punto di ingresso ma 2 diversi punti di uscita: ci sono 2 diversi statement `return`) e questo "problema" può essere risolto usando il cosiddetto "if aritmetico", come mostrato in Figura 1.3.

```
unsigned int gcd(unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int tmp;

    tmp = b;
    b = a % b;
    a = tmp;
  }

  return a;
}
```

Figura 1.1: Algoritmo di Euclide implementato in C.

```
unsigned int gcd(unsigned int a, unsigned int b)
{
  if (b == 0) {
    return a;
  }

  return gcd(b, a % b);
}
```

Figura 1.2: Algoritmo di Euclide implementato in modo ricorsivo.

```
unsigned int gcd(unsigned int a, unsigned int b)
{
  return (b == 0) ? a : gcd(b, a % b);
}
```

Figura 1.3: Implementazione puramente funzionale dell'algoritmo di Euclide.

Confrontando l'implementazione di Figura 1.1 (che verrà chiamata "implementazione imperativa") con quella di Figura 1.3 (che verrà chiamata "implementazione funzionale"), si possono notare due differenze importanti:

- Mentre l'implementazione imperativa dell'algoritmo lavora modificando il valore di alcune variabili (locali) "`a`", "`b`" e "`tmp`", l'implementazione funzionale non modifica il valore di alcuna variabile (in altre parole, non utilizza l'operatore "`=`" di assegnamento)
- Mentre l'implementazione imperativa utilizza un ciclo "`while(b != 0)`", l'implementazione funzionale utilizza il meccanismo della ricorsione (con la condizione "`b == 0`" come condizione di fine ricorsione / base induttiva)

Si noti come la seconda caratteristica dell'implementazione funzionale sia una conseguenza più o meno diretta della prima: un ciclo "`while`" è basato su un predicato che viene valuatato (ripetendo l'esecuzione del corpo del ciclo) fino a che non diventa falso. Il valore di questo predicato (nell'esempio, "`b != 0`") è calcolato in base al valore di una o più variabili ("`b`" nell'esempio precedente); quindi, se il valore di tali variabili non può essere modificato il valore di verità del predicato sarà sempre vero o sempre falso, rendendo praticamente inutilizzabile il costrutto di ciclo (che può dare origine solo a cicli infiniti o mai eseguiti). Per questo motivo, in assenza di variabili modificabili i costrutti di iterazione (`while` e simili) non possono essere utilizzati.

In generale, un'implementazione funzionale di un algoritmo sarà basata su *funzioni pure*, vale a dire funzioni intese nel senso matematico del termine (relazioni $f \subset \mathcal{D} \times \mathcal{C}$ fra un insieme dominio $\mathcal{D}$ ed un insieme del codominio $\mathcal{C}$ che ad ogni elemento del dominio assegnano alpiù un elemento del codominio), senza alcun tipo di *effetto collaterale*. Formalmente, $f(x) = y$ significa $(x, y) \in f$ ed indica che:

```
int f(int v)
{
  static int acc;

  acc = acc + 1;

  return v + acc;
}
```

Figura 1.4: Esempio di funzione non pura, con effetti collaterali.

- la funzione f associa sempre lo stesso valore $y \in \mathcal{C}$ al valore $x \in \mathcal{D}$
- calcolare y a partire da x è l'unico effetto della funzione

Poiché la modifica del valore di una variabile è un effetto collaterale dell'operatore di assegnamento, questo significa che **il paradigma di programmazione funzionale non prevede il concetto di variabili modificabili**. Come conseguenza, in un programma scritto secondo lo stile funzionale non esistono cicli, che sono sostituiti da chiamate ricorsive. Inoltre, non si usano comandi con effetti collaterali, ma solo espressioni che ritornano un valore e l'esecuzione di un programma non avviene modificando uno stato ma valutando espressioni. Questo è il motivo per cui usando il paradigma funzionale non si utilizza il costrutto condizionale `if` (che seleziona l'esecuzione alternativa di due diversi comandi) ma il cosiddetto *if aritmetico* (il costrutto "`... ? ... : ...`" in C), che genera un risultato valutando una fra due diverse espressioni dipendentemente dal valore di un predicato. La prima conseguenza di questo fatto è che in un programma funzionale i rami "`else`" dei costrutti "`if`" devono essere sempre presenti.

Successive valutazioni della stessa espressione (o invocazioni della stessa funzione pura con gli stessi parametri) devono risultare nello stesso risultato. Sebbene questo requisito possa sembrare banale e scontato (è insito nella definizione matematica di funzione), non è verificato ogni qual volta ci si trovi in presenza di effetti collaterali. Per esempio, si consideri la funzione C di Figura 1.4: successive invocazioni `f(2); f(2); f(2);` ritorneranno valori diversi (2, 3 e 4). Per capire come mai questo possa essere un problema, si consideri l'espressione `(f(2) + 1) * (f(2) + 5)`: se l'invocazione a sinistra è valutata per prima, il risultato è $(2+1)*(3+5) = 24$, altrimenti è $(3+1)*(2+5) = 28$.

1.2 Ricorsione ed Iterazione

Mentre i costrutti base della programmazione imperativa sono (secondo l'approccio strutturato) la sequenza di comandi, il costrutto di selezione (esecuzione condizionale, `if`) ed il ciclo (per esempio, `while`), i costrutti base della programmazione funzionale sono l'invocazione di funzione, l'operatore di if aritmetico e la ricorsione.

In particolare, l'equivalente funzionale dell'iterazione (ciclo) è la ricorsione: ogni algoritmo che codificato secondo il paradigma imperativo richiede un ciclo (finito o infinito), codificato secondo il paradigma funzionale risulta in una ricorsione (ancora, finita o infinita).

La tecnica della ricorsione (strettamente legata al concetto matematico di *induzione*) è usata in informatica per definire un qualche genere di "entità"[1] basata su se stessa. Focalizzandosi sulle funzioni ricorsive, si può definire una funzione $f()$ esprimendo il valore di $f(n)$ come funzione di altri valori calcolati da $f()$ (tipicamente, $f(n-1)$).

In generale, le definizioni ricorsive sono date "per casi", vale a dire sono composte da varie clausole. Una di queste è la cosiddetta *base* (detta anche *base induttiva*); esistono poi una o più clausole o *passi induttivi* che permettono di generare / calcolare nuovi elementi a partire da elementi esistenti. La base è una clausola della definizione ricorsiva che non fa riferimento all'"entità" che si sta definendo (per esempio: il massimo comun divisore fra a e 0 è a, etc...) ed ha il compito di porre fine alla ricorsione: senza una base induttiva, si da' origine ad una ricorsione infinita.

In sostanza, una funzione $f : \mathcal{N} \to \mathcal{X}$ è definibile definendo una funzione $g : \mathcal{N} \times \mathcal{X} \to \mathcal{X}$, un valore $f(0) = a$ ed imponendo che $f(n+1) = g(n, f(n))$. Più nei dettagli, una funzione è definibile per ricorsione quando ha come dominio l'insieme dei naturali (o un insieme comunque numerabile); il codominio può essere invece un generico insieme $\mathcal{X}$. Come base induttiva, si definisce il valore della funzione per il più

[1] Il termine "entità" è qui usato informalmente per indicare genericamente funzioni, insiemi, valori, tipi di dato, ...

```
unsigned int fattoriale(unsigned int n)
{
  if (n == 0) {
    return 1;
  }

  return n * fattoriale(n - 1);
}
```

Figura 1.5: Implementazione ricorsiva della funzione fattoriale.

```
unsigned int fattoriale(unsigned int n)
{
  return (n == 0) ? 1 : n * fattoriale(n - 1);
}
```

Figura 1.6: Implementazione funzionale della funzione fattoriale.

piccolo valore facente parte del dominio (per esempio, $f(0) = a$, con $a \in \mathcal{X}$) e come passo induttivo si definisce il valore di $f(n + 1)$ in base al valore di $f(n)$; come detto sopra, questo si può fare definendo $f(n+1) = g(n, f(n)))$. Notare che il dominio di $g()$ è l'insieme delle coppie di elementi presi dal dominio e dal codominio di $f()$, mentre il codominio di $g()$ è uguale al codominio di $f()$.

L'esempio tipico portato sempre quando si parla di ricorsione è la funzione fattoriale, che può essere codificata in modo ricorsivo come mostrato in Figura 1.5. Una versione più propriamente funzionale (perché utilizza solo espressioni, e non comandi) del fattoriale è mostrata invece in Figura 1.6.

Questo esempio ci può essere utile per vedere come l'esecuzione di un programma scritto secondo il paradigma funzionale possa essere vista come una sequenza di "semplificazioni" o "sostituzioni" (tecnicamente, *riduzioni*) analoghe a quelle fatte per valutare un'espressione aritmetica. Si consideri per esempio il calcolo di `fattoriale(4)`. Dopo che la funzione `fattoriale()` è stata definita (come sopra, per esempio), nell'ambiente esiste un legame fra il nome "fattoriale" ed un'entità denotabile (il corpo della funzione fattoriale). Di fronte all'espressione "`fattoriale(4)`" è quindi possibile cercare nell'ambiente il legame al corpo della funzione e sostituire "`fattoriale`" con la sua definizione, usando "`4`" (parametro attuale) al posto del parametro formale "`n`":

fattoriale (4) → (4 == 0) ? 1 : 4 ∗ fattoriale (4 − 1) → 4 ∗ fattoriale (3)

dove il primo passaggio corrisponde sostanzialmente alla sostituzione del nome "`fattoriale`" col corpo della funzione (secondo il legame trovato nell'ambiente) e la sostituzione del parametro formale "`n`" con il valore "`4`" in tale corpo, mentre il secondo passaggio è avvenuto perché $4 \neq 0$ (quindi, si valuta la seconda sottoespressione "4 ∗ fattoriale (4 − 1)") e $4-3 = 1$. A questo punto, si cerca ancora il nome "`fattoriale`" nell'ambiente e si applica il corpo della funzione al parametro attuale "`3`":

4 ∗ fattoriale (3) → 4 ∗ ((3 == 0) ? 1 : 3 ∗ fattoriale (3 − 1)) → 4 ∗ (3 ∗ fattoriale (2))

procedendo analogamente si ottiene

4 ∗ (3 ∗ fattoriale (2)) → 4 ∗ (3 ∗ ((2 == 0) ? 1 : 2 ∗ fattoriale (2 − 1))) →
→ 4 ∗ (3 ∗ (2 ∗ fattoriale (1))) → 4 ∗ (3 ∗ (2 ∗ ((1 == 0) ? 1 : 1 ∗ fattoriale (1 − 1)))) →
→ 4 ∗ (3 ∗ (2 ∗ (1 ∗ fattoriale (0)))) →
→ 4 ∗ (3 ∗ (2 ∗ (1 ∗ ((0 == 0) ? 1 : 0 ∗ fattoriale (0 − 1))))) →
→ 4 ∗ (3 ∗ (2 ∗ (1 ∗ 1))) = 24.

Si noti come dal punto di vista logico il calcolo del fattoriale abbia richiesto solo operazioni di:

1. ricerca nell'ambiente (per poter applicare una funzione ai suoi argomenti / parametri attuali bisogna trovare nell'ambiente il binding fra il nome della funzione ed il suo corpo)

2. sostituzione testuale (l'applicazione di una funzione ai suoi argomenti si ottiene per semplice sostituzione dei parametri attuali al posto dei parametri formali nel corpo della funzione trovato al punto 1)

3. calcolo aritmetico (questo include sia l'esecuzione di operazioni aritmetiche "semplici" come prodotti e sottrazioni che la valutazione di if aritmetici)

Questo mostra come secondo il paradigma funzionale l'esecuzione di un programma avvenga per riduzione, vale a dire per sostituzione testuale di espressioni e sotto-espressioni: una funzione applicata ad un argomento è sostituita dal corpo della funzione ed il parametro formale è sostituito dal parametro attuale. Concettualmente, questo processo di riduzione non richiede l'esecuzione di istruzioni Assembly o di programmi, ma solo manipolazioni di stringhe come quelle operabili (per esempio) con un editor di testo (in più, sarà necessario effettuare i calcoli richiesti dalle varie operazioni aritmetiche - quindi, volendo continuare con l'analogia di cui sopra si può dire come siano in realtà necessari un editor di testo ed una calcolatrice).

Chiaramente, questo concetto di computazione come riduzione è applicabile solo a funzioni pure, vale a dire senza effetti collaterali: se `fattoriale()` avesse effetti collaterali (come per esempio la modifica di variabili globali, o simili) non sarebbe possibile sostituire semplicemente " fattoriale (4)" con "4 $*$ fattoriale (3)". Questo spiega perché l'assenza di effetti collaterali è (come già anticipato) un requisito fondamentale per il paradigma di programmazione funzionale; l'eliminazione di variabili modificabili è un modo semplice per eliminare tutta una grande classe di effetti collaterali (rimangono gli effetti collaterali dovuti a I/O, ma questi spesso non possono essere eliminati senza rendere inutile il programma).

Il paradigma di programmazione funzionale, che è stato presentato in queste pagine come alternativa al "tradizionale" paradigma imperativo, ha lo stesso potere espressivo del paradigma imperativo (vale a dire: qualsiasi algoritmo codificabile usando un approaccio imperativo può essere implementato anche usando l'approccio funzionale). I lettori più abituati a sviluppare programmi secondo l'approccio imperativo potranno obiettare che "rinunciare" alle variabili modificabili ed all'iterazione sembra complicare lo sviluppo dei programmi e che di conseguenza l'approccio funzionale può apparire un po' innaturale. In realtà questo è più un problema di abitudine ed una volta che si capisce la logica della programmazione funzionale lo sviluppo di programmi secondo questo approccio risulterà più semplice. Inoltre, esistono problemi che sono più facilmente risolubili usando la ricorsione e che non sono propriamente semplici da risolvere usando un approccio puramente imperativo. Per esempio, consideriamo il problema delle torri di Hanoi.

Il problema consiste nello spostare una torre composta da N dischi (di dimensioni decrescenti) da un palo ad un altro, usando un terzo palo "di appoggio" per gli spostamenti. Le regole del gioco prevedono che si possa spostare un solo disco per volta e che non si possa appoggiare un disco più grande sopra ad uno più piccolo. Mentre sviluppare una soluzione non ricorsiva al problema non è semplicissimo (e richiede di utilizzare strutture dati complesse), una soluzione ricorsiva è banale. In pratica, il problema di spostare N dischi dal palo a al palo b (usando il palo c come appoggio) è scomponibile nel problema di:

- Spostare $N - 1$ dischi dal palo a al palo c
- Spostare il rimanente disco (il più grande) dal palo a al palo b
- Spostare gli $N - 1$ dischi dal palo c al palo b

Ora, mentre il secondo passo dell'algoritmo (spostare un disco da un palo all'altro) è semplice, il primo ed il terzo passo richiedono di spostare $N - 1$ dischi e non sono direttamente implementabili. Ma se sappiamo come spostare N dischi, possiamo utilizzare (ricorsivamente!) lo stesso algoritmo per spostare $N - 1$ dischi (e questo richiedera di spostare $N - 2$ dischi, poi un disco e poi ancora $N - 2$ dischi). E questa ricorsione può essere invocata più volte (per la precisone, $N - 1$ volte) fino a che il problema non è ridotto allo spostamento di un solo disco.

La Figura 1.7 mostra una semplice implementazione di questo algoritmo usando il linguaggio C. Si noti come in questo caso la condizione di fine ricorsione (base induttiva) corrisponda allo spostamento di un singolo disco (come nell'algoritmo descritto poc'anzi). Un'implementazione alternativa avrebbe potuto usare la condizione "`n == 0`" (nessun disco da spostare) come base induttiva, risultando nell'implementazione della funzione `move()` mostrata in Figura 1.8.

Un'importante considerazione da fare sul codice proposto è che sebbene utilizzi la ricorsione non è ancora implementato secondo un approccio puramente funzionale: la funzione `move()` utilizza infatti una sequenza di comandi (`move()` e `move_disk()` non hanno alcun valore di ritorno) basando il proprio funzionamento sugli effetti collaterali di tali comandi (la stampa a schermo tramite `printf()`). Per implementare `move()` come una funzione pura, bisognerebbe eliminarne gli effetti collaterali (vale a dire, eliminare la chiamata a `printf()` da `move_disk()`, aggiungendo un valore di ritorno a `move()` e `move_disk()`). La soluzione più ovvia è quella di fare si che `move()` e `move_disk()` ritornino una stringa

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void move_disk(const char *from, const char *to)
{
  printf("Muovi disco da %s a %s\n", from, to);
}

void move(unsigned int n, const char *from, const char *to, const char *via)
{
  if (n == 1) {
    move_disk(from, to);

    return;
  }

  move(n - 1, from, via, to);
  move_disk(from, to);
  move(n - 1, via, to, from);
}

int main(int argc, char *argv[])
{
  unsigned int height = 0;

  if (argc > 1) {
    height = atoi(argv[1]);
  }
  if (height <= 0) {
    height = 8;
  }

  move(height, "Left", "Right", "Center");

  return 0;
}
```

Figura 1.7: Soluzione del problema delle torri di Hanoi.

```c
void move(unsigned int n, const char *from, const char *to, const char *via)
{
  if (n == 0) {
    return;
  }

  move(n - 1, from, via, to);
  move_disk(from, to);
  move(n - 1, via, to, from);
}
```

Figura 1.8: Soluzione alternativa del problema delle torri di Hanoi.

contenente il loro output (in altre parole, la sequenza di mosse da fare per risolvere il problema non deve essere stampata a schermo tramite `printf()`, ma salvata in una stringa).

```
const char *concat(const char *a, const char *b)
{
  char *res;

  res = malloc(strlen(a) + strlen(b) + 1);
  memcpy(res, a, strlen(a));
  memcpy(res + strlen(a), b, strlen(b));
  res[strlen(a) + strlen(b)] = 0;

  return res;
}

const char *move_disk(const char *from, const char *to)
{
  return concat(concat(concat(concat("Move disk from ", from),
                 " to "), to), "\n");
}

const char *move(int n, const char *from, const char *to, const char *via)
{
  return (n == 1) ?
      move_disk(from, to)
    :
      concat(concat(move(n - 1, from, via, to),
                 move_disk(from, to)), move(n - 1, via, to, from));
}
```

Figura 1.9: Soluzione funzionale del problema delle torri di Hanoi.

Una possibile soluzione (purtroppo non leggibilissima a causa della sintassi del linguaggio C) è mostrata in Figura 1.9. In questa soluzione, la funzione di utilità `concat()` riceve in ingresso due stringhe (in C, array di caratteri) e ritorna una stringa che contiene (come il nome suggerisce) la concatenazione delle due. Ci sono un po' di cose importanti da notare:

- L'utilizzo della notazione funzionale per `concat()` (invece di un operatore infisso, come in altri linguaggi) riduce la leggibilità del codice. Il lettore non deve però essere confuso dalle lunghe catene "`concat(concat(concat(...)))`", che non fanno altro che concatenare lunghe sequenze di stringhe

- Le funzioni `move()` e `move_disk()` sono ora *funzioni pure*, in quanto non fanno affidamento su effetti collaterali

- Gli effetti collaterali sono ora tutti concentrati nella funzione `main()`, che effettua I/O... E' chiaro che se il programma deve comunicare coll'ambiente esterno da qualche parte dovrà effettuare I/O, che è un effetto collaterale; un programma non potrà quindi mai essere "puramente funzionale", ma tenderà a concentrare tutti gli effetti collaterali in punti specifici (per i linguaggi funzionali, il ciclo Read-Evaluate-Print o il supporto runtime)

- A conferma del fatto che sono funzioni pure, `move()` e `move_disk()` non modificano il contenuto di variabili (non usano assegnamenti, o altri comandi, ma sono composte solo da espressioni)

- I lettori più attenti si saranno accorti del fatto che il programma presenta dei memory leak: `concat()` alloca dinamicamente memoria per la stringa che ritorna, ma tale memoria non viene mai liberata. Questo fatto è una consegenza del punto precedente (il contenuto delle stringhe non viene mai modificato, ma la concatenazione di due stringhe avviene allocando dinamicamente la memoria per la stringa risultato) e mostra come il paradigma di programmazione funzionale richieda un *garbage collector* (che viene infatti sempre incluso nelle macchine astratte che implementano linguaggi di programmazione funzionali)

```
#include <cstdlib>
#include <iostream>
#include <string>

std::string move_disk(std::string from, std::string to)
{
  return "Move disk from " + from + " to " + to + "\n";
}

std::string move(int n, std::string from, std::string to, std::string via)
{
  return (n == 1) ?
      move_disk(from, to)
    :
      move(n - 1, from, via, to) +
                move_disk(from, to) + move(n - 1, via, to, from);
}

int main(int argc, char *argv[])
{
  int height = 0;
  std::string res;

  if (argc > 1) {
    height = atoi(argv[1]);
  }
  if (height <= 0) {
    height = 8;
  }

  res = move(height, "Left", "Right", "Center");

  std::cout << res;

  return 0;
}
```

Figura 1.10: Soluzione funzionale in C++ del problema delle torri di Hanoi.

Per mostrare che molte delle complicazioni che appaiono nel programma precedente non sono dovute allo stile di programmazione funzionale, ma alla sintassi del linguaggio C (ed alla sostanziale mancanza di funzioni per la gestione delle stringhe) la Figura 1.10 mostra una reimplementazione in C++ (che fornisce un supporto per le stringhe molto più avanzato rispetto al C), che appare molto più pulita.

1.3 Ricorsione e Stack

Come precedentemente visto, per valutare un'espressione tramite sostituzione/riduzione non è concettualmente necessario introdurre i concetti di invocazione di subroutine, stack, record di attivazione e simili. D'altra parte, se la macchina astratta che esegue il nostro programma (o, valuta la nostra espressione) è implementata su un'architettura hardware basata sul modello di Von Neumann (come tutti i moderni PC) dovrà per forza utilizzare il meccanismo di chiamata a subroutine (istruzione Assembly `call` su architetture Intel, etc...) per invocare l'esecuzione di una funzione[2].

[2]Anche implementazioni alternative, che interpretano la funzione invece di compilarla in Assembly per rimanere più fedeli al modello di valutazione per sostituzione visto in precedenza, dovranno utilizzare strutture dati a pila (o simile) che crescono ad ogni applicazione della funzione.

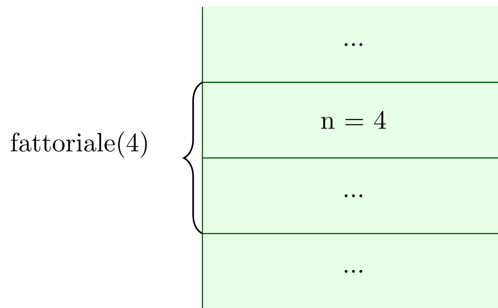

Figura 1.11: Stack frame per l'invocazione `fattoriale(4)`.

Tornando all'esempio del fattoriale visto in precedenza, ogni volta che la funzione `fattoriale()` invoca se stessa (ma questo vale anche per la generica invocazione di altre funzioni) dovrà quindi essere aggiunto un nuovo record di attivazione (o stack frame) sullo stack, facendone crescere la dimensione. In particolare, ad ogni invocazione ricorsiva della funzione verrà aggiunto sullo stack un record di attivazione contenente il parametro attuale con cui `fattoriale()` è stata invocata, un link dinamico (ma in alcuni linguaggi anche un link statico) al precedente stack frame ed un po' di spazio per memorizzare il valore di ritorno. Quando si invoca "`fattoriale(4)`" si crea quindi la situazione visibile in Figura 1.11.

Poiché anche "`fattoriale(3)`" si invocherà ricorsivamente, lo stack evolve come mostrato in Figura 1.12, crescendo ad ogni invocazione ricorsiva. Questo comporta che il calcolo del fattoriale di un numero `n` abbastanza grande richiederà una grande quantità di memoria. Si noti che il record di attivazione corrispondente a "`fattoriale(n)`" non può essere rimosso dallo stack fino a che "`fattoriale(n - 1)`" non è terminata, perché contiene il valore "`n`" per cui il risultato di "`fattoriale(n - 1)`" deve essere moltiplicato. In sostanza, quando si arriva a valutare "`fattoriale(0)`, gli stack frame precedenti contengono i numeri da moltiplicare; man mano che le varie istanze di `fattoriale()` terminano, gli stack frame vengono rimossi dallo stack uno dopo l'altro (dopo aver eseguito la moltiplicazione per il valore di "`n`" contenuto nello stack frame). I vari stack frame sono quindi necessari fino a che la relativa istanza di "`fattoriale`" non termina, e non possono essere rimossi prima dallo stack.

Questo problema sembrerebbe compromettere la reale utilizzabilità delle tecniche di programmazione funzionale, in quanto lunghe catene di chiamate ricorsive porterebbero ad un consumo di memoria eccessivo (mentre lunghe iterazioni hanno generalmente un consumo di memoria basso e costante). Per capire meglio questa cosa, si consideri il problema di implementare una funzione che testa se un numero naturale è pari o dispari senza usare operazioni di divisione o modulo. Una possibile soluzione è mostrata in Figura 1.13. La soluzione proposta usa una mutua ricorsione fra le funzioni `pari()` e `dispari()`, basandosi sull'idea che 0 è pari, 1 è dispari (basi induttive) ed ogni numero $n > 1$ è pari se $n - 1$ è dispari o è dispari se $n - 1$ è pari (passo induttivo). A parte la bizzarria di questa soluzione, si può immediatamente immaginare come l'invocazione di `pari()` o `dispari()` su numeri grandi possa finire per causare una grossa crescita dello stack (fino allo stack overflow). Infatti, se si compila il programma con `gcc paridispari.c` e se ne testa il funzionamento per numeri piccoli tutto sembra funzionare... Ma provando con numeri abbastanza grandi (si provi per esempio 24635743, come suggerito nel commento) si ottiene un segmentation fault dovuto a stack overflow.

La cosa sorprendente è però che provando a compilare il programma con `gcc -O2 paridispari.c` il programma funziona correttamente con qualsiasi numero si immetta in ingresso! Questo accade perché il problema della crescita dello stack è facilmente aggirabile usando la cosiddetta "tail call optimization" (ottimizzazione delle chiamate in coda, abilitata dallo switch "`-O2`" di `gcc`), che permette, sotto opportune ipotesi, di sostituire invocazioni di funzioni con semplici salti (trasformando quindi una ricorsione in un'iterazione).

Per capire meglio come funziona questa ottimizzazione, consideriamo la versione "tail recursive" del fattoriale, mostrata in Figura 1.14. Intuitivamente, si può vedere come la funzione `fattoriale_tr()` utilizzi un secondo argomento per "accumulare" il risultato: la moltiplicazione per "`n`" avviene *prima* della chiamata ricorsiva (per calcolare il valore del secondo parametro attuale) e non dopo. Questo comporta che il valore "`n`" non deve essere salvato per essere utilizzato quando la chiamata ricorsiva termina... L'espressione "`fattoriale(4)`" viene valutata come segue:

$\text{fattoriale}\ (4) \rightarrow\ \text{fattoriale_tr}\ (4,\ 1) \rightarrow (4\ ==\ 0)\ ?\ 1:\ \text{fattoriale_tr}\ (4\ -\ 1,\ 4 * 1) \rightarrow$

$\rightarrow\ \text{fattoriale_tr}\ (3,\ 4) \rightarrow (3\ ==\ 0)\ ?\ 4:\ \text{fattoriale_tr}\ (3\ -\ 1,\ 3 * 4) \rightarrow$

$\rightarrow\ \text{fattoriale_tr}\ (2,\ 12) \rightarrow (2\ ==\ 0)\ ?\ 12:\ \text{fattoriale_tr}\ (2\ -\ 1,\ 2 * 12) \rightarrow$

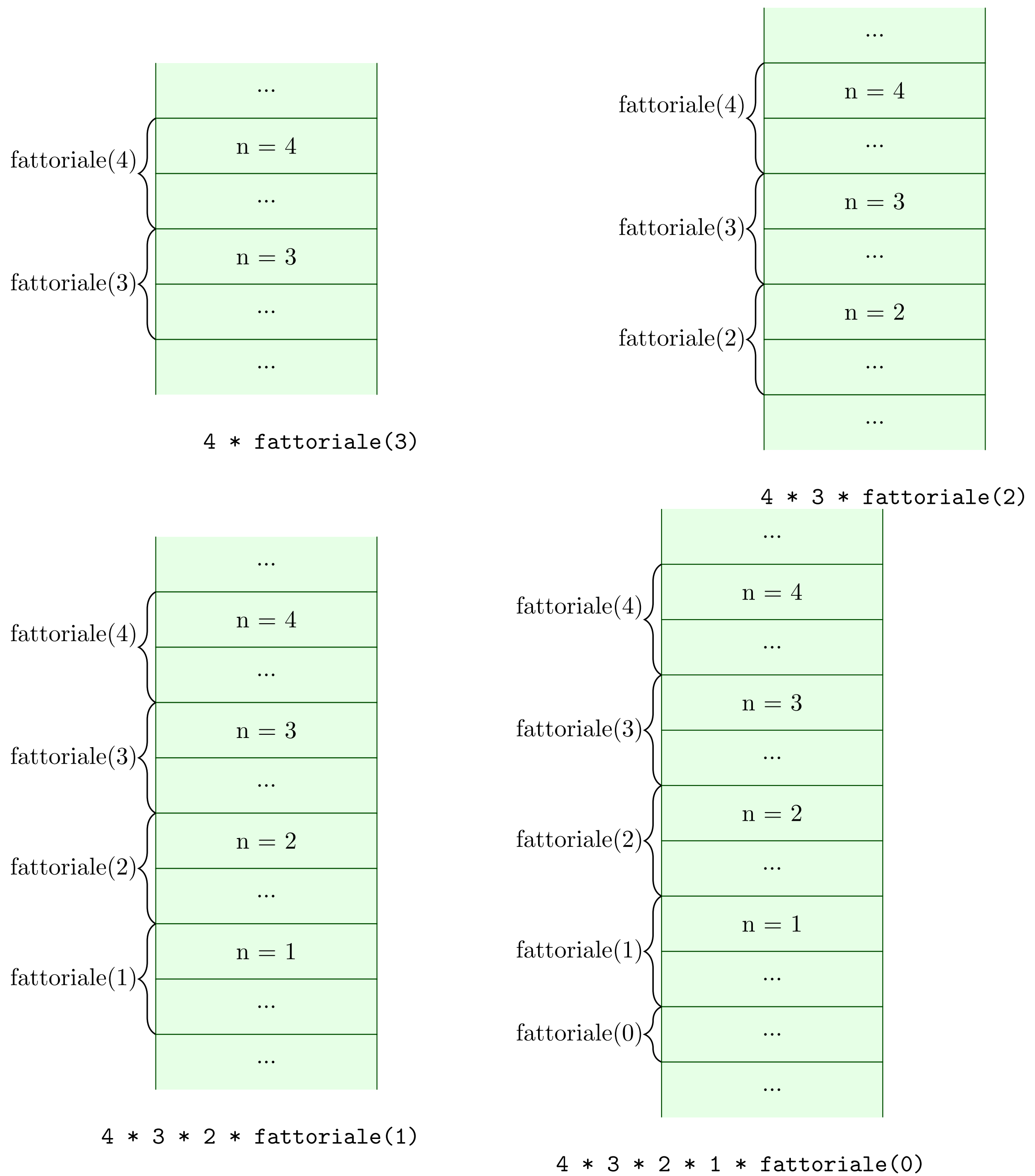

Figura 1.12: Evoluzione dello stack per l'invocazione `fattoriale(4)`.

```
/* Try 24635743... */

unsigned int pari(unsigned int n);
unsigned int dispari(unsigned int n)
{
  if (n == 0) return 0;
  return pari(n - 1);
}

unsigned int pari(unsigned int n)
{
  if (n == 0) return 1;
  return dispari(n - 1);
}
```

Figura 1.13: Funzioni (mutuamente) ricorsive per testare se un numero è pari o dispari.

```
unsigned int fattoriale_tr(unsigned int n, unsigned int res)
{
  return (n == 0) ? res : fattoriale_tr(n - 1, n * res);
}

unsigned int fattoriale(unsigned int n)
{
  return fattoriale_tr(n, 1);
}
```

Figura 1.14: Versione tail recursive del fattoriale.

→ fattoriale_tr (1, 24) → (1 == 0) ? 24 : fattoriale_tr (1 − 1, 1 ∗ 24) →
→ fattoriale_tr (0, 24) → (0 == 0) ? 24 : fattoriale_tr (0 − 1, 0 ∗ 24) → 24

L'osservazione fondamentale qui è che quando "`fattoriale_tr(0, 24)`" ritorna, "`fattoriale_tr(1, 24)`" può immediatamente ritornare il valore da essa ritornato... E così pure "`fattoriale_tr(2, 12)`", "`fattoriale_tr(3, 4)`" e "`fattoriale_tr(4, 1)`". Quindi, non è più necessario salvare sullo stack i record di attivazione per contenere il valore di "`n`", il valore di ritorno e l'indirizzo di ritorno: "`fattoriale_tr(0, 24)`" può direttamente ritornare il risultato 24 al chiamante originario. La Figura 1.15 mostra i dettagli dell'evoluzione dello stack derivante dall'invocazione di "`fattoriale(4)`", chiarendo ancora di più che durante l'esecuzione di un'istanza di `fattoriale_tr()` i record di attivazione corrispondenti alle precedenti istanze contengono dati che rimangono inutilizzati (risultando quindi essere non necessari / inutili!!!). In sostanza, quando si arriva a valutare "`fattoriale_tr(0, 24)`", tutti i dati necessari al calcolo sono contenuti nei parametri attuali ed i record di attivazione di "`fattoriale_tr(1, 24`"..."`fattoriale_tr(4, 1)`" contenuti sullo stack non vengono acceduti. Tali record di attivazione vengono rimossi dallo stack uno dopo l'altro (quando le varie istanze di `fattoriale_tr()` terminano) senza dover eseguire ulteriori operazioni su di essi: quando "`fattoriale_tr(n - 1, ...)`" termina, "`fattoriale_tr(n, ...)`" ritorna direttamente il suo valore di ritorno, senza eseguire ulteriori operazioni. Questo significa che quando la chiamata ricorsiva ritorna, ogni istanza di `fattoriale_tr()` può terminare immediatamente, passando direttamente al proprio chiamante il valore di ritorno ricevuto dall'invocazione ricorsiva. I vari stack frame possono quindi essere rimossi dallo stack al momento della ricorsione (prima che la funzione associata termini), trasformando di fatto una chiamata ricorsiva in un semplice salto.

Questa ottimizzazione è possibile ogni volta che una funzione ritorna come valore di ritorno il risultato ottenuto dall'invocazione di un'altra funzione (in sostanza, "`return altrafunzione(...)`"): in pratica, statement del tipo "**return** f(n);" non vengono compilati come invocazioni alla subroutine `f()`, ma come salti al corpo di tale funzione. Questo permette di implementare la ricorsione senza causare eccessivi consumi di stack, rendendo praticabile l'utilizzo del paradigma di programmazione funzionale (a patto di scrivere codice che utilizzi chiamate in coda).

Per capire come funziona la tail call optimization in pratica, si consideri il codice Assembly x86_64

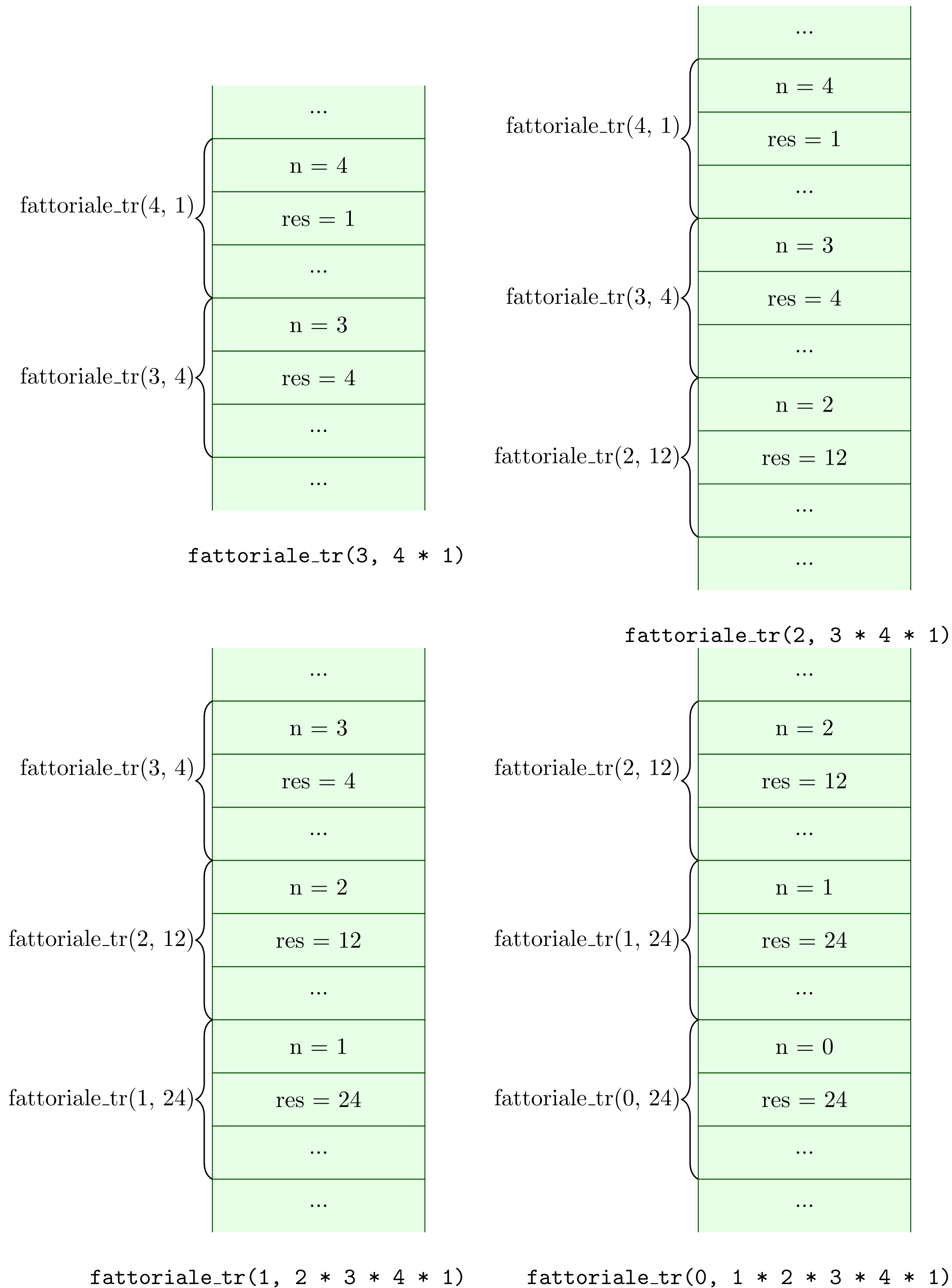

Figura 1.15: Evoluzione dello stack per l'invocazione `fattoriale(4)`.

```
fattoriale_tr:
        movl    %esi, %eax
        testl   %edi, %edi
        je      .L2
        subq    $8, %rsp
        imull   %edi, %esi
        subl    $1, %edi
        call    fattoriale_tr
        addq    $8, %rsp
.L2:
        ret
```

Figura 1.16: Versione tail recursive del fattoriale compilata senza ottimizzazioni.

```
fattoriale_tr:
        movl    %esi, %eax
        testl   %edi, %edi
        je      .L2
        imull   %edi, %esi
        subl    $1, %edi
        jmp     fattoriale_tr
.L2:
        ret
```

Figura 1.17: Versione tail recursive del fattoriale ottimizzata.

generato da `gcc -O1` quando compila la funzione `fattoriale_tr()` di Figura 1.14. Questo codice è mostrato in Figura 1.16.

La prima istruzione (`movl`) copia il secondo argomento (contenuto nel registro `%esi`) nel registro `%eax` (che verrà utilizzato per il valore di ritorno); la seconda istruzione (`testl`) controlla se il primo argomento (contenuto nel registro `%edi`) è 0: in questo caso, si salta subito all'uscita della funzione (label `.L2`) ritornando il valore contenuto in `%eax` (nel quale è appena stato copiato il secondo argomento). Se invece il primo argomento è > 0, la funzione moltiplica il secondo argomento per il primo e poi si invoca ricorsivamente (le istruzioni `subq` ed `addq` applicate ad `%rsp` sono necessarie a causa delle convenzioni di chiamata dell'ABI intel a 64 bit). Si noti che al ritorno dalla chiamata ricorsiva (istruzione successiva alla `call`) non vengono eseguite altre istruzioni e la funzione termina immediatamente. E' allora possibile evitare di pushare sullo stack successivi indirizzi di ritorno inutili (quando una istanza di `fattoriale_tr()` ritorna, tutte le precedenti istanze ritorneranno immediatamente, "a catena", senza frapporre istruzioni Assembly fra le varie "`ret`"). Questo può essere fatto semplicemente eliminando le istruzioni che manipolano lo stack (registro `%rsp`) e sostituendo la `call fattoriale_tr` con una `jmp fattoriale_tr`, come mostrato in Figura 1.17.

1.4 Funzioni e Spezie

In matematica, siamo abituati a considerare funzioni $f : \mathcal{D} \rightarrow \mathcal{C}$ che mappano uno o più elementi del dominio $\mathcal{D}$ in alpiù un elemento del codominio $\mathcal{C}$. In pratica, f è una relazione (sottoinsieme $f \subset \mathcal{D} \times \mathcal{C}$ delle coppie aventi il primo elemento in $\mathcal{D}$ ed il secondo elemento in $\mathcal{C}$) per cui $(x_1, y_1) \in f \wedge (x_1, y_2) \in f \Rightarrow y_1 = y_2$. Se f ha più di un argomento, il dominio $\mathcal{D}$ è rappresentato come prodotto cartesiano di altri insiemi: per esempio, una funzione da coppie di numeri reali in numeri reali sarà $f : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$[3].

Dal punto di vista informatico, siamo invece abituati a considerare funzioni a più argomenti in cui un argomento è rappresentato da un diverso parametro formale. Per esempio, "**int** f(**int** a, **float** x, **unsigned int** z)" è una funzione che riceve un argomento di tipo intero, un argomento che rappresenta l'approssimazione di un numero reale ed un terzo argomento intero positivo (vale a dire, un numero naturale); si potrebbe quindi dire che è equivalente ad una funzione $f : \mathcal{Z} \times \mathcal{R} \times \mathcal{N} \rightarrow \mathcal{Z}$. In alcuni linguaggi di programmazio-

[3]tipicamente, nei corsi avanzati di analisi si considerano funzioni $f : \mathcal{R}^n \rightarrow \mathcal{R}^m$).

```
int sommainteri(int a, int b)
{
  return a + b;
}
```

Figura 1.18: Funzione C che somma 2 interi.

```
int (* somma_c(int a))(int b)
{
  int s(int b) {
    return a + b;
  }

  return s;
}
```

Figura 1.19: Tentativo di currying della funzione `sommainteri()` (Figura 1.18) usand il linguaggio C.

ne è possibile utilizzare un tipo *tupla* per raggruppare tutti gli argomenti e rappresentare meglio valori appartenenti a $\mathcal{Z} \times \mathcal{R} \times \mathcal{N}$.

Questo non è però l'unico modo di rappresentare funzioni a più argomenti: in particolare, è stato mostrato da diversi matematici come qualsiasi funzione a più argomenti sia rappresentabile usando funzioni ad un solo argomento. Per esempio, usando la cosiddetta tecnica del *currying*[4] una funzione ad n argomenti è rappresentabile come una "catena" di funzioni aventi tutte un solo argomento. Il "trucco" per ottenere questo risultato è che ogni funzione di questa "catena" ha come valore di ritorno una funzione (e non un "valore semplice"). Per esempio, una funzione $f : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$ è rappresentabile come $f : \mathcal{R} \rightarrow (\mathcal{R} \rightarrow \mathcal{R})$.

Questo fatto ha alcune importanti conseguenze, sia teoriche che pratiche. La prima conseguenza teorica è che un formalismo matematico (come per esempio il λ-calcolo) che considera solo funzioni aventi un unico argomento può essere perfettamente generico, a patto che queste funzioni possano generare valori di tipo funzione come risultato ed accettare valori di tipo funzione come argomenti (tali funzioni sono spesso chiamate *funzioni di ordine superiore*). La seconda conseguenza, che ha ricadute anche pratiche, è quindi l'introduzione di *funzioni di ordine superiore*, vale a dire funzioni che possono manipolare valori di tipo funzione. Per questo motivo nei linguaggi di programmazione funzionale c'è un'uniformità fra codice e dati, nel senso che le funzioni possono essere valori memorizzabili ed esprimibili[5].

E' importante notare come dal punto di vista informatico la tecnica del currying sia utilizzabile in presenza di funzioni che generano funzioni (e non semplici puntatori a funzione) come valore di ritorno. Questo fatto ne preclude, per esempio, l'utilizzo in linguaggi come il C. Per capire questa cosa, si consideri la funzione `sommainteri()` mostrata in Figura 1.18 e si provi a generarne una "forma curryificata". Tale funzione dovrebbe ricevere in ingresso un intero `a` e generare come risultato una funzione che dato un intero `b` somma `a` a `b`. Utilizzando una piccola estensione al linguaggio C implementata da gcc, che permette di annidare le definizioni di funzioni, si potrebbe pensare di codificare la cosa come in Figura 1.19. A parte la scarsa leggibilità che ci è ancora una volta regalata dalla sintassi del linguaggio C (ed il fatto che si è utilizzata una definizione di `s()` annidata dentro alla definizione di `somma_c()`, cosa non permessa dal linguaggio C standard), questo codice implementa una funzione `somma_c` che riceve un parametro formale `a` di tipo `int` e ritorna un puntatore ad una funzione che riceve un parametro formale di tipo `int` e ritorna un valore di tipo `int`. La funzione è quindi utilizzabile come mostrato in Figura 1.20, dove `f` è una variabile di tipo puntatore a funzione da intero a intero.

Sebbene questo codice sia compilabile con gcc (e sembri addirittura funzionare correttamente in alcuni casi), contiene un grosso errore concettuale: `somma_c()` ritorna un *puntatore* alla funzione `s()` che utilizza una variabile non locale (ad `s()`) `a`. Tale variabile è il parametro attuale di `somma_c`, che sta memorizzato sullo stack durante il tempo di vita di `somma_c`... Ma viene rimosso quando `somma_c` termina! L'invocazione `f(2)` andrà quindi ad accedere a memoria non inizializzata, generando risultati non

[4]Si noti che il nome di questa tecnica deriva da Haskell Curry, non da una spezia!

[5]Si ricorda che un valore è memorizzabile quando può essere assegnato ad una variabile ed è esprimibile quando può essere generato come risultato di un'espressione.

```
int main()
{
  int (*f)(int b);

  f = somma_c(3);

  printf("3 + 2 = %d\n", f(2));

  return 0;
}
```

Figura 1.20: Utilizzo della funzione `somma_c()` di Figura 1.19.

```
int main()
{
  int (*f1)(int b);
  int (*f2)(int b);

  f1 = somma_c(3);
  f2 = somma_c(4);

  printf("3 + 2 = %d\n", f1(2));
  printf("4 + 2 = %d\n", f2(2));

  return 0
}
```

Figura 1.21: Problema con la funzione `somma_c()` di Figura 1.19.

definiti (undefined behaviour). Sebbene qualche semplice test possa funzionare correttamente, l'esempio di Figura 1.21 mostrerà tutti i limiti della soluzione precedentemente proposta.

Il problema può essere risolto solo facendo si che la nostra funzione `somma_c()` ritorni una *vera funzione* (comprensiva anche del suo ambiente non locale!) e non semplicemente un puntatore a funzione. Tecnicamente, questa cosa è implementabile usando una *chiusura*, vale a dire una coppia "(ambiente, puntatore a funzione)" dove l'ambiente dovrà contenere il binding fra il nome "`a`" ed una variabile non allocata sullo stack. La soluzione tipica è di allocare nello heap un record di attivazione che contiene le variabili non locali riferite nella chiusura; questo chiaramente crea dei rischi di memory leak (che con la "tradizionale" allocazione dei record di attivazione sullo stack non si incontrano) e rende necessaria l'implementazione di un garbage collector (quando la chiusura non è più utilizzata, il record di attivazione allocato sullo heap può essere deallocato). D'altra parte, abbiamo già visto in precedenza come il paradigma di programmazione funzionale renda necessaria la presenza di un garbage collector.

Utilizzando una sintassi del tipo "`<tipo1> -> <tipo2>`" per rappresentare una funzione che riceve un argomento di tipo "`<tipo1>`" e ritorna un risultato di tipo "`<tipo2>`", la soluzione corretta al problema di cui sopra (codificare la "forma curryificata" della funzione `sommainteri()` di Figura 1.18) è mostrata in Figura 1.22. In generale, una funzione "*tipo3* `f(`*tipo1* `a,` *tipo2* `b)`" soggetta a currying diventa "*tipo2*`->`*tipo3* `fc(`*tipo1*`a)`" tale che `f(a,b) = (fc(a))(b)` (dal punto di vista matematico, $f(a, b) \to f'(a) = f_a : f_a(b) = f(a, b)$). Se una funzione ha più di 2 argomenti, si rimuovono uno alla volta usando il currying.

Riassumendo, funzioni a più argomenti e funzioni di ordine superiore ad un solo argomento hanno lo stesso potere espressivo; molti linguaggi di programmazione funzionale, fornendo funzioni di ordine superiore si limitano ad un solo argomento / parametro formale. ML ed Haskell adottano questo approccio: anche se esiste una sintassi semplificata che permette di definire funzioni come se avessero più argomenti (per esempio, usando la keyword `fun` in Standard ML) questa viene poi convertita dalla macchina astratta nella definizione della funzione "curryificata". Per esempio, in Standard ML **fun** f p1 p2 p3 ... = exp; è equivalente a **val rec** f = **fn** p1 => **fn** p2 => **fn** p3 . . . => exp;.

Un esempio che può essere utile per capire meglio il concetto di currying è quello della funzione

```
int->int somma_c(int a)
{
  int s(int b) {
    return a + b;
  }

  return s;
}
```

Figura 1.22: Currying corretto della funzione `sommainteri()` (Figura 1.18) usando uno pseudo-lingugaggio simile al C.

```
double calcoladerivata(double (*f)(double x), double x)
{
  const double delta = 0.001;

  return (f(x) - f(x - delta)) / delta;
}
```

Figura 1.23: Calcolo della derivata di una funzione in un punto, in C.

```
double->double derivata(double f(double x))
{
  double f1(double x)
  {
    const double delta = 0.001;

    return (f(x) - f(x - delta)) / delta;
  }

  return f1;
}
```

Figura 1.24: Calcolo della derivata di una funzione.

derivata: si consideri l'implementazione di una funzione `calcoladerivata()` che calcola la derivata (meglio: il rapporto incrementale sinistro per un piccolo valore di δ) di una data funzione in un punto specificato. La funzione riceve quindi come argomenti la funzione $f : \mathcal{R} \rightarrow \mathcal{R}$ di cui calcolare la derivata ed il punto $x \in \mathcal{R}$ in cui calcolare la derivata, ritornando il valore $d \in \mathcal{R}$ della derivata. Una semplice implementazione di `calcoladerivata()` usando il linguaggio C può per esempio essere quella mostrata in Figura 1.23.

Si provi ora ad implementare una funzione `derivata()`, simile alla precedente, ma che ritorna la funzione derivata (ancora: il rapporto incrementale sinistro per un piccolo valore di δ) invece che calcolarne il valore in un punto. La funzione `derivata()` riceve quindi come argomento una funzione $f : \mathcal{R} \rightarrow \mathcal{R}$ e ritorna una funzione $f' : \mathcal{R} \rightarrow \mathcal{R}$; poiché il valore di ritorno deve essere una funzione (completa del proprio ambiente non locale) e non un semplice puntatore a funzione, `derivata()` non può essere implementata in C (vedere al proposito il precedente esempio con `sommainteri()`). Usando un linguaggio simile al C ma che supporta funzioni di ordine superiore (con la sintassi descritta in precedenza), si può implementare come mostrato in Figura 1.24. I lettori più attenti si saranno sicuramente accorti del fatto che `derivata()` non è nient'altro che la forma "curryificata" di `calcoladerivata()`[6]!

A titolo di esempio, in Figura 1.25 si riporta l'implementazione di `derivata()` in C++ (usando le estensioni funzionali fornite da C++11, come le lambda expression). Dal codice, si possono vedere alcune cose interessanti. Prima di tutto, la classe "`std::function`" fornita dal linguaggio C++ (a partire dallo standard C++11) permette di utilizzare una sintassi più semplice ed intuitiva di quella dei puntatori

[6]Provando a re-implementare le due funzioni con un qualsiasi linguaggio funzionale, la cosa diventa ancora più evidente

```
#include <iostream>
#include <functional>

double f(double x)
{
  return x * x + 2 * x + 1;
}

std::function<double (double)> derivata(std::function<double (double)> f)
{
  const double epsilon = 0.0001;

  return [epsilon, f](double x) {
    return (f(x + epsilon) - f(x)) / epsilon;
  };
}

int main()
{
  double x = 2;
  std::function<double (double)> f1;

  std::cout << "f'(" << x << ") = " << (derivata(f))(x) << std::endl;

  f1 = derivata(f);
  std::cout << "f'(" << x << ") = " << f1(x) << std::endl;

  return 0;
}
```

Figura 1.25: Calcolo della derivata di una funzione in C++.

a funzione del C. Inoltre, tale classe non memorizza semplicemente un puntatore a funzione (o una funzione), ma un'intera chiusura (formata dalla funzione e dal suo ambiente); tale chiusura memorizzerà (all'interno dell'oggetto di classe "`std::function`") i valori di `epsilon` ed `f`. Per finire, è da notare come la sintassi "`[epsilon,f](double x)`" permetta di definire una *funzione anonima*, che viene memorizzata (assieme al suo ambiente) nel valore di ritorno di "`derivata()`" (questo costrutto è chiamato "lamba function", per motivi che diventeranno chiari leggendo le prossime sezioni).

1.5 Linguaggi di Programmazione Funzionale

Riassumendo quanto detto fin qui, il paradigma di programmazione funzionale si contraddistingue per l'assenza del concetto di variabili modificabili (in modo da eliminare gli effetti collaterali ad essi legati), il conseguente utilizzo della ricorsione al posto dell'iterazione ed il fatto che i programmi siano composti da espressioni e non da comandi (che hanno effetti collaterali). Un'importante conseguenza della mancanza di effetti collaterali è la possibilità di eseguire i programmi usando un meccanismo di riduzione / sostituzione invece che modificando lo stato della macchina astratta. Le espressioni e funzioni diventano inoltre entità esprimibili e memorizzabili, portanto ad un'altra interessante caratteristica della programmazione funzionale: la presenza di funzioni di ordine superiore (funzioni che possono operare su altre funzioni, ricevendo funzioni come argomenti e generando funzioni come risultati). L'utilizzo di funzioni di ordine superiore diventa addirittura necessario se si limita ad uno il numero di possibili argomenti per una funzione (utilizzando la tecnica del currying per implementare funzioni a più argomenti).

Sebbene questo stile di programmazione sia utilizzabile anche con linguaggi "più tradizionali", esistono dei linguaggi, detti *linguaggi di programmazione funzionali* che cercano di favorirne (o addirittura forzarne) l'utilizzo. Caratteristica fondante di questa classe di linguaggi è quindi il tentativo di ridurre al minimo gli effetti collaterali: sebbene alcuni linguaggi funzionali prevedano il concetto di variabile

modificabile, essi possono essere utilizzati anche senza fare uso di tale costrutto; alcuni linguaggi funzionali (come per esempio Haskell), poi, non prevedono proprio l'esistenza di variabili modificabili. Tali linguaggi sono detti *linguaggi funzionali puri.* Altra caratteristica fondamentale dei linguaggi funzionali è poi la possibilità di trattare in modo omogeneo codice e dati (oltre ai tradizionali tipi di dato esiste il "tipo funzione", esistono funzioni di ordine superiore, etc...).

In questa sede non vengono discusse le specifiche sintassi e/o semantiche dei vari linguaggi di programmazione funzionale, ma per tali dettagli si rimanda a specifici documenti.

Nei linguaggi funzionali esiste quindi un ambiente che contiene legami (binding) fra nomi e valori (di tipi scalari, strutturati, o funzione). Tali legami vengono creati quando si invoca una funzione (legame fra parametro formale ed espressione passata come parametro attuale), ma poiché è spesso utile (anche se non strettamente necessario - ma di questo parleremo poi) avere anche un ambiente non locale ad alcuna funzione ogni linguaggio di programmazione funzionale fornisce qualche modo per associare nomi a valori in un ambiente globale. Un linguaggio funzionale fornisce quindi:

- Un sistema di tipi (type system), vale a dire un insieme di tipi predefiniti (e di valori per questi tipi), più un insieme di operatori per combinare valori dei vari tipi costruendo espressioni ed una serie di regole per assegnare un tipo ad ogni valore e verificare la corretta tipizzazione di ogni espressione;

- Un qualche modo per definire funzioni, vale a dire un meccanismo di *astrazione* che data un'espressione la astrae rispetto al valore di un parametro formale (tipicamente si considera un solo argomento e si usa il currying). Questo meccanismo è spesso fornito da un operatore che fornisce come risultato un valore di tipo funzione;

- Un modo per associare nomi a valori nell'ambiente globale (`define` in scheme, `val` in Standard ML, etc...)

- Dipendentemente dal sistema di tipi usato dal linguaggio, vengono forniti modi per definire nuovi tipi di dato, per combinare tipi esistenti tramite prodotto cartesiano (tuple) o unione, etc...

Il modo in cui un linguaggio di programmazione implementa questi punti da' origine a differenti linguaggi di programmazione funzionale, con caratteristiche diverse. Per esempio, il sistema di tipi utilizzato dal linguaggio può variare, facendo controlli più o meno stretti, a tempo di esecuzione (come per esempio in Lisp, scheme e linguaggi simili) o a tempo di compilazione (come per esempio in Haskell, Standard ML, ma anche C++). Un sistema di tipi più dinamico (e meno "stretto") aumenterà la possibilità di commettere errori di programmazione (non riuscendo ad identificare alcune classi di errori legati ai tipi) ed introdurrà un maggiore overhead (effettuando alcuni controlli a runtime invece che a tempo di compilazione), ma d'altra parte risulterà più flessibile e potente (per esempio, implementare un Y combinator in Lisp o scheme è molto più facile che implementarlo in Haskell, Standard ML o C++).

Per quanto riguarda invece la definizione di funzioni, alcuni lettori saranno probabilmente più abituati all'approccio seguito da molti linguaggi di programmazione imperativi, che generalmente forniscano un unico meccanismo che contemporaneamente specifica il corpo della funzione e crea nell'ambiente un legame fra il corpo della funzione ed il suo nome. Al contrario, i più comuni linguaggi di programmazione funzionali distinguono due diversi meccanismi: l'astrazione, che genera un valore di tipo funzione senza assegnargli un nome (una cosiddetta "funzione anonima" — si consideri come esempio il costrutto lambda expression del C++, accennato in precedenza) ed un secondo meccanismo che permette di modificare l'ambiente (anche globale) associando un nome ad un generico valore (che può essere di tipo funzione o altro). Come importante conseguenza, *nel momento in cui si definisce il corpo di una funzione tale funzione non è ancora associata ad un nome.* Questo può chiaramente creare dei contrattempi quando si cerca di definire una funzione ricorsiva, come si vedrà meglio in futuro parlando di λ calcolo.

Per finire, un aspetto fondamentale dei linguaggi di programmazione funzionale è chiaramente l'invocazione di funzione. Sebbene possa sembrarci una cosa semplice e naturale, merita un minimo di discussione: se ci troviamo infatti di fronte alla definizione della funzione `fattoriale(unsigned int n)` come "(n == 0) ? 1 : n $*$ fattoriale (n $-$ 1)", ci viene infatti naturale pensare che " fattoriale (4)" sia valutata come

(4 == 0) ? 1 : 4 $*$ fattoriale (4 $-$ 1) $\rightarrow$ 4 $*$ fattoriale (3) $\rightarrow$ ...

andando in pratica ad eseguire subito l'operazione $4 - 1 = 3$, ma questa non è l'unica soluzione possibile. Un'alternativa potrebbe essere

(4 == 0) ? 1 : 4 $*$ fattoriale (4 $-$ 1) $\rightarrow$ 4 $*$ fattoriale (4 $-$ 1) $\rightarrow$
$\rightarrow$ 4 $*$ ((4 $-$ 1 == 0) ? 1 : (4 $-$ 1) $*$ fattoriale (4 $-$ 1 $-$ 1)) $\rightarrow$
$\rightarrow$ 4 $*$ ((4 $-$ 1) $*$ fattoriale (4 $-$ 1 $-$ 1)) $\rightarrow$ ...

eseguendo in pratica le operazioni aritmetiche solo quando strettamente necessario.

Un linguaggio di programmazione funzionale deve quindi specificare in modo chiaro come e quando valutare le espressioni. In particolare, si possono avere:

- strategie di valutazione *eager*, in cui quando una funzione `f` è applicata ad un'espressione `e` l'espressione è valutatata (riducendola ad un valore irriducibile) prima di invocare la funzione
- strategie di valutazione *lazy*, in cui quando una funzione `f` è applicata ad un'espressione `e` la funzione è invocata senza prima provare a ridurre l'espressione che riceve come argomento (passando quindi un'espressione non valutata e non un valore irriducibile).

Si noti come la prima strategia coincide sostanzialmente con il passaggio di parametri per valore (il parametro attuale è valutato fino a divenire un valore irriducibile prima di invocare la funzione), mentre la seconda strategia coincide col passaggio di parametri per nome (alla funzione viene passato un *thunk*, vale a dire un'espressione senza variabili locali, che non viene valutata prima di essere effettivamente utilizzata).

Generalmente, i linguaggi funzionali puri (come Haskell) tendono a prediligere valutazioni lazy, mentre linguaggi funzionali che ammettono effetti collaterali (come ML, Scheme, etc...) sono costretti ad usare strategie di valutazione eager. Per capire come mai, si consideri un linguaggio con variabili modificabili (e quindi "non troppo funzionale") ed una funzione `bad_bad_function()` definita come

```
void bad_bad_function(void)
{
  x++;
}
```

dove "`x`" è una variabile globale. Se il valore di "`x`" è inizialmente 0, quanto vale "`x`" dopo aver invocato some_function(bad_bad_function(), bad_bad_function())? Se si usa una strategia di valutazione lazy, non è possibile dirlo a priori, perché dipende da quante volte `some_function()` valuta i suoi argomenti (mentre se si usa una strategia eager è possibile dire che `bad_bad_function()` verrà invocata una volta per argomento e quindi il valore di "`x`" sarà 2).

Sebbene il precedente esempio riguardi sostanzialmente un non-problema (i linguaggi di programmazione funzionale non dovrebbero implementare variabili modificabili), le operazioni di I/O rappresentano problemi ben più reali e seri. Qualsiasi operazione di input o output costituisce infatti un effetto collaterale ed in presenza di valutazione lazy crea quindi dei non-determinismi nel comportamento del programma (quale sarebbe l'output di some_function(bad_bad_function(), bad_bad_function()) se `bad_bad_function()` stampasse qualcosa sullo schermo?). I linguaggi funzionali puri generalmente affrontano questo problema modellando le funzioni di I/O come funzioni che ricevono un'entità "mondo" in ingresso e producono in output una versione modificata di tale "mondo". I linguaggi che prevedono valutazione lazy forniscono poi vari tipi di meccanismi per serializzare l'esecuzione delle operazioni di I/O, in modo da rendere deterministiche le interazioni del programma col mondo esterno. Tipicamente, la serializzazione dell'esecuzione di due funzioni `f1()` ed `f2()` è implementabile facendo si che `f2()` sia invocata ricevendo in ingresso l'output di `f1()` (l'entità "mondo", per esempio). Poiché questa soluzione porta a notazioni complesse e poco intuitive, alcuni linguaggi come Haskell forniscono una sintassi semplificata per questi meccanismi, che ricorda la sintassi dei linguaggi imperativi (per fare questo, Haskell utilizza strumenti matematici complessi come le monadi della teoria delle categorie).

Gli effetti pratici dell'uso di differenti strategie di valutazione sono visibili, per esempio, provando ad implementare il Y combinator (un'implementazione in Haskell non darà problemi, mentre un'implementazione in Standard ML o Scheme non sarà possibile e costringerà ad implementare un differente combinator, come per esempio Z).

E' dimostrabile che se il meccanismo di valutazione lazy e quello eager riducono entrambi un'espressione ad un valore, allora il valore ottenuto tramite valutazione lazy e quello ottenuto tramite valutazione eager coincidono (a tale proposito, vedere il teorema di Church-Rosser). Inoltre, se la valutazione lazy porta ad una ricorsione infinita allora anche la valutazione eager porta ad una ricorsione infinita (ma d'altra parte esistono situazioni in cui la valutazione eager genera ricorsione infinita mentre la valutazione lazy permette di ridurre l'espressione ad un valore - ancora, vedere Y combinator).

1.6 Computazione per Riduzione

Basandosi sui meccanismi appena descritti, un programma scritto secondo il paradigma funzionale può essere eseguito tramite la computazione per riduzione, implementata ripetendo 2 operazioni:

- Ricerca di nomi nell'ambiente (e sostituzione testuale di un nome con il corrispondente valore funzionale - rappresentato come astrazione)
- Applicazione di funzioni (sostituzione testuale del parametro formale col parametro attuale)

Un programma funzionale è quindi rappresentato come un insieme di definizioni ed operazioni di modifica dell'ambiente (creazioni di binding) che per essere processate possono richiedere la valutazione di espressioni. La computazione di tale programma verrà effettuata dalla macchina astratta tramite una serie di riscritture / riduzioni che porteranno a semplificare fino a che non si arriva a forme semplici non ulteriormente riducibili (dette *valori*).

Per rendere meno ostico l'utilizzo di tecniche di programmazione funzionale vengono spesso forniti altri costrutti che pur non essendo strettamente necessari semplificano notevolemente lo sviluppo del codice. Esempi sono:

- Un modo per modificare l'ambiente locale (generalmente, il costrutto `let`);
- Un meccanismo analogo al fixed point operator `fix`, che permette di definire funzioni ricorsive;
- Alcuni costrutti che costituiscono uno "zucchero sintattico" per definire funzioni a più argomenti (nascondendo l'utilizzo dell currying), etc...

Rigurardo al costrutto (generalmente chiamato `let`) usato per modificare l'ambiente locale in cui viene valutata un'espressione, si noti che questo costrutto non è strettamente necessario perché implementabile tramite chiamata a funzione e passaggio parametri. Per esempio, si consideri un generico costrutto "`let` *x* `=` *e1* `in` *e2*" (dove "*x*" è un generico nome mentre "*e1*" e "*e2*" sono due espressioni) che associa il nome "*x*" all'espressione "*e1*" durante la valutazione di "*e2*". Questo è implementabile definendo una funzione `f()` con parametro formale "*x*" e corpo "*e2*" ed invocando tale funzione con parametro attuale "*e1*"[7]. Ancora meglio, si può usare una funzione anonima: usando la sintassi di Standard ML "`let` *x* `=` *e1* `in` *e2*" diventa "(**fn** x => e2) e1".

Il costrutto equivalente al fixed point operator `fix` è invece molto utile per semplificare la definizione di funzioni ricorsive: come accennato in precedenza (e come diventerà più chiaro studiando il λ calcolo), senza questo meccanismo la definizione di funzioni ricorsive non sarebbe possibile in modo semplice: il nome di una funzione non può essere usato nella sua definizione, perché non è ancora legato a nessun valore nell'ambiente globale. Per definire funzioni ricorsive sarebbe necessario implementare un fixed point combinator (come Y o Z) ed applicare tale operatore alla "versione chiusa" della funzione che si vuole definire. Questo meccanismo (chiamato `val rec` o `fun` in Standard ML, `letrec` in Scheme, etc...) permette invece di usare la ricorsione in modo diretto.

In linguaggi che usano sistemi di tipi più potenti è spesso fornito anche un meccanismo di *pattern matching* che permette di manipolare valori di tipi definiti dall'utente (per esempio, distinguendo i vari varianti di un tipo, etc...).

1.7 Un Linuguaggio Funzionale Minimale

Per concludere, i lettori più curiosi potrebbero chiedersi come sia fatto il più semplice linguaggio di programmazione funzionale possibile, che non contenga funzionalità "di alto livello" utili per rendere il codice più leggibile ma non strettamente necessarie. In altre parole, cosa si ottiene rimuovendo da un linguaggio di programmazione funzionale le caratteristiche non indispensabili per la Turing-completezza, come

- la presenza di un ambiente globale (che come detto semplifica la definizione di funzioni ricorsive)
- la "tipizzazione stretta' e la presenza di tipi di dati più complessi (che aumentano la leggibilità del codice ma non sono fondamentali: si noti come anche nel paradigma di programmazione imperativo il linguaggio Assembly non definisca tipi di dato ma consideri solo valori binari)
- i vari costrutti che costituiscono "zucchero sintattico".

Quel che resta è un linguaggio in cui i programmi sono espressioni (pure!) composte semplicemente da:

1. nomi (termini irriducibili)

[7] Questo trucco di sostituire una variabile con un parametro formale ed il suo valore col parametro attuale è spesso usato per reimplementare codice imperativo usando il paradigma funzionale.

2. definizioni di funzioni (il concetto di astrazione)
3. applicazioni di funzioni

Per quanto riguarda i nomi, la scelta più semplice e minimale è quella di usare singole lettere minuscole, anche se talvolta si permettono di usare identificatori composti da più caratteri.

Per quanto riguarda l'applicazione di funzioni, i programmatori che hanno familiarità con linguaggi della famiglia del C (C, C++, Java, ...) sono abituati ad indicare con "`f(x)`" l'applicazione della funzione "`f`" al parametro attuale "`x`" (si ricordi che per semplicità si possono considerare solo funzioni ad un argomento). Le parentesi attorno al parametro attuale sono però inutili, quindi si potrebbe anche usare la sintassi "`f x`", che è spesso preferita. La se l'applicazione di funzioni associa a sinistra, la composizione $g \circ f$ delle funzioni f e g può essere quindi implementata come "`g (f x)`" invece che "`g(f(x))`". La sintassi "`g f x` è invece equivalente a "`(g(f))(x)`" (e questo, come si vedrà, rende più naturale la sintassi del currying). Alcuni linguaggi della famiglia LISP prevedono invece le parentesi attorno all'applicazione di funzione invece che attorno al parametro attuale ("`(f x)`" invece che "`f(x)`"); in questo caso, $g \circ f$ diventa "`(g (f x))`".

Per finire, il linguaggio deve prevedere di costruire espressioni che vengono valutate a funzioni (permettendo in qualche modo di "definire" una funzione a partire da un'espressione "`e`" ed un parametro formale "`x`"). Indipendentemente dalla sintassi che il linguaggio usa, questo costrutto *astrae* l'espressione "`e`" dallo specifico valore del parametro formale "`x`"; deve quindi contenere una qualche keyword specifica del linguaggio (che può essere la lettera greca λ, il simbolo "\", la parola "`fn`", una sequenza di parentesi quadre, tonde e graffe, o altro), il nome del parametro formale e l'espressione da astrarre. Esempi possono essere "$\lambda x.e$", "`\x -> e`", "`fn x => e`", "`[](auto x) { e }`", "`(lambda (x) (e))`" o simili...

Non prevedendo una tipizzazione stretta, questo "linguaggio minimale" conosce solo generiche "funzioni" che operano su espressioni (ricevono un'altra funzione generica come argomento e generano una funzione generica come risultato), senza che siano specificati in modo più preciso dominio e codominio di tali funzioni (tali insiemi coincidono con l'insieme delle espressioni che compongono il linguaggio). Ma sorprendentemente il linguaggio risultante (noto come λ calcolo) è ancora Turing completo: è possibile codificare valori naturali, booleani e di altro tipo usando solo funzioni ed è possibile usare vari tipi di fixed point combinator per implementare funzioni ricorsive anche in assenza di ambiente non locale. Per questo motivo, il λ calcolo è spesso considerato come una sorta di "Assembly dei linguaggi di programmazione funzionale".

Capitolo 2

Primo Incontro col Lambda Calcolo

2.1 Introduzione al Lambda-Calcolo

Il λ calcolo è un formalismo (o, se preferiamo vederlo dal punto di vista informatico, un linguaggio di programmazione) che permette di definire i concetti fondamentali della programmazione funzionale: funzioni, definizione di funzioni ed applicazione di funzioni.

Se vediamo il λ calcolo come un linguaggio di programmazione, si può notare come esso introduca i meccanismi di base necessari per scrivere programmi funzionali senza introdurre le astrazioni che caratterizzano i linguaggi di programmazione funzionale di più alto livello. Per questo, il λ calcolo può essere visto come l'equivalente del linguaggio Assembly per la programmazione funzionale. E' interessante però notare come esistano linguaggi funzionali di livello ancora più basso, perché introducono ancora meno astrazioni (per esempio, vedremo che il costrutto di definizione di funzioni può essere omesso).

Da un altro punto di vista, il λ calcolo può essere visto come il fondamento teorico per la programmazione funzionale, in quanto è possibile dimostrare che è Turing-completo. Questo risultato ha una notevole importanza, perché significa che il paradigma di programmazione funzionale permette di implementare qualsiasi algoritmo calcolabile (vale a dire: ha la stessa potenza espressiva del paradigma di programmazione imperativo).

Riassumendo, gli elementi di base del λ calcolo sono semplicemente nomi, il concetto di astrazione (definizione di funzioni) e l'operazione di applicazione di funzione. Non esistono quindi i concetti di tipo di dato, di ambiente globale, e simili. Non esistendo diversi tipi di dato, gli elementi di base del λ calcolo sono generiche "funzioni", che ricevono un'altra funzione come argomento e generano una funzione come risultato. Il dominio ed il codominio di tali funzioni sono generiche espressioni (meglio, λ-espressioni) e non sono specificati espressamente.

Vedremo come esista una versione tipizzata del λ calcolo, in cui il tipo di una funzione è caratterizzato dal dominio e dal codominio della funzione stessa (come tradizionalmente fatto nei vari corsi di analisi o algebra). Paradossalmente, però, tale formalismo perde il potere espressivo del λ calcolo originario e non è più Turing-completo.

L'idea di base del λ calcolo è quella di esprimere algoritmi (o codificare programmi) sotto forma di espressioni, chiamate λ-espressioni nel seguito.Come vedremo, l'esecuzione di un programma consiste allora nella valutazione di una λ-espressione (utilizzando un meccanismo di semplificazione chiamato "riduzione"). Vediamo quindi come sono composte le λ-espressioni. La sintassi incredibilmente semplice riflette il fatto che le λ-espressioni sono costituite a partire dai tre semplici concetti già citati:

1. Variabili (che in realtà rappresentano funzioni). Constuiscono gli elementi terminali del linguaggio e sono indicate tramite nomi (identificatori), che verranno rappresentati da da singole lettere corsive (per esempio, "x", "y" o "f") in seguito

2. Astrazioni, che permettono di specificare che una variabile "x" è un argomento nell'espressione che segue. Tecnicamente, si dice che un'astrazione "lega" una variabile in un'espressione (il perché di questa terminologia diverrà chiaro in seguito). In pratica un'astrazione "crea" (informalmente parlando) una funzione a partire da una λ espressione, specificando l'argomento della funzione (variabile legata)

3. Applicazioni di funzioni. Rappresentano l'operazione inversa rispetto all'astrazione e permettono di trasformare un'astrazione ed una λ espressione in una singola λ espressione rimuovendo la variabile legata (in realtà, l'intera astrazione viene rimossa)

Usando la notazione BNF, la sintassi di una λ-espressione è:

```
<expression> ::= <name>                          ; lettera minuscola
              | ( λ<name>.<expression>)          ; astrazione
              | (<expression> <expression>)      ; applicazione
```

Questo è equivalente alla seguente definizione induttiva:

- Un nome / variabile / funzione (indicato con una singola lettera corsiva in seguito) è una λ espressione
- Se e è una λ espressione ed x è un nome, allora $(\lambda x.e)$ è una λ espressione
- Se e_1 ed e_2 sono λ espressioni, allora $(e_1 e_2)$ è una λ espressione

In base alle precedenti definizioni, i seguenti sono esempi di λ espressioni:

- x (variabile / funzione)
- $(\lambda x.(xy))$ (astrazione: lega la variabile "x" nell'espressione "xy", trasformando tale espressione in una funzione della variabile "x")
- $((xy)z)$ (applicazione: applica la funzione "x" ad "y", poi applica il risultato a "z")
- $(\lambda x.(xy))z$ (espressione più complessa)

In base alle definizioni date fin qui, ogni astrazione o applicazione di funzione andrebbe racchiusa fra parentesi (per rendere la sintassi meno ambigua); in realta', si assumono le seguenti convenzioni per ridurre il numero di parentesi utilizzate:

- L'applicazione di funzione associa a sinistra: $((xy)z)$ è quindi equivalente a xyz
- L'operatore "λ" associa a destra ed ha precedenza inferiore rispetto all'applicazione: $(\lambda x.(xy))$ è quindi equivalente a $\lambda x.xy$

Un'altra convenzione spesso usata in letteratura è che le variabili legate da più astrazioni immediatamente successive vengono raggruppate; per esempio, $\lambda x.\lambda y.f$ può essere scritta come $\lambda xy.f$. Questa convenzione non verrà però usata nel seguito e manterremo una sola variabile per λ.

E' interessante notare come la sintassi del λ calcolo permetta di distinguere la definizione di una funzione dalla sua applicazione: nella notazione matematica comunemente usata, infatti, il termine "$f(x)$" si utilizza sia per indicare che la funzione "$f()$" è applicata al valore "x" che per definire tale funzione (come in "$f(x) = x^2$"). Nel λ calcolo, invece, "fx" rappresenta l'applicazione di "f" ad "x", mentre "$\lambda x.f$" rappresenta la definizione di una funzione con argomento "x".

Un'altra cosa interessante da notare è che data l'assenza di un ambiente globale non è possibile *creare dinamicamente* associazioni fra λ espressioni e nomi *a livello globale*. In altre parole, oltre a non esistere (come in tutti i linguaggi funzionali) il concetto di assegnamento di valore a variabile modificabile, non esiste nel λ calcolo non esiste neanche l'equivalente del costtrutto `val` di standard ML (o l'operatore "`=`" di Haskell), ne' esiste niente di simile alla dichiarazione di variabili (neanche immutabili). Per comodità è possibile usare nomi simbolici per λ espressioni complesse (vedere il λ calcolo applicato, più avanti), ma questi sono simili a macro, definite in modo statico, non a legami in un ambiente globale che possa variare dinamicamente a tempo di esecuzione.

Come risultato, nel λ calcolo si possono creare solamente espressioni anonime e *funzioni anonime* (analogamente a quanto fatto in standard ML con il costrutto `fn`, in Haskell con "\" o in C++ con le lambda expression "`[](...){...}`"). Questo sembrerebbe implicare che il λ calcolo non permette di definire funzioni ricorsive (e di conseguenza non è Turing completo); vedremo in seguito come sia invece possibile definire funzioni che richiedono la ricorsione utilizzando il concetto di *punto fisso* e di *fixed point combinator*.

Gli unici legami nome / valore che si possono creare dinamicamente sono i legami *a livello locale* fra parametri formali e parametri attuali che si creano durante l'applicazione di funzione. Questo significa che nel λ calcolo esiste almeno il concetto di ambiente locale.

2.2 Semantica del Lambda Calcolo

Come precedentemente anticipato, il λ calcolo permette di codificare programmi come espressioni, che vengono "eseguite" valutandole tramite un processo chiamato riduzione. A livello informale, si può dire che tale processo è basato sui significati che sono stati associati ai vari elementi di base che compongono una λ espressione; per poter però definire in modo più formale la semantica del λ calcolo, occorre prima introdurre alcuni concetti, come il concetto di variabili libere (free variables) e variabili legate (bound variables).

A livello intuitivo, una variabile "x" è legata da un costrutto "$\lambda x.E$" (dove E è una generica λ espressione), mentre è libera in un'espressione "E" se in "E" non esiste alcun costrutto λ che la lega. Per dare una definizione più formale, bisogna invece rifarsi alla definizione ricorsiva delle λ espressioni: in particolare, se $\mathcal{B}(E)$ rappresenta l'insieme delle variabili legate in "E" e $\mathcal{F}(E)$ rappresenta l'insieme delle variabili libere in "E", si può dire che:

- Per ogni variabile "x", $\mathcal{F}(x) = \{x\}$ e $\mathcal{B}(x) = \emptyset$
- $\mathcal{F}(E_1E_2) = \mathcal{F}(E_1) \cup \mathcal{F}(E_2)$; $\mathcal{B}(E_1E_2) = \mathcal{B}(E_1) \cup \mathcal{B}(E_2)$
- $\mathcal{F}(\lambda x.E) = \mathcal{F}(E) - \{x\}$; $\mathcal{B}(\lambda x.E) = \mathcal{B}(E) \cup \{x\}$

Sostanzialmente, questa definizione dice che se un'espressione è composta da una sola variabile, quella variabile è libera; componendo due espressioni (applicando un'espressione ad un'altra) non si cambia lo stato delle variabili (variabili libere rimangono libere e variabili legate rimangono legate) e l'operatore "$\lambda x.E$" lega la variabile "x" nell'espressione "E" (rimuove "x" dall'insieme delle variabili libere di "E" e la aggiunge all'insieme delle variabili legate). La si dice che l'operatore λ lega la variabile "x" in "$\lambda x.E$" perché causa l'aggiunta nell'ambiente locale di "E" di un legame fra il nome "x" (nome del parametro formale) ed il parametro attuale quando "$\lambda x.E$" viene applicata al parametro attuale.

In base a questa semplice definizione ricorsiva è possibile calcolare l'insieme delle variabili libere e delle variabili legate per ogni λ espressione. Una λ espressione che non contiene variabili libere ma è composta solo da variabili legate) è chiamata "combinator" ed ha l'importante proprietà che il risultato della sua valutazione dipende unicamente dagli argomenti (parametri attuali) usati per valutarla. Più formalmente, una λ espressione E è un combinator se $\mathcal{F}(E) = \emptyset$.

Si può ora definire il concetto di α equivalenza fra due λ espressioni. Informalmente, due λ espressioni E_1 e E_2 sono α equivalenti ($E_1 \equiv_\alpha E_2$) se differiscono solo per il nome di un parametro. Questo significa che quando si definisce una funzione il nome dell'argomento della funzione non è importante (usando una notazione matematica a cui siamo abituati, $f_1(x) = x^2$ e $f_2(y) = y^2$ rappresentano la stessa funzione); quindi, per esempio, $\lambda x.xy \equiv_\alpha \lambda z.zy$. La definizione corretta di α equivalenza è ovviamente più complessa, perché, per esempio, $\lambda x.x\lambda x.xy$ non è α equivalente a $\lambda z.z.\lambda x.zy$ ma a $\lambda z.z.\lambda x.xy$. In pratica, $\lambda x.E$ è α equivalente a $\lambda z.E[x \to z]$, dove $E[x \to z]$ rappresenta l'espressione "E" in cui la variabile "x" è sostituita da "z" solo se è libera:

- Se "x" e "y" sono variabili ed E è una λ espressione, $x[x \to E] = E$ e $y \neq x \Rightarrow y[x \to E] = y$
- Date due λ espressioni E_1 ed E_2, $(E_1E_2)[x \to E] = (E_1[x \to E]E_2[x \to E])$
- Se "x" e "y" sono variabili ed E è una λ espressione,
  - $y \neq x \wedge y \notin \mathcal{F}(E') \Rightarrow (\lambda y.E)[x \to E'] = \lambda y.(E[x \to E'])$
  - $y = x \Rightarrow (\lambda y.E)[x \to z] = \lambda y.E$

Tornando all'esempio precedente, si può notare come la regola "$y = x \Rightarrow (\lambda y.E)[x \to z] = \lambda y.E$" permetta di ottenere il risultato corretto: $\lambda x.x\lambda x.xy \equiv_\alpha \lambda z.(x\lambda x.xy)[x \to z] = \lambda z.x[x \to z](\lambda x.xy)[x \to z] = \lambda x.z\lambda x.xy$ come ci aspettiamo. E' interessante notare anche come la regola "$y \neq x \wedge y \notin \mathcal{F}(E') \Rightarrow (\lambda y.E)[x \to E'] = \lambda y.(E[x \to E'])$" contenga la condizione "$y \notin \mathcal{F}(E')$": tale condizione serve ad evitare sostituzioni errate come $(\lambda x.xy)[y \to x] = \lambda x.xx$ che porterebbero ad α equivalenze tipo $\lambda y.\lambda x.xy \equiv_\alpha \lambda x.\lambda x.xx$, chiaramente errate. Questo fenomeno, nel quale una variabile "y" libera in "$\lambda x.xy$" diventa legata dopo una sostituzione viene chiamato *cattura della variabile* (in quanto una semplice sostituzione trasforma una variabile libera in una variabile legata) e deve essere evitato durante le sostituzioni. Il meccanismo di sostituzione $E[x \to y]$ definito qui sopra viene quindi chiamato *sostituzione senza cattura* e può essere utilizzato per definire formalmente la relazione di α equivalenza:

$$\lambda x.E \equiv_\alpha \lambda y.E[x \to y]$$

Come suggerito dal nome, l'α equivalenza è una relazione di equivalenza: $E_1 \equiv_\alpha E_2$ è quindi una relazione *simmetrica*, *riflessiva* e *transitiva* fra λ espressioni:

- $E \equiv_\alpha E$
- $E_1 \equiv_\alpha E_2 \Rightarrow E_2 \equiv_\alpha E_1$
- $E_1 \equiv_\alpha E_2 \wedge E_2 \equiv_\alpha E_3 \Rightarrow E_1 \equiv_\alpha E_3$

La sostituzione senza cattura riveste un ruolo fondamentale nel λ calcolo, in quanto oltre che per le α equivalenze è utilizzata anche nel meccanismo di riduzione usato per semplificare le λ espressioni. Informalmente parlando, la riduzione di una λ espressione consiste nell'applicazione di funzioni, rimuovendo le astrazioni, come in $(\lambda x.xy)z \rightarrow zy$. Questo procedimento può apparire semplice, ma nasconde una serie di complicazioni; per esempio, la riduzione $(\lambda x.(x\lambda y.xy))y \rightarrow y\lambda y.yy$ è chiaramente sbagliata, perchè la "y" è stata catturata nel processo (questo è uno dei motivi per cui in precedenza è stato definito il meccanismo di sostituzione senza cattura!). Quindi, ogni volta che si ha un'astrazione (costrutto "$\lambda x.E$") applicata ad un'espressione E_1, si può usare una sostituzione **senza cattura** di E_1 in E (al posto di x) per ridurre la λ espressione eliminando l'astrazione.

Più formalmente, si definisce *redex* (*red*ucible *ex*pression) una λ espressione del tipo $(\lambda x.E)E_1$ e si dice che $E[x \rightarrow E_1]$ è il suo ridotto. In base a questo, si può definire la β riduzione "$\rightarrow_\beta$" come la sostituzione di un redex col suo ridotto:

$$(\lambda x.E)E_1 \rightarrow_\beta E[x \rightarrow E_1]$$

In base a questa sostituzione, potrebbe sembrare che ci siano dei redex non riducibili (il che sembra una contraddizione in termini) perché la riduzione senza cattura non può essere usata ("E" contiene un'altra astrazione che lega una variabile "y" ed "y" compare fra le variabili libere di E_1): per esempio, si consideri la λ espressione $(\lambda y.\lambda x.xy)(xz)$: questa espressione rappresenta chiaramente un redex, quindi si potrebbe provare a ridurla usando il meccanismo di β riduzione, che porterebbe a $(\lambda x.xy)[y \rightarrow (xz)]$. Si noti però che $x \in \mathcal{F}(xz)$, quindi nessuna delle regole presentate nella definizione del meccanismo di sostituzione senza cattura può essere usata (ancora: $(\lambda x.xy)[y \rightarrow (xz)] = \lambda x.x(xy)$ non è una sostituzione senza cattura, perché catturerebbe la "x" rossa). Come si procede allora per ridurre questo tipo di redex? Il concetto di α equivalenza ci viene in aiuto, permettendoci di rinominare le variabili legate nell'espressione E in modo che esse non compaiano fra le variabili libere di E_1. Tornando al nostro esempio:

$$(\lambda y.\lambda x.xy)(xz) \equiv_\alpha (\lambda y.\lambda k.ky)(xz) \rightarrow_\beta (\lambda k.ky)[y \rightarrow (xz)] = \lambda k.k(xz)$$

che stavolta appare una riduzione corretta (la sostituzione senza cattura $(\lambda k.ky)[y \rightarrow (xz)]$ ora è possibile perché $k \notin \mathcal{F}(xz)$).

Contrariamente alla α equivalenza, la β riduzione non è una relazione di equivalenza, in quanto non gode della proprietà riflessiva: $E_1 \rightarrow_\beta E_2$ non implica $E_2 \rightarrow_\beta E_1$. Una relazione di equivalenza (detta β equivalenza "$\equiv_\beta$") può però essere creata a partire dalla β riduzione calcolandone la chiusura riflessiva e transitiva. In pratica, $E_1 \equiv_\beta E_2$ significa che esiste una qualche catena di β riduzioni che "collegano" E_1 ed E_2 (β riducendo più volte E_1 ed E_2 è possibile arrivare ad una stessa espressione E).

Più formlamente, la β equivalenza $\equiv_\beta$ è definita in base alle seguenti regole:

- $E_1 \rightarrow_\beta E_2 \Rightarrow E_1 \equiv_\beta E_2$
- $\forall E, E \equiv_\beta E$
- $\forall E_1, E_2 : E_1 \equiv_\beta E_2, E_2 \equiv_\beta E_1$
- $E_1 \equiv_\beta E_2 \wedge E_2 \equiv_\beta E_3 \Rightarrow E_1 \equiv_\beta E_3$

Per finire, è interessante notare come una generica λ espressione in genere contenga molteplici redex e le regole del λ calcolo non definiscano un ordine in cui applicare le possibili β riduzioni. In questo caso è possible ridurre l'espressione seguendo un qualsiasi ordine per le β riduzioni (purché si rispettino le parentesi e le regole di associatività e precedenza fra operatori).

L'ordine in cui valutare i vari redex può essere quindi deciso definendo una strategia di valutazione che va ad aggiungersi alle regole di riduzione del λ calcolo (per esempio, procedere verso destra a partire dal redex più a sinistra, o partire dal redex "più interno", etc...). Da queste regole derivano poi le varie strategie di valutazione (lazy vs eager, per nome vs per valore, etc...) utilizzate dai linguaggi di programmazione di più alto livello.

Fortunatamente, esiste un teorema (Teorema di Church-Rosser) che ci assicura che se una λ espressione E può essere ridotta ad E_1 tramite 0 o più β riduzioni ed E può essere ridotta a $E_2 \neq E_1$ tramite 0 o più β riduzioni, allora esiste E_3 tale che sia E_1 che E_2 possono essere ridotte ad E_3 tramite 0 o più β riduzioni ($E \rightarrow_\beta ... \rightarrow_\beta E_1 \wedge E \rightarrow_\beta ... \rightarrow_\beta E_2 \Rightarrow \exists E_3 : E_1 \rightarrow_\beta ... \rightarrow_\beta E_3 \wedge E_2 \rightarrow_\beta ... \rightarrow_\beta E_3$).

Questo teorema implica anche che se E è riducibile ad una forma normale (λ espressione che non contiene più alcun redex), allora tale forma normale non dipende dall'ordine delle β riduzioni. In altre parole, ogni λ espressione E ha alpiù una forma normale. Si noti l'utilizzo del termine "*alpiù*", in quanto esistono λ espressioni che non possono essere ridotte ad una forma normale (il processo di riduzione non termina mai). Un esempio tipico è il combinator $\Omega = \omega\omega$, dove $\omega = \lambda x.xx$:

$$\Omega = \omega\omega = (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (xx)[x \rightarrow (\lambda x.xx)] = (\lambda x.xx)(\lambda x.xx) = \omega\omega = \Omega$$

quindi, $\Omega \rightarrow_\beta \Omega$!!! Questo è l'equivalente di un ciclo infinito in un linguaggio imperativo, o di una ricorsione infinita in un linguaggio funzionale. L'esistenza di questo tipo di espressioni (che generano una riduzione infinita) è necessaria per la Turing-completezza del λ calcolo (la macchina di Turing permette di codificare computazioni infinite; se il λ calcolo non permettesse di codificare riduzioni infinite, non potrebbe implementare tali programmi della macchina di Turing).

2.3 Codifica di Linguaggi di Alto Livello

Dopo aver visto le più importanti definizioni del λ calcolo ed il funzionamento del meccanismo di riduzione, è abbastanza difficile riuscire a capire come un formalismo così semplice ed apparentemente poco espressivo possa essere Turing completo (cosa ribadita più volte in precedenza). In effetti, potrebbe sembrare che il λ calcolo possa essere utile solo per manipolare funzioni o cose simili.

La cosa diventa meno sorprendente se ricordiamo che siamo abituati al fatto che qualsiasi programma scritto in un linguaggio di alto livello sia trasformato in Assembly (tramite un processo di compilazione o di interpretazione) per essere eseguito da una CPU fisica. Così come il λ calcolo permette di lavorare solo con funzioni (e di compiere operazioni di riduzione relativamente semplici su espressioni composte solo da funzioni, astrazioni ed applicazioni) anche il linguaggio Assembly permette di operare solo su numeri binari (memorizzati nei registri della CPU o in RAM) e non ha il concetto di tipi di dato o di ambiente globale. Eppure non abbiamo problemi a pensare che un programma scritto in linguaggio di alto livello dotato di ambiente globale e con tipizzazione stretta venga convertito in Assembly: si tratta "solo" di implementare tutti i concetti di alto livello che ci servono a partire da semplici istruzioni Assembly che operano su registri o memoria. Allo stesso modo, gli stessi concetti di alto livello possono essere implementati usando solo funzioni, astrazioni ed applicazioni di funzioni.

In generale, per codificare le varie astrazioni di alto livello si useranno dei combinator (che, come già detto, sono delle λ espressioni in cui non compaiono variabili libere). Questo perché qualsiasi tipo di codifica non deve dipendere dal contesto (quindi, non deve fare riferimento ad alcun simbolo che non sia un parametro formale / argomento dell'espressione). A titolo di esempio, alcuni combinator notevoli noti in letteratura sono:

- Il combinator che rappresenta la funzione identità: $I = \lambda x.x$
- Il combinator che rappresenta la composizione di funzione: $B = \lambda f.\lambda g.\lambda x.f(gx)$
- $K = \lambda x.\lambda y.x$
- $S = \lambda f.\lambda g.\lambda x.fx(gx)$
- $\omega = \lambda x.xx$
- $\Omega = \omega\omega$
- $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

L'importanza del combinator Y è fondamentale per il λ calcolo, in quanto si tratta di un cosiddetto *fixed point combinator*, come verrà spiegato in seguito. I combinator K ed S sono invece interessanti in quanto permettono di definire un sottoinsieme del λ calcolo (chiamato "calcolo SK" o "calcolo SKI") nel quale non si utilizza esplicitamente il costrutto λ di astrazione (!!!). Anche questo verrà spiegato meglio in seguito. Prima di proseguire, si noti (ancora!) come il simbolo "=" usato informalmente qui sopra per definire i vari combinator rappresenti semplicemente un qualche tipo di uguaglianza / equivalenza (indica,

per esempio che scrivere "I" è equivalente a scrivere "$\lambda x.x$") e non sia un costrutto definito formalmente nel λ calcolo per modificare un qualche tipo di ambiente o creare legami fra nomi e simboli (ancora: nel λ calcolo, non esiste un costrutto che permetta di creare legami fra nomi e simboli a livello globale!).

Dopo queste doverose premesse, cominciamo a vedere come implementare i costrutti di alto livello a cui siamo interessati usando il λ calcolo. Ovviamente, la prima cosa da fare è trovare una "λ codifica" per i numeri naturali (e le operazioni che si possono compiere su di essi). Questi potranno essere poi utilizzati per codificare i numeri interi (naturali con segno), razionali, reali e così via.

Sarà poi possibile usare λ espressioni per rappresentare valori booleani, le operazioni logiche di base ed il cosiddetto if aritmetico (che permette di valutare un'espressione E_1 oppure un'espressione E_2 dipendentemente dal valore di verità di un'espressione booleano). Per finire, si possono implementare strutture dati più complesse per mostrare come tipi di dato di più alto livello siano rappresentabili tramite λ espressioni.

Ricordando che il λ calcolo è un formalismo funzionale, è abbastanza chiaro che sarà necessario utilizzare una definizione induttiva dei numeri naturali, simile a quella di Peano:

- 0 è un numero naturale
- Dato un numero naturale n, il successivo di n (calcolabile come $succ(n)$) è un numero naturale

L'idea è quindi quella di utilizzare una λ espressione per rappresentare il numero naturale 0 e definire un combinator che applicato alla rappresentazione di un numero naturale n calcola la rappresentazione di $n + 1$. La più famosa fra le codifiche di questo tipo è data dai *numerali di Church*:

- Il numero naturale 0 è rappresentato dalla λ espressione $\lambda f.\lambda x.x$
- La funzione $succ()$ che calcola il successivo di un numero naturale n è codificata dalla λ espressione $\lambda n.\lambda f.\lambda x.f(nfx)$

Questa codifica ha l'interessante proprietà che il numero naturale $n \in N$ è rappresentato dalla funzione f applicata n volte ad x: $n \equiv \lambda f.\lambda x.\overbrace{f(\ldots f}^{n}(x)\ldots)$ (per semplificare la notazione, l'espressione "$\overbrace{f(\ldots f}^{n}(x)\ldots)$" viene talvolta scritta come "$f^n(x)$").

A titolo di esercizio, si può provare a calcolare la rappresentazione del numero naturale 1 come $1 = succ(0)$:

$$1 = succ(0) = (\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.x)$$

$$(\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.x) \to_\beta (\lambda f.\lambda x.f((nf)x))[n \to (\lambda f.\lambda x.x)] = \lambda f.\lambda x.f(((\lambda f.\lambda x.x)f)x) \to_\beta$$

$$\to_\beta \lambda f.\lambda x.f(((\lambda x.x)[f \to f])x) = \lambda f.\lambda x.f((\lambda x.x)x) \to_\beta \lambda f.\lambda x.f((x)[x \to x]) = \lambda f.\lambda x.f(x)$$

Analogamente, si può calcolare la codifica di 2:

$$2 = succ(1) = (\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.f(x))$$

$$(\lambda n.\lambda f.\lambda x.f((nf)x))(\lambda f.\lambda x.f(x)) \to_\beta (\lambda f.\lambda x.f((nf)x))[n \to (\lambda f.\lambda x.f(x))] = \lambda f.\lambda x.f(((\lambda f.\lambda x.f(x))f)x) \to_\beta$$

$$\to_\beta \lambda f.\lambda x.f(((\lambda x.f(x))[f \to f])x) = \lambda f.\lambda x.f((\lambda x.f(x))x) \to_\beta \lambda f.\lambda x.f(f(x))$$

L'esercizio può essere ripetuto per altri numeri naturali.

Senza pretendere di dare dimostrazioni rigorose, proviamo a capire come sia possibile ricavare la codifica di $succ()$ a partire dal suo funzionamento: informalmente parlando, $succ$ deve trasformare $\lambda f.\lambda x.f^n(x)$ in $\lambda f.\lambda x.f(f^n(x))$. Questo può essere fatto:

1. "Rimuovendo" in qualche modo le due astrazioni $\lambda f.\lambda.x.$ dalla codifica di n
2. Applicando f all'espressione così ottenuta
3. Aggiungendo di nuovo le due astrazioni $\lambda f.\lambda.x.$ rimosse al passo 1
4. Astraendo il tutto rispetto al numero n

Il primo passo (rimozione delle astrazioni $\lambda f.\lambda.x.$) si può facilmente compiere applicando la codifica del numero naturale ad f ed a x: infatti, $(\lambda f.\lambda x.f^n(x))f \rightarrow_\beta \lambda x.f^n(x)$ e $((\lambda f.\lambda x.f^n(x))f)x \rightarrow_\beta (\lambda x.f^n(x))x \rightarrow_\beta f^n(x)$. Quindi, se la funzione n rappresenta la codifica di un naturale, allora $(nf)x$ è un'espressione contenente f applicata n volte ad x. Come detto (passo 2), a questa espressione va applicata f ancora una volta, ottenendo $f((nf)x)$; dopo il passo 3 si ottiene quindi $\lambda f.\lambda x.f((nf)x)$ ed astraendo il tutto rispetto ad n (in modo che n sia un argomento del combinator *succ* e non una variabile libera) si ottiene $\lambda n.\lambda f.\lambda x.f((nf)x)$ che è proprio la codifica di *succ* presentata sopra.

Basandosi sulle definizioni dei numerali di Church appena presentate, è possibile definire la codifica delle varie operazioni sui numeri naturali. Per esempio, la somma è codificabile tramite un combinator che applicato a due codifiche di naturali genera la codifica della loro somma. L'espressione di tale combinator è $\lambda m.\lambda n.\lambda f.\lambda x.(mf)((nf)x)$ e può essere ricavata in modo analogo a quanto fatto precedentemente per *succ*:

1. Prima di tutto, si applica n a f e x per rimuovere le astrazioni $\lambda f.\lambda x.$, analogamente a quanto fatto per la codifica di *succ*

2. Poi, si applica m a f per rimuovere l'astrazione λf.

3. A questo punto, applicando il risultato di mf (vale a dire, "m" senza "λf") al risultato di $((nf)x)$ (che è "$f^n x$") si aggiungono m "$f($" alla sinistra di "$f^n x$". Come risultato si ottiene "$f^{n+m}x$"

4. Come fatto per *succ*, si astrae di nuovo rispetto a f e x per aggiungere il $\lambda f.\lambda x.$ rimosso al passo 1

5. Per finire, si astrae rispetto a n e m, in modo da ottenere un combinator

In modo più o meno semplice è possibile definire la codifica anche delle altre operazioni su numeri naturali, anche se qui si omettono i dettagli. La codifica di un operatore "*pred*" (che calcola il predecessore di un numero naturale) non è semplice (ed a tal proposito si raccontano strani aneddoti che coinvolgono Alonso Church - inventore del λ calcolo - un suo dottorando - che ha risolto il problema della codifica di *pred* - ed un barbiere) ma è comunque possibile. Senza entrare nei dettagli, la codifica di tale operatore prevede di trasformare la codifica di n in una coppia contentente la codifica di n e la codifica di $n-1$, per poi prendere il secondo elemento della coppia. Il passaggio dalla codifica di n alla codifica di $(n, n-1)$ è fattibile partendo dalla codifica di $(0,0)$ ed iterando n volte una funzione $\hat{f}$ che trasforma la coppia (n, m) in $(n+1, n)$. Ora, se si ricorda che la codifica di n è un combinator che applica n volte il suo primo argomento al suo secondo argomento, diventa chiaro che $(n, n-1)$ può essere ottenuta applicando n a $\hat{f}$ e poi applicando la funzione risultante alla codifica di $(0,0)$. A questo punto, come detto la codifica di $n-1$ può essere ottenuta applicando al risultato una funzione che ritorna il secondo elemento di una coppia. Al solito, il tutto va astratto rispetto ad n. Alla luce di questo, è quindi importante capire come codificare le coppie usando il *lambda*-calcolo.

La coppia "(a, b)" può essere codificata come $\lambda z.zab$ ed in generale la funzione che genera la codifica della coppia "(a, b)" a partire da "a" e "b" è $\lambda x.\lambda y.\lambda z.zxy$. Data la codifica di una coppia, è possibile ottenerne il primo elemento tramite la funzione "first" = "$\lambda z.z(\lambda x.\lambda y.x)$" ed il secondo elemento tramite la funzione "second" = "$\lambda z.z(\lambda x.\lambda y.y)$".

Come già detto, oltre ai numeri naturali ed alle operazioni aritmetiche il λ calcolo permette di codificare tutto quanto serve per implementare qualsiasi algoritmo. Una cosa fondamentale in questo senso è codificare i valori booleani `true` e `false` e l'operazione di selezione (if aritmetico). Una codifica semplice per `true` può essere "$\lambda t.\lambda f.t$", mentre `false` può essere codificato come "$\lambda t.\lambda f.f$": informalmente parlando, `true` e `false` vengono codificati come λ-espressioni a due argomenti, che ritornano il primo oppure il secondo argomento (a cui convenzionalmente si associa il significato di vero o falso). La funzione di selezione (if aritmetico), invece può essere codificata come "$\lambda c.\lambda a.\lambda b.cab$": è una λ-espressione che riceve 3 argomenti "c", "a" e "b", dove "c" è la codifica di un valore booleano. Se "c" è la codifica di `true`, allora l'espressione viene valutata ad "a", altrimenti viene valutata a "b":

$$(\lambda c.\lambda a.\lambda b.cab)(\lambda t.\lambda f.t) \rightarrow_\beta \lambda a.\lambda b.(\lambda t.\lambda f.t)ab \rightarrow_\beta \lambda a.\lambda b.(\lambda f.a)b \rightarrow_\beta \lambda a.\lambda b.a$$

e

$$(\lambda c.\lambda a.\lambda b.cab)(\lambda t.\lambda f.f) \rightarrow_\beta \lambda a.\lambda b.(\lambda t.\lambda f.f)ab \rightarrow_\beta \lambda a.\lambda b.(\lambda f.f)b \rightarrow_\beta \lambda a.\lambda b.b$$

Basandosi su queste codifiche poi è possibile implementare gli operatori booleani `and` ($\lambda p.\lambda q.pqp$), `or` ($\lambda p.\lambda q.ppq$), e così via[1].

[1]Si lascia al lettore la verifica della correttezza delle codifiche di `and` e `or`.

```c
unsigned int fattoriale(unsigned int n)
{
  unsigned int i res = 1;

  for (i = 2; i <= n; i++) {
    res = res * i;
  }

  return res;
}
```

Figura 2.1: Implementazione iterativa della funzione `fattoriale()`.

E' poi possibile codificare predicati booleani come "is zero" (che riceve come argomento la codifica di un numero naturale e viene valutato a `true` se il numero è 0), "less than", "equal" e simili.

Sebbene le codifiche di valori ed operazioni presentate fin qui rendano possibile implementare funzioni del tutto generiche (manca ancora un meccanismo per implementare / codificare ricorsione o iterazione, ma verrà mostrato a breve), le lambda espressioni che ne derivano rischiano di essere così complesse da risultare intrattabili. Per esempio, la semplice espressione aritmetica "2 + 3" viene codificata come "$2 + 3 \equiv (\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(f(fx)))$"!!! E la codifica della funzione "$f(a) = a + 2$" risulta "$\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx))$". Per semplificare le espressioni, si può ricorrere ad una notazione talvolta nota come "lambda calcolo applicato", in cui si sostituiscono le codifiche dei vari valori e delle operazioni con i simboli matematici a cui siamo generalmente abituati. Quindi, "+" è un sinonimo di "$(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))$", "2" è un sinonimo di "$(\lambda f.\lambda x.f(fx))$" e così via. Diventa allora possibile scrivere "$\lambda a.a + 2$" invece della lambda espressione citata sopra.

A questo punto, la cosa più importante che sembra mancare per ottenere un linguaggio general purpose è un costrutto di ciclo (o meglio, ricorsione, visto che stiamo parlando di programmazione funzionale!). Come già accennato, a causa della mancanza di un ambente globale sembrerebbe impossibile implementare la ricorsione; in realtà, l'ultima sorpresa del λ-calcolo, il concetto di fixed point combinator, ci viene in aiuto.

Come noto, se la versione "imperativa" di un algoritmo contiene un ciclo la sua implementazione secondo il paradigma funzionale è basata su ricorsione: in altre parole, l'implementazione di una funzione richiama la funzione stessa (l'esempio tipico è il fattoriale). Ma il λ calcolo permette di definire solo funzioni anonime (astrazioni λ) ed in assenza di ambiente globale, una funzione non può richiamare se stessa, non avendo nome. In altre parole, una funzione ricorsiva contiene almeno una variabile libera (non è quindi un combinator), che indica il nome della funzione stessa, da richiamare ricorsivamente. Il primo passo per implementare la ricorsione nel λ calcolo è quindi quello di eliminare questa variabile libera (trasformando quindi la funzione ricorsiva in un combinator) passando come argomento il nome della funzione da richiamare ricorsivamente. Se quindi $f = E$ è un'espressione che richiama ricorsivamente f (se stessa), viene trasformata in $f_c = \lambda f.E$, legando la variabile f, che diventa quindi il primo argomento di f_c.

In altre parole, si può dire che f è ottenibile passando f come argomento ad f_c: $f = f_c f$, dove in questo caso "=" significa "$\equiv_\beta$" (β equivalente). Quello che sembrerebbe un semplice trucco sintattico ci permette invece di riformulare il nostro problema come la ricerca di un "punto fisso" di f_c: $f \equiv_\beta f_c f$ può essere vista come un'equazione la cui soluzione f è la funzione ricorsiva che stiamo cercando. L'esistenza dei *fixed point combinator* (combinator che data una funzione f_c ci permettono di calcolarne il punto fisso $f = f_c f$) ci dimostra che la ricorsione è implementabile nel λ-calcolo, anche definendo solo funzioni anonime.

Il più famoso fixed point combinator è $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

A livello di esempio, vediamo come utilizzare il fixed point combinator Y per calcolare la funzione fattoriale. Ricordiamo che una possibile implementazione imperativa del fattoriale è mostrata in Figura 2.1, mentre la tradizionale implementazione ricorsiva è mostrata in Figura 2.2. Riscritta secondo il paradigma funzionale, tale funzione appare come in Figura 2.3. Un primo tentativo (non troppo riuscito, per il vero) di conversione in λ-espressione potrebbe essere

$$\texttt{fattoriale} = \lambda n.\texttt{cond}\ (n = 0)1(n * (\texttt{fattoriale}\ (\text{pred}\ n))$$

```
unsigned int fattoriale(unsigned int n)
{
  if (n == 0) return 1;
  return n * fattoriale(n - 1);
}
```

Figura 2.2: Implementazione ricorsiva della funzione `fattoriale()`.

```
unsigned int fattoriale(unsigned int n)
{
  return (n == 0) ? 1 : n * fattoriale(n - 1);
}
```

Figura 2.3: Implementazione funzionale della funzione `fattoriale()`.

Si noti che questa funzione, che potrebbe apparire strana quando si parla di λ-calcolo puro (in quanto contiene funzioni come "`pred`" ed predicati come $n = 0$ ed espressioni come $n - 1$, che non fanno parte del λ-calcolo puro), è stata scritta usando il λ-calcolo applicato. Come visto, infatti, la selezione "`cond`" (if aritmetico) è sostituibile con la λ-espressione $\lambda c.\lambda a.\lambda b.cab$, il predicato $n = 0$ è sostituibile con la λ-espressione che codifica "is zero" e `pred` è sostituibile con la sua codifica descritta precedentemente.

Come già notato più volte, questa è comunque una strana forma di definizione perché richiede la presenza di un legame per il proprio nome nell'ambiente globale. Questo problema è risolubile definendo la funzione f_c come

$$\lambda f.\lambda n.\texttt{cond}(n = 0)1(n * (f(\texttt{pred}\ n))$$

e trovando la funzione `fattoriale` tale che `fattoriale`$= f_c$`fattoriale` (punto fisso di f_c). Tale funzione f è calcolabile usando il fixed point combinator Y: `fattoriale`$= Y f_c$.

Per finire, va notato come le codifiche di tipi di dato[2] e funzioni di alto livello presentate qui sopra non siano univoche. Per esempio, sempre usando i numerali di Church è possibile definire le operazioni di somma o di predecessore in modo diverso (ma equivalente dal punto di vista delle funzionalità) da quello presentato.

Spingendosi oltre, si può notare come la codifica di Church sia solo una delle possibili codifiche di strutture dati e funzioni di alto livello ed altre codifiche alternative siano possibili. Per esempio, i cosiddetti *numerali di Scott* propongono una codifica dei numeri naturali (e delle operazioni su di essi) alternativa a quella di Church:

- Il numero naturale 0 è rappresentato dalla λ espressione $\lambda f.\lambda x.f$
- La funzione `succ` che calcola il successivo di un numero naturale n è codificata dalla λ espressione $\lambda n.\lambda f.\lambda x.xn$

Sebbene la codifica di Scott sia meno nota (e meno utilizzata!) rispetto a quella di Church, in alcuni casi presenta dei vantaggi (per esempio, permette di semplificare la definizione della funzione predecessore `pred` che è codificabile come $\lambda n.n(0)(\lambda x.x)$).

2.4 Rimozione delle Astrazioni

Come visto, i costrutti principali del lambda calcolo sono l'astrazione (definizione di funzioni) e l'applicazione di funzione. In realtà, è possibile definire un linguaggio di programmazione funzionale minimale anche senza utilizzare il meccanismo di astrazione, a patto di fornire un adeguato insieme di funzioni predefinite.

Quello che si ottiene è un "calcolo dei combinator", (*combinatory calculus* in inglese) cosìchiamato per i combinator predefiniti (l'insieme di funzioni predefinite di cui sopra) su cui si basa.

La sintassi di un'espressione di questo tipo di calcoli può essere definita come:

[2] In realtà, è stata presentata solo la codifica dei numeri naturali... Ma è possibile codificare in λ-calcolo qualsiasi tipo di dati.

```
<expression> ::= <name>                          ; lettera minuscola
               | <Combinator>                    ; funzione predefinita
               | (<expression> <expression>)     ; applicazione
```

dove <name> è il nome di una variabile e <Combinator> è una funzione predefinita (con un comportamento ben definito). Questo è equivalente alla seguente definizione induttiva:

- Un nome / variabile (indicato con una singola lettera minuscola) è un'espressione del calcolo dei combinator
- Un combinator (indicato con una singola lettera maiuscola) è un'espressione del calcolo dei combinator
- Se e_1 ed e_2 sono espressioni del calcolo, allora $(e_1 e_2)$ è un'espressione

Si noti come tutte le variabili che compaiono in un'espressione siano variabili libere (in quanto non esiste il concetto di astrazione).

Il tipo di calcolo dipende chiaramente dai combinator predefiniti ed è chiaro che non tutte le combinazioni di combinator danno origine ad un calcolo turing completo.

Il più importante dei combinatory calculi è probabilmente il "calcolo SK" (talvolta noto come "calcolo SKI"), in cui i combinator predefiniti sono S e K (più opzionalmente il combinator identità I) definiti dalle seguenti proprietà:

$$\begin{aligned} Kxy &= x \\ Sxyz &= xz(yz) \\ Ix &= x \end{aligned}$$

Il combinator I è spesso usato per semplificare le espressioni del calcolo, ma non è strettamente necessario, in quanto può essere ottenuto come $I = SKK$: $(SKK)x = SKKx = Kx(Kx) = x$.

Si lascia al lettore la dimostrazione che le definizioni dei combinator S e K presentate nella Sezione 2.3 ($S = \lambda f.\lambda g.\lambda x.fx(gx)$ e $K = \lambda x.\lambda y.x$) godono delle proprietà descritte qui sopra. E' quindi semplice convertire un'espressione del calcolo SK (o del calcolo SKI) in una λ-espressione. Poiché è anche possibile convertire ogni λ-espressione in un'espressione del calcolo SK, il calcolo SK ha lo stesso potere espressivo del λ calcolo ed è quindi Turing-completo.

La conversione di una generica λ-espressione E in un'espressione del calcolo SK può essere fatta procedendo per casi. In particolare, E può essere:

- Un identificatore x, che nell'espressione x del calcolo SK
- Un'applicazione $E_1 E_2$, che si mappa nell'espressione $E_1 E_2$ del calcolo SK
- Un'astrazione $\lambda x.E'$, che va convertita in un'espressione E'' del calcolo SK procedendo ancora per casi:
  - Se E' è un identificatore, può essere x, nel qual caso $E = \lambda x.x$ si converte in $E'' = I = SKK$, oppure un simbolo $y \neq x$, nel qual caso $E = \lambda x.y$ si converte in $E'' = Ky$
  - Se E' è un'applicazione $E_1'E_2'$, $E = \lambda x.E_1'E_2'$ va convertita E'' tale che $(\lambda x.E'1E_2')v = E''v$. Questo comporta che

    $$(\lambda x.E_1'E_2')v = E''v \Rightarrow E_1'[x \to v]E_2'[x \to v] = E''v \Rightarrow$$
    $$(\lambda x.E_1'v)(\lambda x.E_2'v) = E''v \Rightarrow S(\lambda x.E_1')(\lambda x.E_2')v = E''v \Rightarrow$$
    $$E'' = S(\lambda x.E_1')(\lambda x.E_2')$$
  - Se E' è un'astrazione $\lambda y.E_1'$, $E = \lambda x.\lambda y.E_1'$ va convertita in E'' applicando ricorsivamente questo procedimento ad E_1'.

Applicando questi ragionamenti, si può convertire qualunque λ-espressione in un'espressione basata solo su variabili libere, l'operatore S e l'operatore K.

Un'altra proprietà importante del calcolo SK è che ogni λ-espressione che non contiene variabili libere (vale a dire, un combinator) può essere convertita in un'espressione del calcolo SK che non contiene

variabili. In altre parole, per modellare solo combinator è possibile rimuovere la prima clausola (un nome di variabile è un'espressione del calcolo SK) dalla definizione del calcolo SK.

Il processo che permette di convertire un'espressione "E" in un'espressione $R_x(E)$ che non contiene la variabile libera "x" ma si comporta come $\lambda x.E$ (vale a dire, $R_x(E)E_1 = E[x \to E_1]$) è noto come *bracket abstraction* e può essere utilizzato per convertire qualsiasi λ-espressione in un'espressione del calcolo SK eliminando le astrazioni $\lambda x.$ una ad una. Un algoritmo di bracket abstraction molto semplice (anche se non efficiente) è basato sulle seguenti trasformazioni:

1. $R_x(x) = SKK$, dove "x" è la variabile che si vuole eliminare
2. $R_x(y) = Ky$, dove y è un combinator predefinito (S o K) oppure una variabile diversa dalla variabile x che si vuole eliminare
3. $R_x(E_1E_2) = SR_x(E_1)R_x(E_2)$

Per rendere leggermente più efficiente l'algoritmo, la seconda regola può essere sostituita da $R_x(E) = KE$, dove "E" è un'espressione che non contiene come variabile libera la variabile "x" che si vuole eliminare.

Per convertire una λ-espressione in un'espressione del calcolo SK, si può procedere rimuovendo le astrazioni λ una ad una applicando le tre regole di qui sopra. Si noti la stretta relazione fra questo algoritmo di bracket abstraction e la metodologia di conversione mostrata sopra.

2.5 Lambda Calcolo con Tipi

Come precedentemente detto, nel λ-calcolo "puro" non esiste il concetto di tipo di dato. E' stato mostrato come sia possibile utilizzare espressioni del λ-calcolo non tipizzato per codificare i vari tipi di dato (e le operazioni su di essi), ma le variabili del λ-calcolo rappresentano generiche funzioni, di cui non sono specificati dominio e codominio.

Sebbene la mancanza di tipi di dato non impatti sull'espressività del formalismo (come detto, il λ-calcolo è Turing completo), ne può compromettere la leggibilità e semplicità d'uso, rendendo più semplice commettere errori di programmazione (per questo il λ-calcolo viene considerato una sorta di "Assembly dei linguaggi funzionali"). Per esempio, se si codifica la funzione $f(a) = a + 2$ usando il λ-calcolo si ottiene $\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx))$ (epressione non proprio intuitiva...), che usando il λ-calcolo applicato si semplifica in $\lambda a.a + 2$. Questa codifica ha però perso una caratteristica fondamentale della funzione iniziale: il fatto che la funzione operasse su numeri! E' infatti possibile applicare $\lambda a.a + 2$ (che, ricordiamo, è equivalente a $\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx)))$ a qualsiasi λ-espressione, anche se essa non codifica un numero! Se applichiamo la funzione ad una espressione E che codifica un numero naturale, otterremo come risultato una λ-espressione che codifica un numero naturale, altrimenti otterremo una λ-espressione E' alla quale non sappiamo dare un significato.

Per risolvere problemi di questo genere, si può provare ad associare un tipo ad ogni λ-espressione (o ad ogni argomento). Per esempio, introducendo il vincolo che $\lambda a.(\lambda n.\lambda m.\lambda f.\lambda x.(nf)((mf)x))a(\lambda f.\lambda x.f(fx))$ è una funzione $\mathcal{N} \to \mathcal{N}$ o, meglio, che in tale espressione "a" è la codifica di un numero naturale ($a : \mathcal{N}$).

In questa sezione (che non pretende di essere esaustiva, ma solo di introdurre alcuni concetti che potranno essere poi approfonditi dal lettore) verrà mostrato come sia possibile estendere il formalismo originario per introdurre il concetto di tipo di una funzione, specificandone dominio e codominio. Questo chiaramente può essere fatto in vari modi che risultano in differenti definizioni di λ-calcolo con tipi, le più famose delle quali sono dovute ancora una volta ad Alonso Church e Haskell Curry. Si parlerà quindi di λ-calcolo con tipi "a-la Church" o "a-la Curry".

In modo abbastanza sorprendente, associando un tipo alle funzioni in realtà si finisce per ridurre la potenza espressiva del formalismo, che non risulta più essere Turing completo. Questo perché si può dimostrare che la riduzione di qualsiasi espressione "correttamente tipizzata" (questo concetto verrà introdotto a livello intuitivo nelle prossime pagine) di un λ-calcolo con tipi termina sempre (non è quindi più possibile esprimere ricorsioni infinite). Una semplice intuizione di questo fatto si può avere provando a calcolare il tipo dell'operatore Y, necessario per codificare funzioni ricorsive.

Prima di tutto, per definire un λ-calcolo con tipi è necessario introdurre il concetto di tipo; questo può essere fatto introducendo un insieme $\mathcal{P}$ di tipi base, o *tipi primitivi*, ed una regola per definire nuovi tipi di dato a partire da tipi di dato esistenti (questo è analogo a quanto fatto per definire le espressioni del λ-calcolo, create a partire da un insieme di nomi di base e 2 regole che permettono di creare nuove espressioni a partire da espressioni valide). Poiché stiamo definendo tipi per un λ-calcolo, un nuovo tipo γ si potà definire a partire da due tipi α e β come $\gamma = \alpha \to \beta$ (γ è il tipo delle funzioni che hanno α

come dominio e β come codominio). In altre parole, l'insieme $\mathcal{T}$ dei possibili tipi è generabile tramite la seguente definizione induttiva:

- Un nome di tipo primitivo indica un tipo: $\alpha \in \mathcal{P} \Rightarrow \alpha \in \mathcal{T}$
- Se α ed β sono tipi, allora anche $\alpha \to \beta$ è un tipo: $\alpha, \beta \in \mathcal{T} \Rightarrow \alpha \to \beta \in \mathcal{T}$

Come è facile intuire da questa definizione, il numero di possibili tipi (la cardinalità dell'insieme $\mathcal{T}$ dei tipi) è infinito.

Data una λ-espressione E, il suo tipo è calcolabile in base alle seguenti regole:

- Il tipo di una variabile libera x deve essere noto a priori
- Se E_1 ed E_2 sono espressioni con tipi $\alpha \to \beta$ e α, il tipo di $E_1 E_2$ è β: $E_1 : \alpha \to \beta, E_2 : \alpha \Rightarrow E_1 E_2 : \beta$
- Se E è un'espressione di tipo β, $\lambda x.E$ ha tipo $\alpha \to \beta$: $E : \beta \Rightarrow \lambda x.E : \alpha \to \beta$

spesso per chiarire la terza regola la sintassi dell'astrazione viene modificata per permettere di specificare il tipo dell'argomento. In questi casi si usa $\lambda x : \alpha.E$ (tipizzazione esplicita) invece che $\lambda x.E$ (tipizzazione implicita). Per fare un'analogia con linguaggi di più alto livello, si possono considerare il linguaggio C (in cui nel momento in cui si dichiara una variabile è necessario specificarne il tipo), il linguaggio C++ (in cui la keyword "`auto`" permette di chiedere al compilatore di inferire il tipo di una variabile), standard ML o Haskell (in cui non è necessario specificare il tipo dei vari valori, perché il compilatore è in grado di inferirlo da solo - anche se è comunque permesso di specificare opzionalmente il tipo di un simbolo).

Un'altra cosa importante da notare è che per poter associare un tipo ad un'espressione E è necessario fare delle assunzioni sui tipi delle variabili libere in essa contenute (vedere la prima regola qui sopra). Tali assunzioni (ovviamente necessarie solo per espressioni non chiuse) sono contenute in una sorta di "ambiente dei tipi", o "contesto dei tipi". Riassumendo, il tipo di un'espressione chiusa può essere in qualche modo "calcolato" senza bisogno di informazioni aggiuntive, mentre il tipo di un'espressione aperta dipende dall'ambiente (o contesto) dei tipi.

Informalmente parlando, si può dire che un'espressione E è correttamente tipizzata se è possibile associare ad E un tipo $\alpha \in \mathcal{T}$ che sia consistente con le regole esposte sopra. Per esempio, l'espressione $\lambda x : \texttt{int}.x$ ha è correttamente tipizzata (ed ha tipo $\texttt{int} \to \texttt{int}$), cosìcome l'espressione $I = \lambda x.x$ (o $I = \lambda x : \alpha.x$), che ha tipo $\alpha \to \alpha$. Il combinator $\omega = \lambda x.xx$ non è invece correttamente tipizzato: assumendo che x abbia tipo α ($x : \alpha$), si ha che $\omega : \alpha \to \beta$, dove β è il tipo dell'espressione "xx". Ma perché "xx" sia un'espressione valida, bisogna che x sia una funzione, con dominio α (il tipo dell'argomento). Quindi, $x : \alpha \to \beta$, ma anche $x : \alpha$, da cui si deriva $\alpha = \alpha \to \beta$, che non è un'espressione valida nel sistema di tipi che abbiamo definito (vale a dire, usando le regole di generazione dei tipi esposte sopra non è possibile "costruire" un tipo $\alpha \in \mathcal{T}$ che abbia questa proprietà).

Una volta introdotto il concetti di tipi e di espressioni correttamente tipizzate, si possono seguire due differenti approcci:

- Il primo approccio consiste nel *definire la semantica delle espressioni indipendentemente dal loro tipo* (in pratica, si definiscono regole di β riduzione che non dipendono dai tipi delle espressioni). Il concetto di tipo viene poi usato solo a posteriori per "rigettare" come invalide le espressioni non correttamente tipizzate
- Il secondo approccio consiste nello specificare la semantica solo alle espressioni correttamente tipizzate. In altre parole, se un'espressione non è correttamente tipizzata (vale a dire, non è possibile associarle un tipo), non ha neanche senso provare a ridurla.

Secondo il primo approccio, che da' origine al cosiddetto "λ-calcolo con tipi a-la Curry", l'introduzione dei tipi è utilizzata per "eliminare" dal nostro calcolo le espressioni che "non si comportano come vogliamo" (per esempio, le espressioni la cui riduzione non termina). Ma la riduzione di tali espressioni è comunque definita. In sostanza, i tipi aggiungono semplicemente degli ulteriori vincoli sulle espressioni, che caratterizzano le "espressioni valide".

Nel secondo approccio, invece, che da' origine al cosiddetto "λ-calcolo con tipi a-la Church", il tipo di un'espressione è considerato fondamentale per la sua semantica (non è possibile definire la semantica di un'espressione non correttamente tipizzata). Le regole di riduzione delle espressioni fanno quindi esplicitamente riferimento al loro tipo.

In letteratura a volte si utilizza la tipizzazione implicita nel calcolo a-la Curry e tipizzazione esplicita e quello a-la Church, ma questo non è strettamente necessario.

Indipendentemente dal fatto che si utilizzi tipizzazione implicita o esplicita, è possibile ridurre una espressione del λ-calcolo con tipi utilizzando la tradizionale regola di β riduzione del λ-calcolo senza tipi, dopo aver rimosso le annotazioni di tipo ($: \alpha$ e simili) dalle variabili legate. Se si segue un approccio a-la Church, chiaramente questo può essere fatto solo a patto di aver verificato la corretta tipizzazione dell'espressione.

Alternativamente, una volta che ad ogni espressione è associato un tipo, la regola della β riduzione deve essere aggiornata per tenerne conto: se nel λ-calcolo senza tipi

$$(\lambda x.E)E_1 \rightarrow_\beta E[x \rightarrow E_1]$$

nel lambda calcolo con tipi la riduzione è possibile solo se "x" ed "E_1" hanno lo stesso tipo:

$$E : \alpha \Rightarrow (\lambda x : \alpha.E)E_1 \rightarrow_\beta E[x \rightarrow E_1]$$

.

Capitolo 3

Linguaggi di Programmazione Funzionale for Dummies

3.1 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

[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.

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 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).

3.2 Tipi ed Espressioni in Standard ML

I tipi di dato base forniti da Standard ML sono: `unit`, `bool`, `int`, `real`, `char` e `string`; oltre a fornire questi tipi di base, Standard ML permette anche di combinarli usando delle *tuple*, di definire dei sinonimi per tipi di dato esistenti e di definire nuovi tipi di dato (i cui valori sono generati da apposite funzioni chiamate *costruttori*).

Il tipo `unit` è composto da un unico valore, `()` e viene utilizzato come tipo di ritorno per espressioni che non genererebbero alcun valore (e che sono importanti solo per i propri effetti collaterali). In teoria non dovrebbe esistere in un linguaggio di programmazione puramente funzionale, ma esiste in Standard ML solo per supportare funzioni con effetti collaterali (principalmente input / output).

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 è definito l'operatore `~`, che nega il segno di un numero; quindi, per esempio, `~3` rappresenta il numero -3. Altri operatori definiti sul tipo `int` sono le operazioni aritmetiche di base `*`, `+` e `-`[3]. L'operatore `/` di divisione non è definito sugli interi (mentre esiste la divisione intera `div`).

Il tipo `real` è composto da un insieme di approssimazioni di numeri reali, che 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 `inf` possono essere usati per indicare valori non rappresentabili come numeri reali (per esempio, la radice quadrata di un numero negativo) o valori infiniti (il risultato della divisione di un numero reale per 0). È infine da notare che i valori di tipo `real` sono confrontabili con gli operatori $<$, $>$, $<=$ e $>=$ (che vengono valutati a valori di tipo `bool`), ma non si può usare il testi di uguaglianza su di essi.

Il tipo `char` è composto dall'insieme dei caratteri. Un valore di tale tipo viene rappresentato con il prefisso `#` e fra virgolette; per esempio, `#"a"`.

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

Va inoltre notato che Standard ML non effettua conversioni automatiche di tipo. Quindi, espressioni tipo `5 + 2` (che viene valutata a 7, di tipo `int`) e `5.0 + 2.0` (che viene valutata a 7.0 di tipo `real`) sono corrette, ma l'espressione `5.0 + 2` genera un errore (somma fra un valore di tipo `real` ed un valore di

[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]notare che in Standard ML esiste la differenza fra l'operazione `-` (sottrazione) e l'operatore unario `~` che inverte il segno di un numero.

tipo `int`). Standard ML fornisce però varie operazioni per convertire valori fra i vari tipi; per esempio, `ord` converte un valore di tipo `char` in un valore di tipo `int` (che rappresenta il suo codice ASCII) e `chr` effettua l'operazione inversa.

Per finire, oltre ai "classici" operatori sui vari tipi di variabili Standard ML 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 Standard ML 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 (notare che `<exp1>` e `<exp2>` devono avere lo stesso tipo 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 Standard ML sia spesso considerato l'equivalente a livello di espressione dell'operazione di selezione `if` fornita dai linguaggi imperativi, è importante notare alcune differenze. Per esempio, l'operazione di selezione di un linguaggio imperativo permette 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 Standard ML, 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 Standard ML è equivalente all'if aritmetico "*predicato* ? *espressione1* : *espressione2*" del linguaggio 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`.

3.3 Associare Nomi a Valori

Le espressioni che compongono un programma ML possono usare come operandi direttamente valori o possono usare degli *identificatori* definiti in un *ambiente* per rappresentare dei valori. Un ambiente può essere visto come un insieme di coppie (identificatore, valore) che associano nomi (o identificatori) a valori[4].

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 la keyword `val`:

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

Questo codice (chiamato *dichiarazione* in ML) aggiunge all'ambiente un legame tra l'identificaotore `<name>` ed il valore `<value>` (di tipo `<type>`). Il valore `<value>` può essere anche il risultato della valutazione di un'espressione; in questo caso, la dichiarazione assume la forma

```
val <name> = <expression>;
```

per esempio, `val v = 10 / 2;`.

Notare che una dichiarazione in Standard ML (introdotta dalla keyword `val`) è talvolta vista come una *dichiarazione di variabile*: per esempio, si può dire che `val pi = 3.14` crea una variabile identificata dal nome `pi` e la lega al valore reale 3.14. Ma va comunque notato che in ML le variabili sono semplicemente nomi per dei valori, non sono contenitori di valori modificabili[5]. In altre parole, una variabile ha sempre un valore costante, non modificabile, ed una successiva dichiarazione `val pi = 3.1415` non modifica il valore della variabile `pi` ma crea un nuovo valore 3.1415 di tipo `real` e lo associa al nome `pi`, "mascherando" l'associazione precedente. ML utilizza sempre l'ultimo valore che è stato associato ad un nome. Questo significa che la keyword `val` **modifica l'ambiente, non il valore di variabili**: `val` definisce sempre una nuova variabile (inizializzata col valore specificato) e crea un nuovo legame (fra il nome specificato e la variabile creata) nell'ambiente.

[4]Si noti che qui per ambiente si intende un ambiente globale (vari blocchi di codice avranno poi il loro ambiente locale.

[5]In questo documento non vengono considerate le cosiddette variabili *reference*, che introducono effetti collaterali.

3.4 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 3.8) 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, in Standard ML un valore di tipo funzione è generato dall'operatore `fn` (equivalente a λ nel λ-calcolo). La sintassi è:

```
fn <param> => <expression>;
```

dove `<param>` è il nome del parametro formale (con eventualmente specificato il suo tipo) mentre `<expression>` è un'espressione valida, che può utilizzare i nomi presenti nell'ambiente globale più il nome `<param>`. L'espressione **fn** x:t => exp quando valutata ha quindi come risultato un valore di tipo funzione: in particolare, una funzione che accetta come parametro un valore di tipo `t`. 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`.

Per esempio,

```
fn n => n + 1;
```

è una funzione che incrementa un numero naturale (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,

```
(fn n => n + 1) 5;
```

applica la funzione `fn 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 costrutto `val` di Standard ML. Per esempio, il seguente codice definirà una variabile (non modificabile, ricordate!) `incrementa` avente come valore una funzione da interi ad interi che somma 1 al valore passato come parametro (in altre parole, assocerà il nome `incrementa` a tale funzione):

```
val incrementa = fn n => n + 1;
```

Il tipo di questa variabile è `fn : int -> int` (equivalente alla dizione matematica $f : \mathcal{Z} \rightarrow \mathcal{Z}$). 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

```
(fn x => x+1) ((fn x => x+1) 2);
```

e

```
incrementa (incrementa 2);
```

Per finire, in Standard ML 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):

```
fun <name>(<param>) = <expression>;
```

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

Figura 3.1: Esempio di funzione definita per casi.

```
val f = fn a => case a of
                  0 => 1000.0
                | x => 1.0 / (real x);
```

Figura 3.2: Esempio di binding creato tramite pattern matching.

è (circa) equivalente a

```
val <name> = fn <param> => <expression>;
```

Per esempio, la funzione `incrementa` è definibile come

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

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

Notare infine che la keyword `fun` non introduce nuove funzionalità ma è soltanto *zucchero sintattico* per semplificare la scrittura di definizioni `val ... = fn ...`[6].

3.5 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 con ogni valore del parametro. Questo può essere facilmente fatto utilizzando l'operatore `case`:

```
fn 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 mostrato in Figura 3.1. 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 Standard ML (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", Standard ML utilizza un meccanismo di *pattern matching*. Se un

[6] In realtà le cose sono leggermente più complicate di questo, in caso di funzioni ricorsive o di funzioni a più parametri... Ma per il momento consideriamo `fun` come un semplice sinonimo di `val ... = fn ...`.

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

Figura 3.3: Esempio di funzione definita per casi, sintassi alternativa.

pattern per esempio contiene dei nomi, quando Standard ML 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 di Figura 3.2 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 per definire funzioni in più variabili:

```
val somma = fn (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

```
fn x = case x of
    <pattern_1> => <expression_1>
  | <pattern_2> => <expression_2>
  ...
  ...
  | <pattern_n> => <expression_n>;
```

sia equivalente a

```
fn <pattern_1> => <expression_1>
 | <pattern_2> => <expression_2>
 ...
 ...
 | <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, Standard ML 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.*

Usando questa sintassi, la definizione della funzione `giorno` di Figura 3.1 può essere riformulata come in Figura 3.3. 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

```
val 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

```
val (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>:<type>, di tipo <type>, che matcha un qualsiasi valore di tipo <type>, dopo aver creato un binding fra il nome <var> ed il valore;
- un *tuple pattern* (<pattern_1>, <pattern_2>, ..., <pattern_n>), di tipo <type_1> * <type_2> * ... * <type_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 () è un esempio di tuple pattern (tupla nulla).

3.6 Ricorsione

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 quidni importante capire come implementare la ricorsione usando ML.

In teoria, in una funzione ricorsiva non è una funzione simile alle altre e non dovrebbe avere alcun bisogno di essere definita in modo particolare: semplicemnte, l'espressione che implementa il corpo della funzione richiama la funzione stessa. In realtà esistono però delle complicazioni dovute alla visibiltà degli identificatori: quando si usa val per associare un nome ad una funzione, il nome diventa visibile solo dopo che l'intera espressione val ... è stata completamente valutata. Quindi, non è ancora visibile quando dovrebbe essere usato dall'espressione che costituisce il corpo della funzione. Per questo motivo, un'espressione come

```
val fact = fn n => if n = 0 then 1 else n * fact (n - 1);
```

è destinata a fallire: il nome fact non è ancora visibile durante la definizione della funzione fact.

La soluzione a questo problema è usre val rec invece di val:

```
val rec fact = fn n => if n = 0 then 1 else n * fact (n - 1);
```

val rec aggiunge il legame fra il nome (fact, in questo caso) e la funzione che si sta definendo *prima* che l'espressione sia stata completamente valutata. Quindi, rende possibile chiamare ricorsivamente il nome dalla definizione del corpo della funzione.

Le espressioni fun rendono invece possibile definire funzioni ricorsive senza bisogno di ricorrere alla keyword rec o ad altri accorgimenti:

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

Quindi, quando nella Sezione 3.4 è stato scritto che **fun** <name>(<param>) = <expression> è equivalente a **val** <name> = **fn** <param> => <expression>; è stata commessa una lieve imprecisione: **fun** <name>(<param
è in realtà equivalente a **val rec** <name> = **fn** <param> => <expression>;.

Per capire meglio il significato di val rec e la differenza fra val e val rec, si considerino le differenze fra

```
val f = fn n => n + 1;
val f = fn n => if n = 0 then 1 else n * f(n - 1);
```

e

```
val f = fn n => n + 1;
val rec f = fn n => if n = 0 then 1 else n * f(n - 1);
```

In particolare, qual'è il valore di f 4 nei due casi?

```
val fact = fn n =>
        let
            val rec fact_tr = fn n => fn res =>
                if n = 0 then
                    res
                else
                    fact_tr (n - 1) (n * res)
        in
            fact_tr n 1
        end;
```

Figura 3.4: Implementazione della funzione fattoriale con ricorsione in coda in standard ML, nascondendo la funzione `fact_tr` tramite costrutto `let`.

3.7 Controllare l'Ambiente

Come molti dei moderni linguaggi, ML 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 la keyword `fn` (o la "sintassi abbellita" `fun`) 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 "**val** v = 1" crea un legame fra il simbolo "`v`" ed il valore "`1`" nell'ambiente globale. Una definizione "**val** f = **fn** 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. come nel seguente esempio:

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 possono essere risolti in tale momento. Per esempio, una dichiarazione**val** f = **fn** x => x + y; risulterà in un errore se quando tale dichiarazione viene processata il simbolo `y` non è legato ad alcun valore.

Standard ML 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** <dichiarazioni> **in** <espressione> **end**;) ed uno per modificare l'ambiente all'interno di una dichiarazione (**local** <dichiarazioni> **in** <dichiarazione> **end**;).

In altre parole, "**let** <dichiarazioni> **in** <espressione> **end**;" è un'espressione valutata al valore di `<espressione>` dopo che l'ambiente è stato modificato aggiungendo i binding definiti in `<dichiarazioni>`. 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 **val rec** fact = **fn** 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: **val rec** fact_tr = **fn** n => **fn** res => **if** n = 0 **then** res **else** fact1 (n − 1) (n * res);. Questa funzione riceve quindi due argomenti, a differenza della funzione `fact` originale. E' quindi necessario un wrapper che invochi `fact_tr` con i giusti parametri: **val** fact = **fn** n => fact1 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 3.4.

Il costrutto "**local** <dichiarazioni> **in** <dichiarazione> **end**;" permette invece di utilizzare i binding definiti da `<dichiarazioni>` durante la definizione compresa fra `in` e `end`, ripristinando poi l'ambiente originario.

L'utilità di `local` può essere capita meglio considerando il seguente esempio: si supponga di voler implementare in ML una funzione $f : \mathcal{N} \to \mathcal{N}$, anche se ML 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, richiaman-

```
local
    val rec fact_tr = fn n => fn res =>
        if n = 0 then
            res
        else
            fact_tr (n - 1) (n * res)
in
    val fact = fn n => fact_tr n 1
end;
```

Figura 3.5: Implementazione della funzione fattoriale con ricorsione in coda in standard ML, nascondendo la funzione `fact_tr` tramite costrutto `local`.

dola da una funzione `f` che controlla se il valore del parametro è positivo o no, ritornando -1 in caso di argomento negativo:

```
local
    val integer_f = fn n => ...
  in
    val f = fn n => if n < 0 then ~1 else integer_f n
  end;
```

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

Analogamente, `local` può essere usato per "nascondere" la funzione `fact_tr` a due argomenti nella definizione tail recursive del fattoriale (vedere esempio precedente in Figura 3.4), come mostrato in Figura 3.5.

Confrontando i due esempi di Figura 3.4 e 3.5, è facile capire come esista una stretta relazione fra il costrutto `let` ed il costrutto `local` e come possa essere sempre possibile usare `let` al posto di `local` (spostando la clausola `val` dal blocco `in`...`end` all'esterno del costrutto).

3.8 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 di una funzione $f : \mathcal{R} \to \mathcal{R}$. Cominciando definendo una funzion `derivata1` che accetta come parametro la funzione f di cui calcolare la derivata ed un numero $x \in \mathcal{R}$. La funzione `derivata1` ritorna il valore della funzione $f'()$ derivata di $f()$ (in realtá, dell rapporto incrementale sinistro di $f()$...) calcolata nel punto x. Tale funzione, che ha un parametro di tipo funzione ed uno di tipo `real` e ritorna un valore di tipo `real`, può essere implementata (o meglio, approssimata) in questo modo:

```
val derivata1 = fn (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 $\Delta = 0.001$).

È da notare come ML sia in grado di inferire il tipo dei parametri `x` (che risulta essere `real` a causa dell'espressione `x - 0.001`) ed `f` (che risulta essere `real -> real` poiché `f` viene applicata ad `x` ed il suo valore di ritorno viene diviso per `0.001`). Il tipo di `derivata1` sarà quindi `((real -> real) * real) -> real`.

L'esempio presentato non è comunque sorprendente, perché qualcosa di simile alla funzione `derivata1` presentata qui sopra si può facilmente implementare anche usando un linguaggio prevalentemente impe-

rativo (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 `derivata2` 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 `derivata2` ha un unico parametro, di tipo `real -> real` e ritorna un valore di tipo `real -> real`. Il tipo di questa funzione sarà quindi `(real -> real) -> (real -> real)` Una sua possibile implementazione in ML è la seguente:

```
val derivata2 = fn f => (fn x => (f(x) - f(x - 0.001)) / 0.001);
```

Cerchiamo di capire meglio questa definizione di `derivata`: **val** derivata2 = **fn** 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 **fn** x => (f(x) – f(x – 0.001)) / 0.001 (le parentesi sono state aggiunte nell'esempio di qui sopra per aumentarne la leggibilità), che indica una funzione della variabile `x` calcolata come $\frac{f(x)-f(x-0.001)}{0.001}$. Quindi, il valore di ritorno della funzione `derivata2` è una funzione (del parametro `x`, che ML può identificare come `real` a causa dell'espressione $x - 0.001$). Tale funzione è calcolata in base alla funzione `f`, che è parametro di `derivata`. Valutando la definizione, ML può inferire il tipo di `f` (funzione da `real` a `real`: `real -> real`).

Come ulteriore considerazione, si può notare come la funzione `derivata2` possa essere vista come un'applicazione parziale della funzione `derivata1`. 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 diventare un numero reale quando applicato ad un ulteriore parametro `x` (di tipo reale)... Questo "oggetto" è quindi una funzione $f' : \mathcal{R} \to \mathcal{R}$.

Il procedimento usato per passare dalla prima versione di `derivata` alla seconda è in realtà del tutto generico ed è noto in letteratura col nome di *currying*. 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

```
val somma => fn (x, y) => x + y;
```

può essere scritta tramite currying come

```
val sommac = fn x => (fn y => x + y);
```

(ancora una volta le parentesi sono state aggiunte nel tentativo di rendere la cosa più leggibile).

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 ML queste funzioni sono definite come

```
val f  = fn (x, y)       => x * x + y * y;
val fc = fn x => (fn 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 ML, che fornisce una sintassi semplificata per definire funzioni "curryficate" tramite l'espressione `fun`:

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

è equivalente (ricordare che `fun` è sostanzialmente zucchero sintattico) a

```
var rec f = fn a => fn b = exp;
```

Questa è un'altra piccola correzione rispetto alla definizione leggermente imprecisa di `fun` data nella Sezione 3.4 e corretta (sostituendo `val` con `val rec`) nella Sezione 3.6.

Usando la sintassi semplificata fornita da `fun`, le definizioni di `f` ed `fc` (esempio precedente) diventano quindi

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

mentre la definizione di `derivata2` diventa

```
fun derivata2 f x = (f(x) - f(x - 0.001)) / 0.001;
```

3.9 Ancora sui Tipi di Dato

Fino ad ora abbiamo lavorato coi tipi di dato predefiniti di ML: `bool`, `int`, `real`, `char`, `string` e `unit` (anche se questo ultimo tipo non è stato molto utilizzato... In pratica, è utile prevalentemente per modellare funzioni "strane" che non accettano argomenti o che non ritornano alcun valore).

Tutti i tipi riconosciuti da Standard ML (siano essi "tipi semplici" come i tipi predefiniti 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 prende come unico argomento una tupla composta dai vari argomenti:

```
val sommaquadrati = fn (a,b) => a * a + b * b;
```

Questa funzione ha tipo "int * int -> int"; in altre parole, il suo parametro è di tipo "int * int", che rappresenta il prodotto cartesiano dell'insieme di definizione di `int` per se stesso (matematicamente parlando, $\mathcal{N} \times \mathcal{N}$).

Più formalmente parlando, una tupla è un insieme ordinato di più valori, ognuno dei quali avente un tipo riconosciuto da ML. L'insieme di definizione di una tupla è dato dal prodotto cartesiano degli insieme di definizione dei valori che la compongono. Notare come il tipo `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 `int`; quindi, l'espressione "(6)" viene valutata come "6" e l'espressione "(6) = 6" viene valutata come `true`.

Notare che il meccanismo del pattern matching applicato alle tuple permette di creare binding fra più nomi e simboli contemporaneamente: se "**val** coppia=("pi_greco", 3.14)" crea un legame fra il simbolo "`coppia`" e la coppia "("pi_greco", 3.14)" (avente tipo "`string * real`", "**val** (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 tipo real). Questo stesso meccanismo è stato usato finora per passare parametri multipli ad una funzione usando una tupla come unico argomento.

Standard ML permette di *definire dei sinonimi* per tipi di dati esistenti, usando il costrutto `type`: per esempio, se in un programma usiamo valori interi per rappresentare il tempo, si potrebbe usare l'identificatore `time_t` per valori riferiti ai tempi, usando una definizione del tipo "**type** time_t = int". Questa definizione permette di associare simboli a valori di tipo `time_t`, con definizioni tipo "**val** a:time_t = 5". I valori di tipo `time_t` rimangono però di fatto dei valori interi ed è quindi lecito sommare `a` a valori di tipo intero. Questo utilizzo di `type` aumenta l'espressività e rende più semplice capire il codice: nell'esempio citato sopra, è immediatamente evidente che `a` e `b` rappresentano dei tempi e non dei generici valori interi. Notare però che `type` non introduce nuovi tipi: tornando all'esempio precedente, un'espressione "`a + c`" con `c` associato ad un valore intero (quindi questa espressione somma un valore `time_t` ad un valore `int`) è valida. In altre parole, `time_t` è semplicemente un sinonimo per `int`, non un nuovo tipo.

Questo meccanismo può risultare particolarmente utile per associare nomi simbolici a tipi ottenuti come tuple:

```
type coppiadiinteri = int * int;
type nomecognome = string * string;
type coppiadireali = real * real;
...
```

Quindi per esempio una definizione "**type** coppiadireali = real * real" associa il nome `copiadireali` al tipo `real * real` e permette di definire in modo semplice e chiaro funzioni che ricevono una coppia di numeri floating point come argomento: "**val** fr = **fn** (x, y): coppiadireali => x * y" (una definizione come "**val** fr = **fn** (x, y) = > x * y" avrebbe definito una funzione da coppie di interi ad interi).

Come ulteriore esempio, si consideri una funzione `c2t` che riceve come parametri una coppia di interi `(i1, i2)` ed un intero `i3` generando come risultato la tripletta di interi `(i1, i2, i3)`. Un primo tentativo di definizione di `c2t` può essere:

```
type currency = string;
type money = real * currency;

fun convert (amount, to) =
    let val toeur = fn
        (x, "eur") => x
      | (x, "usd") => x / 1.05
      | (x, "ounce gold") => x * 1113.0
    in
        ( case to of
            "eur" => toeur amount
          | "usd" => toeur amount * 1.05
          | "ounce gold" => toeur amount / 1113.0
        , to)
    end;
```

Figura 3.6: Conversione di denaro fra valute, usando stringhe per rappresentare le varie valute. Si noti che i pattern non sono esaustivi (non coprono tutti i possibili valori).

```
val c2t = fn (x, y) => fn z => (x,y,z);
```

o analogamente

```
fun c2t (x, y) z = (x, y, z);
```

ma questa funzione risulta avere tipo "'a * 'b -> 'c -> 'a * 'b * 'c" che non è proprio quel che ci attendevamo (int * int -> int -> int * int * int). Qui `'a`, `'b` e `'c` rappresentano tipi generici (tutto diventerà più chiaro parlando di polimorfismo). In altre parole, la funzione `c2t` come definita sopra può essere invocata come c2t ("ciao", "mondo") #"c" (ritornando come risultato "("ciao", "mondo", #"c")", di tipo "`string * string * char`"). Una definizione tipo

```
val c2t = fn (x:int, y:int) => fn z:int => (x, y, z);
```

chiaramente risolve il problema. La definizione può essere resa più elegante introducendo un nome per il tipo "`int * int`":

```
type coppiainteri = int * int;
val c2t = fn (x, y):coppiainteri => fn z:int => (x, y, z);
```

Ancora, è importante notare che, come precedentemente detto, `type` definisce un sinonimo per un tipo esistente, ma non definisce un nuovo tipo. Questo può comportare problemi in alcune situazioni, per esempio quando si vuole che una variabile possa assumere solo un numero finito di valori (per semplificare il pattern matching). Come esempio di questa situazione, si consideri il problema di sviluppare una funzione "`convert`" che riceve come argomento un quantitativo di denaro in una valuta nota e lo converte in una diversa valuta. Gli argomenti in ingresso sono il quantitativo di denaro da convertire (associato alla sua valuta) e la valuta in cui convertire. Usando delle stringhe (con valori, per esempio, `"eur"`, `"usd"` e `"ounce gold` per rappresentare le valute, un quantitativo di denaro in una determinata valuta può essere rappresentato con una coppia "`(real, string)`"; ovviamente, si possono usare dei sinonimi per "`string`" e per le coppie "`(real, string)`" in modo da rendere il codice più chiaro. Una possibile implementazione di `convert` usando questa tecnica è mostrata in Figura 3.6. Prima di tutto, è importante notare come "`type`" sia usato per definire "`currency`" e "`money`" come sinonimi per "`string`" e "`real * currency`". A questo punto, la funzione "`convert`" può ricevere come argomento una coppia il cui primo elemento è di tipo "`money`" (e rappresenta il quantitativo di denaro da convertire) ed il secondo elemento è di tipo "`currency`" (e rappresenta la valuta in cui vogliamo convertire il denaro). La funzione potrà essere quindi invocata come "convert ((1.0, "eur"), "usd")".

La funzione "`convert`" lavora utilizzando una "funzione privata" "`toeur`" (presente solo nell'ambiente locale di "`convert`", grazie al costrutto "`let ... in ... end`") che converte un quantitativo di denaro (di tipo "`money`") in euro, ritornando un numero reale. Tale funzione utilizza il pattern matching sulla coppia "`amount`" (ricordare: "`amount`" è di tipo "`real * currency`" quindi è una coppia il cui primo elemento è un numero floating point ed il secondo elemento è una stringa). Se la stringa che rappresenta la valuta in cui è espresso "`amount`" è `"eur"`, "`toeur`" può ritornare il primo elemento della coppia (il

```
type currency = string;
type money = real * currency;

fun convert (amount, to) =
    let val toeur = fn
        (x, "eur")         => x
      | (x, "usd")         => x / 1.05
      | (x, "ounce_gold") => x * 1113.0
      | (_, _)             => ~1
    in
        ( case to of
            "eur"          => toeur amount
          | "usd"          => toeur amount * 1.05
          | "ounce_gold"  => toeur amount / 1113.0
          | _              => ~1
        , to)
    end;
```

Figura 3.7: Conversione di denaro fra valute, usando stringhe per rappresentare le varie valute. Si noti che i pattern ora coprono tutti i possibili valori grazie all'utilizzo del wildcard pattern.

denaro è già in euro), se è `"usd"` il primo elemento della coppia va moltiplicato per 1.05 e così via. Si noti che i pattern usati sono pattern coppia composti da un variable pattern (`x`, che va a matchare il numero reale) ed un pattern costante (`"eur"`, `"usd"`, etc che va a matchare la stringa che esprime in che valuta è il denaro). Dopo aver convertito il denaro in euro (invocando "`toeur`"), "`convert`" converte questo quantitativo nella valuta desiderata (espressa da "`to`"). Ancora, la conversione è fatta usando pattern matching.

Il codice presentato in Figura 3.6 sembra funzionare correttamente, ma ha un grosso problema: i pattern che vengono matchati con "`to`" e con il secondo elemento di "`amount`" non coprono tutti i possibili valori di tipo stringa. Questo può causare (oltre a dei warning quando il codice viene compilato od interpretato) degli errori a tempo di esecuzione, se vengono specificate valute non esistenti o non supportate dalla funzione (per esempio, "convert ((1.0, "eur"), "usb")" - banale errore di battitura in cui si è digitato "`usb`" invece di "`usd`"). Questo problema può essere alleviato usando il wildcard pattern "_" per terminare le due espressioni di pattern matching, in modo da rendere i pattern esaustivi (ricordare che "_" matcha qualsiasi valore e può essere utilizzato per implementare qualcosa di simile alle clausole "`default:`" degli switch C o C++). Il problema è decidere quale valore ritornare quando viene specificata una valuta inesistente o non supportata): per esempio, in Figura 3.7 si è deciso di ritornare -1 in caso di errore. Ora, invece di generare un errore a tempo di esecuzione, "`convert`" ritorna valori apparentemente senza senso in caso di input sbagliati.

Anche la soluzione di Figura 3.7 non è ottimale, perché in caso di input non validi invece di ottenere un errore a run-time sarebbe preferibile avere un errore di compilazione (o un errore dell'interprete basato su un'analisi statica del codice). Ma questo sarebbe possibile solo se il tipo di "`currency`" avesse solo pochi valori validi. In pratica, sarebbe necessario definire un nuovo tipo, che ha come possibili valori solo "`eur`", "`usd`" e "`ounce_gold`", in modo simile ad un tipo enumerato. Per venire incontro ad esigenze di questo genere, ML permette di definire nuovi tipi tramite la keyword `datatype`; in particolare, `datatype` permette di far riconoscere all'interprete valori non conosciuti per default. Il modo più semplice per usare `datatype` è la definizione dell'equivalente di un tipo enumerato (proprio quel che ci serve in questo caso!); per esempio

```
datatype currency = eur | usd | ounce_gold;
```

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 "**val** c = eur") si ottiene un errore. Ma dopo la definizione, "**val** 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`". I nomi separati da "`|`" rappresentano valori costanti che stiamo definendo. Tecnicamente, essi sono dei *costruttori di valore*, vale

```
datatype currency = eur | usd | ounce_gold;
datatype money = Eur of real | Usd of real | Ounce_gold of real;

fun convert (amount, to) =
    let val toeur = fn
          Eur x        => x
        | Usd x        => x / 1.05
        | Ounce_gold x => x * 1113.0
    in
        ( case to of
            eur        => toeur amount
          | usd        => toeur amount * 1.05
          | ounce_gold => toeur amount / 1113.0
        , to)
    end;
```

Figura 3.8: 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.

```
datatype currency = eur | usd | ounce_gold;
datatype money = Eur of real | Usd of real | Ounce_gold of real;

fun convert (amount, to) =
    let val toeur = fn
          Eur x        => x
        | Usd x        => x / 1.05
        | Ounce_gold x => x * 1113.0
    in
        case to of
            eur        => Eur        (toeur amount)
          | usd        => Usd        (toeur amount * 1.05)
          | ounce_gold => Ounce_gold (toeur amount / 1113.0)
    end;
```

Figura 3.9: Conversione di denaro fra valute, versione finale.

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 (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 il tipo `money` non come un sinonimo di una coppia, ma come un nuovo tipo di dato:

```
datatype money = Eur of real | Usd of real | Ounce_gold of real;
```

(notare che qui si sono usati "`Eur`, "`Usd`" ed "`Ounce_gold`" come nomi per i costruttori, perché i nomi "`eur`, "`usd`" ed "`ounce_gold`" sono già usati come costruttori per un altro tipo).

Come tutte le funzioni di Standard ML, 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, "`Eur`" non rappresenta un valore (costante) del tipo "`money`" (come accadeva invece con "`eur`" per il tipo "`currency`"), ma è una funzione da numeri floating point a valori di tipo `money` (in Standard ML, "real −> money"). I valori di tipo `money` sono quindi "`Eur 0.5`" o simili.

La Figura 3.8 mosta come la funzione "`convert`" 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 la conversione della funzione ai 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,

```
val sommanumeri = fn (intero a, intero b) => intero (a + b)
                   | (intero a, reale  b) => reale  ((real a) + b)
                   | (reale  a, intero b) => reale  (a + (real b))
                   | (reale  a, reale  b) => reale  (a + b);
val sottrainumeri = fn (intero a, intero b) => intero (a - b)
                     | (intero a, reale  b) => reale  ((real a) - b)
                     | (reale  a, intero b) => reale  (a - (real b))
                     | (reale  a, reale  b) => reale  (a - b);
...
```

Figura 3.10: Esempio di operazioni definite sul tipo di dato `numero`.

currency) invece che un valore di tipo "`money`". L'implementazione di Figura 3.9 risolve questo ultimo problema.

Quando si usa "`datatype`" 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 (torando all'esempio precedente, definire i tipi "`currency`" e "`money`" senza definire la funzione "`convert`" non è molto utile...).

Come ulteriore esempio, si consideri un tipo `numero` che rappresenta numeri reali o interi. Tale tipo potrebbe essere definito come

```
datatype numero = intero of int | reale of real;
```

Ma se non si definiscono delle funzioni che operano su tale tipo (implementando le operazioni aritmetiche) il tipo "`numero`" non è utilizzabile. E' quindi importante definire anche funzioni come mostrato per esempio in Figura 3.10. Si lascia al lettore l'esercizio di definire "`moltiplicanumeri`", "`dividinumeri`" e via dicendo.

Per finire, è interessante notare come un tipo di dato definito tramite "`datatype`" possa essere definito anche basandosi sul tipo stesso: fino ad ora, abbiamo visto che un costruttore può avere 0 argomenti (costruttore costante) o 1 argomento, usando tipi di dato più semplici (`int`, `real`, coppie...) per tale argomento. Ma l'argomento di un costruttore di un tipo "`T`" può anche avere un argomento di tipo "`T`" o di un tipo derivato da "`T`". In questo caso, si parla di *tipo di dato ricorsivo*. Per esempio, una lista di interi è definibile in modo ricorsivo come

```
datatype lista = foglia of int | nodo of (int * lista);
```

usando un costruttore "`nodo`" che ha un argomento di tipo "`int * lista`" (quindi, il tipo "`lista`" è definito in termini di se stesso; per questo si chiama ricorsivo). Basandosi su questa definizione, la funzione "`Len`" che calcola la lunghezza di una lista può essere definita come

```
val rec Len = fn foglia _      => 1
               | nodo   (_, l) => 1 + Len l;
```

mentre la funzione "`Concatena`" può essere definita come

```
val Concatena = fn (foglia v, l)        => nodo (v, l)
                 | (nodo (v, l1), l2) => nodo (v, Concatena (l1, l2));
```

Notare che la precedente definizione di lista di interi non permette di rappresentare le liste vuote ed una rappresentazione tradizionalmente più usata (perché permette di rappresentare liste vuote) è la seguente:

```
datatype lista = vuota | cons of (int * lista);
```

Si rimanda alle dispense dedicate ai tipi di dato ricorsivi per maggiori dettagli al proposito delle liste in Standard ML.

Per finire, Standard ML implementa il concetto di *type variable* (una variabile i cui valori sono i tipi di dato), che permette di supportare il concetto di polimorfismo parametrico implicito (definendo "classi di funzioni" o "classi di tipi di dato" che sono parametrizzate rispetto a uno o più tipi, indicati come α, β, etc...). Per esempio, la funzione

```
fn x => x
```

```
struct
  val pi = 3.14152;
  val areacerchio = fn r => r * r * pi
end
```

Figura 3.11: Esempio di struttura anonima.

ha tipo

```
'a -> 'a
```

che significa $\alpha \to \alpha$ (i programmi ML sono scritti usando caratteri ASCII, quindi le lettere greche come α non si possono usare... Si usano quindi le lettere dell'alfabeto latino con un apice davanti). Vale a dire, questa funzione (che i lettori potranno facilmente riconoscere come l'identità) mappa un valore di un generico tipo α in un altro valore **dello stesso tipo**.

I type variable possono essere anche usati per definire tipi di dati generici (o polimorfici), che in realtà non sono dei veri tipi di dati, ma delle classi di tipi di dati (parametrizzate rispetto ad una o più variabili α, β, etc...). Per esempio, la definizione

```
datatype 'a oerrore = errore | valore of 'a
```

definisce una classe di tipi di dato "α oerrore" i cui valori sono "`errore`" o valori di un generico tipo α. Potremmo quindi avere un tipo "`int oerrore`" che ha come valori "`errore`" o numeri interi (in realtà, valori generati dai numeri interi usando il costruttore "`valore`"). Ma il sistema conoscerà anche una "regola" per generare tipi "`real oerrore`", "`string oerrore`" e così via (per ogni tipo conosciuto da ML). In generale, `datatype` permette di specificare un type variable come argomento (che precede il nome del tipo) e questa variabile può essere usata per descrivere il tipo di uno (o più) argomenti di uno (o più) dei costruttori. Se sono necessari più type variable, si possono racchiudere in una tupla, come nel seguente esempio:

```
datatype ('a,'b) tipostrano = nessuno | uno of 'a | due of 'a * 'b
```

3.10 Cenni di Programmazione Modulare

Tutto quanto descritto fino ad ora si applica alla cosiddetta *programmazione in piccolo*, in cui un programma è generalmente composto da una sola unità di compilazione (compilation unit) e la sua bassa complessità non richiede alcuna modularizzazione del codice.

In altre parole, un programma in Standard ML appare come una sequenza di definizioni ed applicazioni di funzioni che manipolano tutte lo stesso ambiente e questo può causare una serie di problemi per quanto riguarda la riusabilità del codice. Per esempio, si consideri il seguente codice Standard ML:

```
val pi = 3.14152;
val areacerchio = fn r => r * r * pi;
```

Se si volesse riusare questo codice in un altro programma, bisognerebbe assicurarsi che il nuovo programma non usi già il nome "`pi`" per altri scopi (e non contenga quindi differenti definizioni di "`pi`"). Chiaramente, in questo specifico caso il problema può essere risolto "nascondendo" la definizione di "`pi`" tramite un blocco "`let...in...end`", ma in generale per poter riutilizzare una porzione di codice è necessario isolarla in una qualche "entità" (modulo, package, unità di compilazione o che altro) che permetta di controllare in qualche modo la visibilità delle varie definizioni. Molti dei linguaggi di programmazione moderni supportano questa *programmazione in grande* tramite costrutti linguistici di vario genere che permettono di dividere il codice in unità di compilazione (che possono avere nomi diversi in diversi linguaggi) distinte, di specificare chiaramente l'interfaccia software di ogni unità di compilazione e di separare tale interfaccia software dalla reale implementazione (che può essere in qualche modo "nascosta" all'utilizzatore finale del codice).

Il meccanismo fornito da Standard ML per supportare la programmazione in grande e la modularizzazione del codice è quello delle *structure* e *signature*. Informalmente, si può dire che una struttura (structure) SML permette di "raggruppare" una serie di definizioni in un ambiente (environment), in modo da evitare problemi con "collisioni di identificatori" (stesso identificatore usato con scopi diversi in parti diverse del codice) analoghi a quello discusso sopra. In questo senso, una struttura SML è simile

```
structure cerchio = struct
  val pi = 3.14152;
  val areacerchio = fn r => r * r * pi
end
```

Figura 3.12: Esempio di struttura con nome.

ad un namespace C++. Il costrutto `signature` può essere invece utilizzato per specificare un'interfaccia software. Creando un legame fra structure e signature è quindi possibile specificare quali dei legami presenti nella struttura verranno esportati e come verranno esportati. In altre parole, una struttura contiene l'implementazione di un modulo software, mentre un signature può essere usato per specificarne l'interfaccia software (qualcuno sostiene che il signature specifichi il tipo della struttura).

Più formalmente, una struttura SML (da non essere confusa con le strutture o record presenti alcuni linguaggi imperativi come C o C++) definisce un ambiente che contiene i legami (binding) definiti fra la keyword `struct` e la keyword `end`. Un esempio di stuttura può essere quindi quello mostrato in Figura 3.11, ma poiché una struttura "anonima" come quella definita nella figura è generalmente inutile (non c'è modo di accedere ai binding contenuti nel suo ambiente) si mettono assieme definizione e dichiarazione, associando un identificatore alla struttura tramite la keyword `structure`, come mostrato in Figura 3.12.

Le definizioni contenute nella struttura non hanno effetto sull'ambiente globale, ma modificano solo l'ambiente della struttura, che può essere acceduto tramite la sintassi "`<nome struttura>.<identificatore>`" (dot notation). Il nome "`areacerchio`" non è quindi riconosciuto nell'ambiente globale:

```
> areacerchio 1.0;
poly: : error: Value or constructor (areacerchio) has not been declared
Found near areacerchio 1.0
Static Errors
```

ma è invece visibile come "`cerchio.areacerchio`":

```
> cerchio.areacerchio 1.0;
val it = 3.14152: real
```

L'utilizzo di una struttura permette quindi di riutilizzare senza problemi il codice in programmi che già utilizzano il nome `pi` per altri scopi. Purtroppo, però, ci sono situazioni in cui questo non è abbastanza. Si consideri per esempio l'implementazione di un tipo di dato astratto "`set`" che rappresenti un insieme (per la precisione, il tipo "`'a set`" rappresenta insiemi di elementi di tipo "`'a`"). Una possibile implementazione, basata sull'utilizzo di "`'a list`" per implementare un "`'a set`", è mostrata in Figura 3.13. Si noti come il tipo di "`emptyset`" sia "`'a list`", il tipo di "`isin`" sia "`fn: ''a -> ''a list -> bool`" e così via.

Anche definendo "`'a set`" come sinonimo di "`'a list`" (aggiungendo "`type 'a set = 'a list;`" prima delle altre definizioni) la situazione non migliora. Si potrebbero esplicitamente specificare i tipi di "`emptyset`" e delle varie funzioni per usare "`'a set`", ma questo migliorerebbe la situazione solo a livello formale e non pratico (sarebbe, per esempio, ancora possibile applicare "`hd`" al risultato di "`addin`", in quanto non esiste differenza fra "`'a set`" ed "`'a list`").

Rinchiudendo le definizioni in una struttura (Figura 3.14) si possono al solito eliminare i problemi di "collisione fra identificatori", ma non si riesce ancora a distinguere il tipo "`'a set`" da "`'a list`". Per rendersene conto, si provi hd (Set.addin 1 Set.emptyset); (in teoria vorremmo che questa espressione generasse un errore, in quanto "`hd`" è una funzione che agisce su "`'a list`", non su "`'a set`).

E' però possibile definire un signature (Figura 3.15) che specifica l'interfaccia software del tipo "`'a set`" in modo generico, senza citare dettagli implementativi (come "`'a list`", etc...). Formalmente, un signature è definito da una serie di dichiarazioni di tipo (o tipo di variabile) contenute fra le keyword "`sig`" ed "`end`". Al solito, è possibile associare un nome ad un signature anonimo (usando la keyword "`signature`"; nell'esempio si usa "**signature** SET = ...", dove "`SET`" è il nome associato al signature).

Una volta definito un signature, è possibile creare un legame fra struttura e signature usando "`:>`" come mostrato in Figura 3.16 (si noti che invece di associare il nome "`SET`" al signature ed usare "`:> SET`" si potrebbe utilizzare un signature anonimo). Questo risolve i problemi precedentemente discussi.

```
val emptyset = [];

val rec isin =
      fn x =>
            (fn []    => false
              | y::l =>
                    if (x = y)
                        then
                            true
                        else
                            isin x l);

val addin =
      fn x =>
            fn l =>
                    if (isin x l)
                        then
                            l
                        else
                            x::l;

val rec removefrom =
      fn x =>
            (fn [] => []
              | y::l =>
                    if (x = y)
                        then
                            l
                        else
                            y::(removefrom x l));
```

Figura 3.13: Implementazione di insiemi (`'a set`) senza porsi problemi di riusabilità del codice.

3.11 Lavorare con Standard ML

Una macchina astratta per il linguaggio ML (vale a dire, una macchina astratta in grado di capire ed eseguire programmi scritti in ML) può essere implementata tramite interpreti o compilatori (meglio: può essere implementata in modo prevalentemente interpretato o prevalentemente compilato). Sebbene esistano alcuni compilatori ML, in questa sede ci concetreremo su un'implementazione prevalentemente interpretata di una macchina astratta che capisce Standard ML, perché un interprete permette una modalità di funzionamento interattiva che è inizialmente più intuitiva e semplice da usare. Questo significa che operativamente un utente può eseguire un interprete *interattivo* (quindi si può interagire con esso tramite un prompt). In particolare, in seguito verrà usato l'interprete "Poly/ML" (`http://www.polyml.org/`) anche se tutto quanto detto si può applicare senza problemi anche ad altri interpreti che siano compliant con Standard ML (come per esempio "Standard ML of New Jersey" - `http://www.smlnj.org/`). L'interprete interattivo di Poly/ML può essere lanciato eseguendo il comando "`poly`", che risponde come segue:

```
luca@nowhere:~$ poly
Poly/ML 5.2 Release
>
```

Notare che Poly/ML presenta il suo prompt (caratterizzato dal simbolo "`>`") 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 3.1); quindi, un interprete ML può essere inizialmente visto come un valutatore di espressioni.

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

```
structure Set = struct
  val emptyset = [];

  val rec isin =
        fn x =>
                (fn []    => false
                  | y::l =>
                        if (x = y)
                            then
                                true
                            else
                                isin x l);

  val addin =
        fn x =>
                 fn l =>
                        if (isin x l)
                            then
                                l
                            else
                                x::l;

  val rec removefrom =
        fn x =>
                (fn [] => []
                  | y::l =>
                        if (x = y)
                            then
                                l
                            else
                                y::(removefrom x l));
end;
```

Figura 3.14: Implementazione di insiemi (`'a set`) in una struttura.

```
5;
```

si ottiene risposta

```
val it = 5 : int
```

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

```
> 5 + 2;
val it = 7 : int
> 2 * 3 * 4;
val it = 24 : int
> (2 + 3) * 4;
val it = 20 : int
```

e così via.

Notare che il simbolo "`;`" serve da *terminatore* indicando all'interprete la fine di un'espressione. Quindi, un'espressione non viene valutata fino a che l'interprete non incontra un "`;`":

```
> 1 + 2
# + 3;
val it = 6 : int
```

```
signature SET = sig
  type 'a set

  val emptyset   : 'a set
  val isin       : ''a -> ''a set -> bool
  val addin      : ''a -> ''a set -> ''a set
  val removefrom : ''a -> ''a set -> ''a set
end;
```

Figura 3.15: Interfaccia software (signature) del tipo `'a set`. Si noti come non siano specificato alcun aspetto implementativo.

Notare che dopo aver premuto il tasto "Enter" senza aver digitato un "`;`" (dopo "`1 + 2`"), l'interprete interattivo va a capo, ma non presenta l'usuale prompt ("`>`"). Il nuovo carattere di prompt che viene presentato ("`#`") significa che l'interprete sta ancora attendendo il completamento dell'espressione corrente.

Da questi primi esempi si comincia a vedere come data un'espressione, l'interprete ML possa essere in grado di inferire il tipo dei parametri e del valore di ritorno. Per rendere possibile questo, ML deve usare una tipizzazione stretta e non può effettuare conversioni automatiche come fa per esempio il linguaggio C. Come conseguenza,

```
5 + 2.0;
```

genera un errore:

```
Error-Can't unify Int32.int/int with real (Overloading does not include type) Found near
+
( 5, 2.0)
Static errors (pass2)
```

Questo avviene perché per ML il numero `5` è un intero (tipo `int`), mentre il numero `2.0` è un floating point (tipo `real`). L'operatore `+` (somma) accetta come parametri due numeri interi o due numeri floating point, ma non è definito per operandi di tipo divero; per finire, come detto ML si rifiuta di convertire automagicamente numeri interi in floating point o viceversa (come invece fa un compilatore C: l'espressione `5 + 2.0` viene valutata da un compilatore C dopo aver converito `5` in `5.0`).

Come precedentemente detto, Standard ML fornisce altri tipi di base oltre a `int` e `real`:

```
> 2;
val it = 2 : int
> 2.0;
val it = 2.0 : real
> 2 > 1;
val it = true : bool
> "abc";
val it = "abc" : string
> #"a";
val it = #"a" : char
```

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

```
> "Hello, " ^ "world";
val it = "Hello, world" : string
```

notare che `"Hello, "` e `"world"` sono due valori di tipo stringa e "`^`" 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 un interprete ML 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 ML e quali siano le "entità" denotabili in ML (in

```
signature SET = sig
  type 'a set

  val emptyset   : 'a set
  val isin       : ''a -> ''a set -> bool
  val addin      : ''a -> ''a set -> ''a set
  val removefrom : ''a -> ''a set -> ''a set
end;

structure Set = struct
  val emptyset = [];

  val rec isin =
        fn x =>
                (fn []    => false
                  | y::l =>
                        if (x = y)
                            then
                                true
                            else
                                isin x l);

  val addin =
        fn x =>
                 fn l =>
                        if (isin x l)
                            then
                                l
                            else
                                x::l;

  val rec removefrom =
        fn x =>
                (fn [] => []
                  | y::l =>
                        if (x = y)
                            then
                                l
                            else
                                y::(removefrom x l));
end :> SET;
```

Figura 3.16: Legame fra struttura “Set” e signature “SET”.

un linguaggio imperativo, tali "entità" denotabili erano variabili, funzioni, tipi di dato, ...). Come in tutti i linguaggi funzionali, anche in ML 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:

```
> val n = 5;
val n = 5 : int
> val x = 2.0;
val x = 2.0 : real
```

Questi comandi associano il valore intero `5` al nome `n` ed il valore floating point `2.0` al nome `x`. Chiaramente, è possibile usare qualsiasi tipo di espressione per calcolare il valore a cui associare un nome:

```
> val x = 5.0 + 2.0;
val x = 7.0 : real
> val n = 2 * 3 * 4;
val n = 24 : int
```

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

```
> val x = 5.0 + 2.0;
val x = 7.0 : real
> val y = x * 2.0;
val y = 14.0 : real
> x > y;
val it = false : bool
```

notare che "**val** y = x * 2;" avrebbe generato un errore... 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 come ML esiste un tipo di dato "funzione":

```
> fn x => x + 1;
val it = fn : int -> int
```

in questo caso il valore risultante dalla valutazione dell'espressione è una funzione (nel senso matematico del termine) $\mathcal{N} \to \mathcal{N}$. Poiché non si è usata un'espressione "`val`", a tale funzione non è stato dato un nome. Una funzione può però essere applicata a dei dati anche senza darle un nome:

```
>  (fn x => x +  1) 5;
val it = 6 : int
```

Anche per le funzioni, l'interprete ML è generalmente in grado di inferire correttamente i tipi di dato:

```
> fn x => x + 1;
val it = fn : int -> int
> fn x => x + 1.0;
val it = fn : real -> real
```

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 ML corrisponde ad una definizione di variabile (tecnicamente, il seguente codice definisce una variabile "`doppio`" di tipo funzione da interi ad interi):

```
> val doppio = fn n => n * 2;
val doppio = fn : int -> int
> doppio 9;
val it = 18 : int
> doppio 4.0;
```

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

```
Error—Can't unify int with real (Different type constructors) Found near doppio(4.0
)
Static errors (pass2)
...
```

ML fornisce anche una sintassi semplificata, basata sull'espressione `fun`, per definire funzioni:

```
> fun doppio n = 2 * n;
val doppio = fn : int -> int
> doppio 9;
val it = 18 : int
```

La sintassi di `fun` appare più intuitiva della definizione esplicita `val ... = fn ...`, 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,

```
> val rec fact = fn n => if n = 0 then 1 else n * fact(n - 1);
val fact = fn : int -> int
> fact 5;
val it = 120 : int
```

Notare l'utilizzo di `val rec` invece che il semplice `val`. Una semplice definizione con `val` avrebbe generato un errore:

```
> val fact = fn n => if n = 0 then 1 else n * fact(n - 1);
Error—Value or constructor (fact) has not been declared   Found near
if =( n, 0) then 1 else *( n, fact(......))
Static errors (pass2)
```

Come si può vedere, senza usare la keyword `rec` il nome `fact` non è visibile nella definizione della funzione (il binding fra `fact` e la funzione non fa ancora parte dell'ambiente). Usando `fun` si possono invece definire funzioni ricorsive senza usare particolari accortezze:

```
> fun fact n = if n = 0 then 1 else n * fact(n - 1);
val fact = fn : int -> int
> fact 4;
val it = 24 : int
```

Per capire meglio la differenza fra `val` e `val rec` è utile considerare il seguente esempio:

```
> val f = fn n => n + 1;
val f = fn : int -> int
> val f = fn n => if n = 0 then 1 else n * f(n - 1);
val f = fn : int -> int
> f 4;
val it = 16 : int
```

da cosa deriva il risultato $f(4) = 16$? Dalla definizione "val f = fn n => if n = 0 then 1 else n * f(n − 1);" un utente inesperto potrebbe aspettarsi che il simbolo `f` venga associato alla funzione fattoriale e quindi $f(4) = 1 * 2 * 3 * 4 = 24$. Il risultato ottenuto è invece totalmente diverso, perché nel momento in cui viene valutata la seconda espressione "val f ... ", il simbolo `f` è sempre associato alla prima definizione ("fn n => n + 1"). Quindi, `f 4` viene valutata come "if 4 = 0 then 1 else 4 * f(n − 1);" dove `f` è valutata come "fn n => n + 1". Quindi, `f 4` è valutata come `4 * ((4 - 1) + 1)` vale a dire 16. L'esempio

```
> val f = fn n => n + 1;
val f = fn : int -> int
> val rec f = fn n => if n = 0 then 1 else n * f(n - 1);
val f = fn : int -> int
> f 4;
val it = 24 : int
```

```
val rec fact_tr = fn n =>
                  fn res =>
                    if n = 0 then
                               res
                             else
                               fact_tr (n - 1) (n * res);
val fact = fn n => fact_tr n 1;

fun fact_tr1 n res = if n = 0 then
                                res
                              else
                                fact_tr (n - 1) (n * res);
fun fact1 n = fact_tr1 n 1;
```

Figura 3.17: Fattoriale con ricorsione in coda.

```
val rec gcd = fn a => fn b => if b = 0 then
                                   a
                                 else
                                   gcd b (a mod b);


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

Figura 3.18: Massimo Comun Divisore.

avrebbe invece dato il risultato atteso $f(4) = 24$.

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

- Calcolo del fattoriale di un numero *usando ricorsione in coda* (Figura 3.17)
- Calcolo del massimo comun divisore fra due numeri (Figura 3.18)
- Soluzione del problema della torre di Hanoi (Figure 3.19 e 3.20)

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), ML permette di definire ed utilizzare nuovi *tipi di dato*. Il costrutto `type` permette di definire sinonimi per tipi di dato esistenti, mentre il costrutto `datatype` 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):

```
> rosso;
Error-Value or constructor (rosso) has not been declared   Found near rosso
Static errors (pass2)
> rosso 0.5;
Error-Value or constructor (rosso) has not been declared   Found near rosso(0.5)
Static errors (pass2)
> datatype colore = rosso of real | blu of real | verde of real;
datatype colore = blu of real | rosso of real | verde of real
> rosso;
val it = fn : real -> colore
> rosso 0.5;
val it = rosso 0.5 : colore
```

Prima della definizione "**datatype** colore = rosso **of** real blu of real — verde of real;—" la parola "`rosso`" non viene riconosciuta da Poly/ML, che lamenta l'utilizzo di un costruttore (`rosso`) che non

```
fun 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);
```

Figura 3.19: Torre di Hanoi.

```
fun 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);
```

Figura 3.20: Torre di Hanoi, versione alternativa.

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 `real` a `colore`).

Per finire, va notato come molti interpreti interattivi di standard ML non siano particolarmente pratici da usare, perche' per esempio non supportano i tasti freccia e/o la storia delle espressioni immesse (funzionalità a cui siamo abituati lavorando con i moderni terminali). Questi problemi possono essere aggirati in due modi:

1. Usando il comando `rlwrap` per abilitare i tasti freccia e la storia dei comandi (forniti dalla libreria `readline`) nel prompt dell'interprete ML. Per esempio, Poly/ML può essere invocato tramite il comando "`rlwrap poly`"

2. Editando programmi ML complessi in file di testo (generalmente con estensione `.sml`), 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 caricato dall'interprete usando la direttiva `use` di ML: "use " file .sml";"

Come esempio di utilizzo di `use`, si può inserire il programma di Figura 3.20 nel file `hanoi.sml`, usando il poprio editor di testo preferito (per esempio `vi`). Il programma può poi essere caricato in Poly/ML con

```
> use "hanoi.sml";
val move = fn : int -> string -> string -> string -> string
val it = () : unit
>
```

La funzione `move` è adesso definita come se fosse stata inserita direttamente da tastiera.

L'utilizzo di `use` 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.

3.12 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 ()[8] 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 -[9]. 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'.

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.

[8] 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

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

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.

3.13 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[10].

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

```
<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.

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

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
```

3.14 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 3.17) 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, associerà il nome `incrementa` a tale funzione):

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

Il tipo di questa variabile è `Num a => a -> a` (equivalente alla dizione matematica $f : \mathcal{X} \rightarrow \mathcal{X}$, 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 `... = \...`[11].

3.15 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:

[11] 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.

```
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)
```

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

Figura 3.21: Implementazione della funzione fattoriale con ricorsione in coda in Haskell, nascondendo la funzione `fact_tr` tramite costrutto `let`.

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)
```

3.16 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 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 3.21.

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

Figura 3.22: Implementazione della funzione fattoriale con ricorsione in coda in Haskell, nascondendo la funzione `fact_tr` tramite il costrutto `where`.

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 3.21), come mostrato in Figura 3.22.

Confrontando i due esempi di Figura 3.21 e 3.22, 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).

3.17 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;
```

3.18 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
```

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
```

```
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)
```

Figura 3.23: 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)
```

Figura 3.24: Conversione di denaro fra valute, versione finale.

(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 3.23 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 3.24 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...).

3.19 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 3.1); 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)[12]; 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:

```
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
```

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

```
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
```

Figura 3.25: Fattoriale con ricorsione in coda.

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 3.25)
- Calcolo del massimo comun divisore fra due numeri (Figura 3.26)
- Soluzione del problema della torre di Hanoi (Figure 3.27 e 3.28)

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):

```
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)
```

Figura 3.26: Massimo Comun Divisore.

```
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)
```

Figura 3.27: Torre di Hanoi.

```
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 3.28 nel file `hanoi.hs`, usando

```
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)
```

Figura 3.28: Torre di Hanoi, versione alternativa.

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).

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.

3.20 Qualche Nota sul C++

Sebbene il linguaggio C++ sia noto principalmente come linguaggio di programmazione orientato agli oggetti (derivando dal "C with Objects"), il moderno C++ è in realtà un linguaggio multi-paradigma, che può essere usato per scrivere programmi sia secondo il paradigma di programmazione orientato agli oggetti che secondo il paradigma imperativo ed addirittura il paradigma funzionale. Focalizzandosi su un sottoinsieme appropriato del linguaggio ed adottando opportune tecniche, un programmatore può essere in grado di scrivere programmi con strutture simili a quella che avrebbero se fossero scritti in linguaggi di programmazione tradizionalmente considerati "più funzionali" del C++ (come per esempio Standard ML o Haskell).

Nelle prossime sezioni sarà mostrato (in modo più o meno informale) come utilizzare il moderno C++ per in qualche modo "riprodurre" alcuni dei costrutti forniti da linguaggi funzionali:

- Prima di tutto, verrà mostrato come utilizzare il meccanismo di inferenza dei tipi di dato del C++ (tale meccanismo non è strettamente necessario per sviluppare programmi secondo uno stile funzionale, ma è comunque presente in molti linguaggi funzionali) e come dichiarare variabili "immutabili"

- Verrà poi visto come il moderno C++ supporti oggetti di tipo funzione, che possono essere memorizzati in variabili, passati come argomenti a funzioni e ritornati come valori da funzioni

- Finalmente, si mostrerà come usare Algebraic Data Types in C++ (come implementare qualcosa di simile alle varianti di Standard ML o Haskell, come fare pattern matching su tali tipi di dato, etc...)

3.21 Variabili in C++

Come noto, il concetto di variabile modificabile (come le variabili del linguaggio C++) è proprio dei linguaggi di programmazione imperativi e non è presente nei linguaggi funzionali (almeno, nei linguaggi funzionali puri).

Per poter utilizzare il C++ secondo il paradigma di programmazione funzionale, è quindi necessario evitare di modificare i valori delle variabili. Un modo semplice per poter imporre l'utilizzo di variabili immutabili (non modificabili) può consistere nel dichiarare tutte le variabili come costanti (utilizzando la keyword `const` del C/C++), inizializzandone il valore nella dichiarazione/definizione.

Un'altra caratteristica notevole di linguaggi come Standard ML o Haskell è la presenza di un meccanismo di *inferenza dei tipi*. In pratica, il programmatore non ha necessità di specificare il tipo dei valori utilizzati nel programma, perché il compilatore è in grado di inferirlo in base ad un'analisi statica del codice. Sebbene questa non sia un requisito fondamentale dei linguaggi di programmazione funzionale, un

```
#include <iostream>

auto somma(auto a, auto b)
{
  return a + b;
}

int main()
{
  const auto v1 = somma(1, 3);
  const auto v2 = somma(1.5, 3.2);

  std::cout << "V1: " << v1 << std::endl;
  std::cout << "V2: " << v2 << std::endl;

  return 0;
}
```

Figura 3.29: Esempio di utilizzo del meccanismo di inferenza dei tipi in C++.

programmatore abituato ad utilizzare Standard ML o Haskell (o simili) potrebbe avvertirne la mancanza passando al C++.

Il moderno C++ fornisce un meccanismo di inferenza dei tipi, basato sulla keyword `auto`: quando si definisce una variabile come "`auto` <*name*>", il compilatore C++ utilizza tecniche di analisi statica del codice (simili a quelle usate da compilatori ML o Haskell) per assegnare alla variabile il tipo corretto. In programmi C++ scritti secondo lo stile funzionale, le variabili dovrebbero quindi essere definite come

```
const auto <name1> = <val1>;
const auto <name2> = <val2>;
...
```

Per finire, è da notare come la keyword "`auto`" possa essere utilizzata anche per i parametri ed i valori di ritorno delle funzioni. Il programma in Figura 3.29 mostra un esempio di utilizzo dell'inferenza dei tipi in C++.

Riguardo ad un utilizzo così pervasivo della keyword "`auto`", esistono varie posizioni, tutte in qualche modo giustificabili in base a qualche punto di vista. Vari documenti raccomandano di utilizzare il meccanismo di inferenza dei tipi il più possibile, in modo da sfruttare al massimo le capacità dei moderni compilatori C++. D'altra parte, un utilizzo troppo aggressivo di questo meccanismo potrebbe far perdere allo sviluppatore il controllo sui tipi di dato utilizzati (si ricordi che anche linguaggi come Haskell o Standard ML permettono di aggiungere annotazioni per specificare i tipi dei dati). Esistono poi situazioni in cui l'inferenza dei tipi di dato proprio non può essere utilizzata (vedere Sezione 3.22 e Figura 3.31 in particolare).

Di sicuro, il meccanismo di inferenza dei tipi (con il conseguente utilizzo della keyword "`auto`") è utile per semplificare il codice e la dichiarazione di variabili atte a memorizzare funzioni (il cui tipo può essere molto complesso).

3.22 Funzioni come Valori in C++

Una delle caratteristiche fondamentali dei linguaggi di programmazione funzionali è quella di considerare le funzioni come valori denotabili, memorizzabili ed esprimibili.

Questo significa che "entità" di tipo funzione possono essere memorizzate in una variabile, passate come argomenti ad una funzione ed essere generati come risultato di un'espressione (quindi possono anche essere il valore di ritorno di una funzione). Per le ultime due caratteristiche si parla spesso di "funzioni di ordine superiore".

Anche il linguaggio C++ fornisce feature di questo tipo, definendo un template "`std::function`" che permette di istanziare classi contenenti entità di tipo funzione. La figura 3.30 mostra un esempio di utilizzo di tale template, che già ci fornisce alcuni spunti di riflessione interessanti:

```
#include <functional>
#include <iostream>

unsigned int inc(unsigned int n)
{
  return n + 1;
}

int main()
{
  unsigned int v = 3;
  std::function<unsigned int(unsigned int)> f = inc;

  std::cout << v << " + 1 = " << f(v) << std::endl;

  return 0;
}
```

Figura 3.30: Esempio di utilizzo del template `std::function` in C++.

- Come prima cosa è importante notare come il programma includa l'header "`functional`", che contiene la definizione del template che stiamo andando ad utilizzare
- La seconda cosa da notare è che "`std::function`" non è una classe, ma è un template, parametrizzato al tipo di ritorno ed ai tipi dei parametri della funzione
- La classe è quindi "`std::function<unsigned int(unsigned int)`" (a rappresentare una funzione che riceve un argomento di tipo "`unsigned int`" e ritorna un valore di tipo "`unsigned int`"

Per semplificare il codice e non dover esprimere esplicitamente il tipo della funzione, si può far ricorso al meccanismo di inferenza dei tipi del linguaggio C++, usando "**auto** f = inc;". Come notato in precedenza, la keyword "`auto`" è utilizzabile anche per la definizione della variabile "`v`" e per il valore di ritorno della funzione `inc()`. Il lettore potrebbe essere tentato di usare il meccanismo di inferenza dei tipi anche per l'argomento della funzione `inc()` (**auto** inc(**auto** n)), ma questo purtroppo non è possibile, perché non essendo `inc()` invocata direttamente il compilatore non può riuscire ad inferirne il tipo. Il programma semplificato tramite l'utilizzo di "`auto`" è riportato in Figura 3.31.

Dal punto di vista pratico, un'istanza del template `std::function` non è altro che una classe che ridefinisce l'operatore "`()`" per eseguire la funzione memorizzata nell'oggetto. Per esempio, nel codice di Figura 3.30 "`f`" è un oggetto il cui tipo ridefinisce l'operatore "`()`" come una funzione che riceve un parametro di tipo `unsigned int` e ritorna un valore di tipo `unsigned int` (uguale al parametro attuale incrementato di 1). Si noti che un oggetto di questo tipo può anche contenere dei membri dati e questi possono essere utili per implementare una chiusura. Questa è una differenza fondamentale fra un oggetto di questo tipo ed un semplice puntatore a funzione, che verrà approfondita in seguito.

Vediamo invece ora come sia possibile esprimere un valore di tipo funzione (meglio, una chiusura) come risultato di un'espressione C++, analogamente a quanto si farebbe (per esempio) in Standard ML con "**fn** x => ...". Questo è fattibile usando una *lambda expression* (circa equivalente al costrutto "`fn`" di Standard ML), come mostrato in Figura 3.32. In sostanza, come in altri linguaggi una lambda expression genera una funzione anonima: più specificamente, " [](**unsigned int** n) {**return** n + 1;}" è un valore di tipo funzione che riceve un argomento di tipo unsigned int e ritorna tale valore aumentato di 1. Si noti che in questo caso si sarebbe potuto usare " [](**auto** n) {**return** n + 1;}" perché essendo la funzione immediatamente applicata al parametro attuale "`v`" il compilatore può tranquillamente inferire il tipo del parametro formale "`n`".

La Figura 3.33 mostra una piccola evoluzione del programma di Figura 3.32, in cui si utilizza una variabile di tipo funzione per memorizzare la lambda expression. Questo costrutto è qualcosa di simile al "**val** f = **fn** n => ..." di Standard ML: in pratica,

```
val f = fn n => n + 1
```

diventa

```
#include <functional>
#include <iostream>

auto inc(unsigned int n)
{
  return n + 1;
}

int main()
{
  auto v = 3;
  auto f = inc;

  std::cout << v << " + 1 = " << f(v) << std::endl;

  return 0;
}
```

Figura 3.31: Esempio di utilizzo implicito (tramite inferenza dei tipi) del template `std::function` in C++.

```
#include <functional>
#include <iostream>

int main()
{
  auto v = 5;

  std::cout << v << " + 1 = " << ([](auto n) {return n + 1;})(v) << std::endl;

  return 0;
}
```

Figura 3.32: Esempio di lambda expression in C++.

```
#include <functional>
#include <iostream>

int main()
{
  auto v = 5;
  auto f = [](unsigned int n) {return n + 1;};

  std::cout << v << " + 1 = " << f(v) << std::endl;

  return 0;
}
```

Figura 3.33: Esempio di lambda expression in C++.

```
#include <functional>
#include <iostream>

auto somma_c(int a)
{
  return [a](int b) {
    return a + b;
  };
}

int main()
{
  auto somma2 = somma_c(2);
  auto somma_3 = somma_c(-3);

  std::cout << "2 + 3  = " << somma2(3)  << std::endl;
  std::cout << "-3 + 3 = " << somma_3(3) << std::endl;

  return 0;
}
```

Figura 3.34: Esempio di lambda expression con variabili catturate in C++.

```
auto f = [](unsigned int n) {return n + 1;}
```

(si noti come nel caso del C++ sia necessario specificare il tipo dell'argomento "n").

Come accennato, un oggetto di tipo funzione del C++ permette in realtà di memorizzare una chiusura (includendo quindi anche l'ambiente in cui risolvere i nomi non appartenenti all'ambiente locale della funzione). Questo è reso esplicito dalla sintassi delle lambda expression, che permette di specificare fra "[" e "] " le variabili non locali (variabili libere della funzione) da "catturare" (includere nei dati memorizzati dall'oggetto). Un esempio si può vedere in Figura 3.34, che mostra la forma curryificata di una funzione che somma due interi. Si noti come l'espressione "[a](**int** b) {" sostanzialmente crei una copia della variabile "a" e la salvi all'interno dell'oggetto ritornato dalla lambda expression. Questo è quello che in C++ viene chiamato "cattura di una variabile per valore" ("cattura" perché la variabile "a" sarebbe libera, ma invece viene risolta correttamente, quindi non lo è più; "per valore" perché sostanzialmente viene fatta una copia del valore di "a"). Sebbene il C++ permetta anche altre forme di cattura, tali forme non vengono trattate in questo contesto.

3.23 Algebraic Data Types in C++

L'algebra dei tipi di dato è un'algebra costituita da alcuni tipi di dato primitivi come supporto e da due operazioni di prodotto (prodotto cartesiano) e somma (unione disgiunta di varianti). In particolare, l'operazione di somma è data dal fatto che i valori di un tipo di dato definito dall'utente possono essere generati da varie funzioni (chiamate *costruttori*) che possono essere usate alternativamente. L'insieme dei valori generati da un singolo costruttore è chiamato *variante*, quindi l'insieme dei valori che un tipo di dato può assumere è l'unione disgiunta di più varianti.

Il linguaggio C++ permette di definire vari costruttori per una classe, quindi si potrebbe pensare di utilizzare questo meccanismo per implementare l'operazione di somma di un'algebra dei tipi di dato. Questa soluzione presenterebbe però vari inconvenienti:

- comprometterebbe la leggibilità del codice, in quanto i differenti costruttori avrebbero tutti lo stesso nome (che è anche il nome della classe) e si distinguerebbero solo per il numero o il tipo dei parametri
- renderebbe difficile distinguere a posteriori i valori generati dai vari costruttori (a meno di non aggiungere un campo "tipo" nella classe

Per rendersi conto di questo, si provi a definire un tipo di dato "Numero", che rappresenta un intero o un reale. In Standard ML, per esempio, si potrebbe usare "**datatype** Numero = Intero **of** int | Reale **of** real",

```
class Numero {
  private:
    union {
      int i;
      double f;
    } v;
    enum {Intero, Reale} tipo;
  public:
    Numero(int i) {
      v.i = i;
      tipo = Intero;
    }
    Numero(double d) {
      v.f = d;
      tipo = Reale;
    }
    bool I(void) {
      return tipo == Intero;
    }
    ...
};
```

Figura 3.35: Esempio di cattiva implementazione di un tipo di dati con varianti.

```
class Numero {
  public:
    virtual bool I(void) = 0;
    virtual bool R(void) = 0;
};

class Intero : Numero {
  private:
    int i;
  public:
    Intero(int _i) : i(_i) {};
    virtual bool I(void) {return true;};
    virtual bool R(void) {return false;};
};

class Reale : Numero {
  private:
    double r;
  public:
    Reale(int _r) : r(_r) {};
    virtual bool I(void) {return false;};
    virtual bool R(void) {return true;};
};
```

Figura 3.36: Esempio di implementazione di un tipo di dati con varianti usando ereditarietà.

che appare molto chiaro e leggibile. Usando invece il C++ con la tecnica descritta sopra, la definizione potrebbe essere simile a quella di Figura 3.35. Dal codice si può vedere come un valore di tipo "`Numero`" sia generabile dal costrutture "`Numero(int n)`" (prima variante) o dal costruttore "`Numero(double f)`" seconda variante e per discriminare i valori generati dai due costruttori la classe debba contenere un membro "`tipo`", che viene acceduto dalle funzioni che agiscono su valori di tipo "`Numero`". Per finire, il valore reale o intero e memorizzato in una `union`, la qual cosa permette di ridurre il footprint della struttura dati in memoria ma rende meno leggibile il codice.

Un'alternativa a questa soluzione può consistere nell'utilizzo del meccanismo di ereditarietà del linguaggio C++, modellando le varianti come classi derivate da una classe base. Un tipo di dati composto da più varianti è quindi implementabile come una classe base (che è in realtà una classe virtuale, non istanziabile) da cui derivano varie classi istanziabili, una per ogni variante. Le diverse varianti possono quindi essere discriminate semplicemente guardando al tipo (derivato) del valore ed il meccanismo di pattern matching può essere emulato usando metodi virtuali. La Figura 3.36 mostra come sia possibile usare questa tecnica per implementare il tipo di dato "`Numero`" mostrato nell'esempio precedente.

Capitolo 4

Tipi di Dato Ricorsivi

4.1 Tipi di Dato

Vari linguaggi di programmazione permettono all'utente di definire nuovi tipi di dato definendo per ogni nuovo tipo l'insieme dei suoi valori e le operazoni che si possono compiere su tali valori.

Un esempio di tipi di dato definiti dall'utente (forse il caso più semplice di tipo definito dall'utente) è costituito dai tipi di dato enumerativi. Per esempio, si può definire un tipo `colore` con valori `rosso`, `blu` e `verde` definendo poi le varie operazioni che si possono compiere su tali valori. Questo modo di definire nuovi tipi di dato rende chiaramente complicata la definizione di un tipo quando il numero di valori è molto alto e rende impossibile definire nuovi tipi di dato quando il numero di valori non è finito. Si può quindi generalizzare il concetto di tipo enumerativo, usando dei *costruttori di dati* invece che semplici valori costanti come `rosso`, `blu` e `verde`. Il nuovo tipo di dati che si va a definire ha quindi valori generati da uno o più costruttori, che operano su uno o più argomenti. E' chiaro che un costruttore che ha un argomento di tipo intero può generare un grande numero di valori (potenzialmente infiniti), risolvendo il problema. Si noti inoltre che i valori `rosso`, `blu` e `verde` di cui sopra altro non sono che casi particolari di costruttori di dati[1].

I valori del nuovo tipo di dato sono quindi partizionati in vari sottoinsiemi, chiamati *varianti*; in altre parole, l'insieme dei possibili valori (insieme di definizione del tipo di dato) è l'unione disgiunta delle varianti del tipo di dato. Ogni variante è quindi associata ad un costruttore, che genera i valori per tale variante; in altre parole, una variante è l'insieme dei valori generati da un singolo costruttore di dato. Notare che se si assume che due costruttori diversi non possano mai generare lo stesso valore, si può dire che l'insieme di definizione del tipo di dato è l'unione (o somma fra insiemi) delle varianti (invece che "unione disgiunta"). Più tardi diventerà chiaro come mai il concetto di "somma" è interessante in questo contesto.

Usando la sintassi di ML, si può quindi definire un nuovo tipo di dato in modo estremamente generico con

```
datatype <name> = <cons1> of [type1] | <cons2> of [type2] | ... ;
```

dove i vari costruttori `<cons1>`... `<consn>` rappresentano le varianti del tipo. Una cosa analoga si può fare usando Haskell (ma in questo caso i nomi dei tipi di dato e dei costruttori devono iniziare con lettere maiuscole):

```
data <name> = <cons1> [type1] | <cons2> [type2] | ... | <consn> [typen]
```

Notare che i costruttori sono vere e proprie funzioni che ricevono un argomento mappandolo in un valore del nuovo tipo di dato. In alcuni linguaggi come Standard ML, se servono più argomenti per un costruttore si può usare un argomento di tipo tupla; in altri linguaggi come Haskell un costruttore può ritornare una funzione che ritorna il valore del nuovo tipo di dato (o ancora una funzione...), usando la tecnica del currying. Haskell fornisce però una sintassi semplificata per queste situazioni, che rende valide definizioni come

```
data Test = Costruttore1 Int Double | Costruttore2 Char String
```

[1] In particolare, sono costruttori che non ricevono argomenti in ingresso - o hanno 0 argomenti. Quindi, ogni costruttore genera un unico valore, che viene identificato con il nome del costruttore.

```
datatype numero = intero of int | reale of real;
val sommanumeri = fn (intero a, intero b) => intero (a + b)
                   | (intero a, reale  b) => reale  ((real a) + b)
                   | (reale  a, intero b) => reale  (a + (real b))
                   | (reale  a, reale  b) => reale  (a + b);
val sottrainumeri = fn (intero a, intero b) => intero (a - b)
                     | (intero a, reale  b) => reale  ((real a) - b)
                     | (reale  a, intero b) => reale  (a - (real b))
                     | (reale  a, reale  b) => reale  (a - b);
 ...
```

Figura 4.1: Esempio di tipo di dato con due costruttori (che ricevono argomenti di tipo diverso), in Standard ML.

```
data Numero = Intero Int | Reale Double
sommanumeri (Intero a, Intero b) = Intero (a + b)
sommanumeri (Intero a, Reale  b) = Reale  (fromIntegral a + b)
sommanumeri (Reale  a, Intero b) = Reale  (a + fromIntegral b)
sommanumeri (Reale  a, Reale  b) = Reale  (a + b);
 ...
```

Figura 4.2: Esempio di tipo di dato con due costruttori (che ricevono argomenti di tipo diverso), in Haskell.

In questo caso, "`Costruttore1`" non è (come si potrebbe pensare) una funzione che accetta due argomenti (uno di tipo "`Int` ed uno di tipo "`Double`"), ma una funzione che accetta un unico argomento (di tipo "`Int`") e ritorna una funzione da "`Double`" a "`Test`". Il tipo di "`Costruttore1`" è quindi "`Int -> Double -> Test`".

Gli esempi di Figure 4.1 e 4.2 (basati su Standard ML ed Haskell) mostrano come definire un tipo `numero` che può avere valori interi o floating point.

Come precedentemente accennato, un costruttore di dato che riceva in ingresso più di un argomento può utilizzare una tupla per raggruppare i suoi argomenti, realizzando quindi un *prodotto cartesiano* fra gli insiemi di definizione degli argomenti[2]. Nuovi tipi di dato possono quindi essere creati a partire da tipi di dato esistenti utilizzando unioni (disgiunte) e prodotti cartesiani. Più semplicemente, si può parlare di *somme o prodotti* fra tipi di dato esistenti.

In questo modo, si definisce quindi un'algebra, o struttura algebrica, dei tipi di dato, costituita da un insieme insieme di sostegno - l'insieme dei tipi di dato - su cui sono definite un'operazione di somma ed un'operazione di prodotto. Si parla quindi spesso di "*Algebraic Data Types*" (ADT) per indicare i tipi di dato definiti in questo modo.

4.2 Ricorsione

La tecnica della *ricorsione* (strettamente legata al concetto matematico di *induzione*) è usata in informatica per definire un qualche genere di "entità"[3] basata su se stessa. L'esempio tipico riguarda le funzioni ricorsive: si può definire una funzione $f()$ esprimendo il valore di $f(n)$ come funzione di altri valori calcolati da $f()$ (tipicamente, $f(n-1)$). Ma si possono usare tecniche simili anche per definire un insieme descrivendone gli elementi in base ad altri elementi contenuti nell'insieme (esempio: se l'elemento n appartiene all'insieme, anche $f(n)$ appartiene all'insieme).

In generale, le definizioni ricorsive sono date "per casi", vale a dire sono composte da varie clausole. Una di queste è la cosiddetta *base* (detta anche *base induttiva*); esistono poi una o più clausole o *passi induttivi* che permettono di generare/calcolare nuovi elementi a partire da elementi esistenti. La base

[2]In alcuni linguaggi, come Standard ML, un costruttore di dato ha il vincolo di avere al massimo un argomento, quindi l'utilizzo del prodotto cartesiano è esplicito. Altri linguaggi, come Haskell, permettono sintatticamente di creare costruttori a più argomenti. In questo secondo caso il prodotto cartesiano è nascosto dalla sintassi del linguaggio, ma concettualmente è sempre presente.

[3]Il termine "entità" è qui usato informalmente per indicare genericamente funzioni, insiemi, valori... Ma, come vedremo a breve, anche tipi di dato!

```
i2n 0 = Zero
i2n x = Successivo (i2n (x - 1))

n2i Zero           = 0
n2i (Successivo n) = 1 + n2i n

nsum Zero           n = n
nsum (Successivo m) n = Successivo (nsum m n)
```

Figura 4.3: Implementazione di varie funzioni che lavorano sulla codifica di Peano dei numeri naturali.

è una clausola della definizione ricorsiva che non fa riferimento all'"entità" che si sta definendo (per esempio: il fattoriale di 0 è 1, o 0 è un numero naturale, o 1 è un numero primo, etc...) ed ha il compito di porre fine alla ricorsione: senza una base induttiva, si da' origine ad una ricorsione infinita (che non risulta essere troppo utile dal punto di vista pratico).

E' noto come sia possibile usare la ricorsione per definire funzioni (e come la ricorsione sia l'unico modo per ripetere ciclicamente un qualche tipo di operazione in linguaggi di programmazione funzionali - in altre parole, la ricorsione può essere considerata come una sorta di "equivalente funzionale" dell'iterazione). In sostanza, una funzione $f : \mathcal{N} \to \mathcal{X}$ è definibile definendo una funzione $g : \mathcal{N} \times \mathcal{X} \to \mathcal{X}$, un valore $f(0) = a$ ed imponendo che $f(n+1) = g(n, f(n))$.

Più nei dettagli, una funzione è definibile per ricorsione quando ha come dominio l'insieme dei naturali (o un insieme comunque numerabile); il codominio può essere invece un generico insieme $\mathcal{X}$. Come base induttiva, si definisce il valore della funzione per il più piccolo valore facente parte del dominio (per esempio, $f(0) = a$, con $a \in \mathcal{X}$) e come passo induttivo si definisce il valore di $f(n+1)$ in base al valore di $f(n)$; come detto sopra, questo si può fare definendo $f(n+1) = g(n, f(n)))$. Notare che il dominio di $g()$ è l'insieme delle coppie di elementi presi dal dominio e dal codominio di $f()$, mentre il codominio di $g()$ è uguale al codominio di $f()$.

E' importante notare che il concetto matematico di induzione può essere usato per definire non solo funzioni ma anche insiemi, proprietà dei numeri o altro. Per esempio, un insieme può essere definito per induzione indicando uno o più elementi che ne fanno parte (base induttiva) e definendo nuovi elementi dell'insieme a partire da elementi ad esso appartenenti (passo induttivo). Usando questa tecnica l'insieme dei numeri naturali può essere definito in base agli *assiomi di Peano*:

- Base induttiva: 0 è un naturale ($0 \in \mathcal{N}$)
- Passo induttivo: $n \in \mathcal{N} \Rightarrow n + 1 \in \mathcal{N}$ (in altre parole, esiste una funzione $s : \mathcal{N} \to \mathcal{N} - \{0\}$ che calcola il successivo di un numero naturale)

4.3 Tipi di Dati Ricorsivi

Come noto, per definire un tipo di dato occorre definire il suo insieme di definizione (insieme dei valori appartenenti al dato), più le operazioni che si possono applicare ai valori del tipo. Poiché un insieme può essere definito per induzione (come appena visto), si può pensare di applicare la ricorsione per definire nuovi tipi di dato.

Se si ricorda che i valori di un tipo possono essere generati da costruttori che ricevono in ingresso uno o più argomenti (in realtà, una n-upla di argomenti), diventa chiaro come applicare la ricorsione ai tipi di dato: un costruttore per un tipo di dato `T` può avere un argomento di tipo `T` (o, una tupla che contiene un elemento di tipo `T`). Chiaramente, per evitare ricorsione infinita occorre una base induttiva; questo vuol dire che una delle varianti del tipo deve avere costruttore che non ha argomenti di tipo `T`. Come passo induttivo, possono invece esistere altre varianti con costruttori che hanno argomenti di tipo `T`.

Per esempio, è possibile definire il tipo `naturale` basato sulla definizione ricorsiva dell'insieme dei naturali fornita da Peano: il tipo sarà costituito da due varianti, una delle quali (la base induttiva) avrà costruttore costante `zero` (che rappresenta il numero 0), mentre l'altra avrà costruttore `successivo`, che genera un naturale a partire da un naturale. Usando il costrutto `data` di Haskell, questo si scriverà

```
data Naturale = Zero | Successivo Naturale
```

```
class naturale {
  public:
    virtual unsigned int n2i(void) const = 0;
};

class zero: public naturale {
  public:
    zero(void) {
    };
    virtual unsigned int n2i(void) const
    {
      return 0;
    };
};

class successivo: public naturale {
    const naturale &prev ;
  public:
    successivo(const naturale &p) : prev(p)
    {
    };
    virtual unsigned int n2i(void) const
    {
      return prev.n2i() + 1;
    };
};
```

Figura 4.4: Esempio di implementazione in C++ della codifica di Peano dei numeri naturali

Per definire completamente il tipo `Naturale`, dovremo poi definire alcune semplici operazioni sui suoi valori. Per esempio la Figura 4.3 mostra come implementare conversione da `Int` a `Naturale` (funzione `i2n`), la conversione da `Naturale` a `Int` (funzione `n2i`) e somma di due numeri naturali (funzione `nsum`).

Per riferimento, la Figura 4.4 mostra come la codifica di Peano dei numeri naturali possa essere implementata in C++. Da questo semplice esempio (mancano l'implementazione della funzione `i2n` e dell'operatore "`+`", ma sono abbastanza semplici) si può immediatamente osservare una cosa interessante: poiché in C++ il costruttore di dati associato ad una classe ha lo stesso nome della classe, per poter avere un costruttore "`zero()`" ed un costruttore "`successivo()`" è stato necessario derivare due classi dalla classe astratta "`naturale`". Questo suggerisce che un tipo di dati con più varianti può essere implementato in C++ tramite una classe astratta di base che rappresenta il tipo di dato, da cui deriva una classe per ogni variante / costruttore.

Un'altra cosa interessante da notare è l'utilizzo di *riferimenti* per tenere conto della natura ricorsiva del tipo di dato "`naturale`". Questo comincia a suggerire che esiste una relazione (che diventerà più chiara considerando le liste) fra tipi di dati ricorsivi e riferimenti / puntatori: i linguaggi che non supportano esplicitamente la ricorsione sui tipi di dato sono costretti ad introdurre il tipo di dato puntatore o riferimento, mente i linguaggi che non supportano puntatori / riferimenti utilizzano i tipi di dati ricorsivi per evitare di perdere potere espressivo.

Per finire, la keyword "`const`" è utilizzata per garantire l'immutabilità delle strutture dati che codificano i naturali di Peano.

Sebbene interessante dal punto di vista matematico, la definizione del tipo `Naturale` non è troppo utile... Esistono però una serie di strutture dati che possono essere definite come tipi ricorsivi e sono molto utili: si tratta di liste, alberi e strutture dati dinamiche di questo genere. Per esempio, una lista di interi è definibile ricorsivamente come segue: una lista è vuota oppure è la concatenazione di un intero ed una lista. Il costruttore "lista vuota" costituisce la base induttiva, mentre il costruttore "concatenazione di intero e lista" costituisce il passo induttivo. Notare che una lista è una struttura dati ricorsiva perché il costruttore "concatenazione di intero e lista" riceve come secondo argomento un dato di tipo lista. Usando la sintassi di Haskell, una lista di interi è quindi definibile come

```
data Lista = Vuota | Cons Int Lista
```

```
isempty Vuota = True
isempty _     = False

lunghezza Vuota      = 0
lunghezza (Cons _ l) = 1 + lunghezza l

concatena Vuota b      = b
concatena (Cons e l) b = Cons e (concatena l b)
```

Figura 4.5: Esempi di funzioni che lavorano su liste in Haskell, usando solo `Vuota`, `Cons`, `car` e `cdr`.

dove `Vuota` è il costruttore di lista vuota, mentre `Cons` genera la variante contente tutte le liste non vuote. Si noti che il costruttore `Cons` ha un argomento di tipo `Int` e ritorna una funzione `Lista -> Lista`, perché una funzione Haskell non può avere più di un argomento (quindi, si ricorre al currying per supportare più argomenti), ma "`Cons Int Lista`" potrebbe essere anche vista come una funzione che riceve due argomenti: un intero ed una lista.

4.4 Liste Immutabili e Non

Il tipo di dato ricorsivo "`Lista`" introdotto nella sezione precedente costituisce un ben particolare tipo di lista, detto "lista immutabile". Il perché di questo nome può essere capito considerando le operazioni implementate su questo tipo. Poiché le liste immutabili (e le strutture dati immutabili in genere) sono prevalentemente utilizzate nei linguaggi funzionali, il linguaggio Haskell verrà usato nei prossimi esempi (ma gli esempi sono facilmente traducibili in altri linguaggi funzionali, come quelli della famiglia ML).

Le prime due operazioni sul tipo `Lista` sono i due costruttori `Vuota` e `Cons`, che generano rispettivamente liste vuote e liste non vuote (vale a dire, concatenazioni di interi e liste). Per poter operare sulle liste immutabili in modo generico, servono altre due operazioni dette tradizionalmente `car` e `cdr`. Data una lista non vuota, `car` ritorna il primo intero della lista (la cosiddetta "testa della lista"), mentre `cdr` ritorna la lista ottenuta eliminando il primo elemento. Sia `car` che `cdr` non sono definite per liste vuote. Una semplice implementazione di queste due funzioni è data da:

```
car (Cons v _) = v
cdr (Cons _ l) = l
```

Si noti che queste definizioni non sono esaustive: sia `car` che `cdr` sono infatti definite usando pattern matching su un valore di tipo `lista`, ma l'unico pattern presente nella definizione matcha con liste non vuote (costruttore `Cons`)... Non è presente alcun pattern che matchi con liste vuote (costruttore `Vuota`). Questo comporta che se `car` o `cdr` è applicata ad una lista vuota si genera un'eccezione (cosa non proprio "puramente funzionale"). Tutto questo riflette il fatto che `car` e `cdr` non sono definite per liste vuote.

Qualsiasi operazione sulle liste è implementabile basandosi solamente su `Vuota`, `Cons`, `car` e `cdr`. Per esempio, la Figura 4.5 mostra come controllare se una lista è vuota, calcolarne la lunghezza o concatenare due liste usando solo le primitive appena citate. Si noti come molte delle funzioni che agiscono sulle liste sono ricorsive (essendo le liste definite come un tipo di dato ricorsivo, questo non dovrebbe stupire più di troppo).

La funzione `isempty` è molto semplice e ritorna un valore booleano basandosi su pattern matching: se la lista passata come argomento matcha una lista generata da dal costruttore `Vuota` (vale a dire, se è una lista vuota), ritorna `True` altrimenti (wildcard pattern) ritorna `False`.

La funzione `lunghezza` lavora in modo ricorsivo, usando il pattern `Vuota` come base induttiva (la lunghezza di una lista vuota è 0) e dicendo che se la lunghezza di una lista `l` è n, allora la lunghezza della lista ottenuta inserendo un qualsiasi intero in testa ad `l` è $n + 1$ (passo induttivo).

La funzione `concatena`, infine, usa la concatenazione di una lista vuota con una generica lista `b` come base induttiva (concatenando una lista vuota con una lista `b`, si ottiene `b`). Il passo induttivo è costituito dal fatto che concatenare una lista composta da un intero `n` ed una lista `l` con una lista `b` equivale a creare una lista composta dall'intero `n` seguito dalla concatenazione di `l` e `b`.

Per finire, una funzione per inserire un numero intero in una lista ordinata può essere implementata come mostrato in Figura 4.6. Quando si vuole inserire un nuovo elemento in una lista vuota (`l = Vuota`), la lista risultante è creata (tramite `Cons`) aggiungendo il nuovo elemento in testa alla lista. Questo viene fatto anche se l'elemento che si vuole inserire è più piccolo del primo elemento della lista (`n < car l`).

```
inserisci = \n -> \l ->
    if ((l == Vuota) || (n < car l))
      then
        Cons n l
      else
        Cons (car l) (inserisci n (cdr l))
```

Figura 4.6: Inserimento ordinato in una lista usando Haskell.

Altrimenti, viene creata (sempre tramite `Cons`) una nuova lista formata dalla testa di `l` (`car l`) seguita dalla lista creata inserendo il numero nel resto di `l` (`inserisci n (cdr l)`). In ogni caso, una lista contenete `n` inserito nella giusta posizione viene creata tramite una o più invocazioni di `cons`, invece di modificare la lista `l`.

Per esempio, si consideri cosa accade quando si invoca

```
inserisci 5 (Cons 2 (Cons 4 (Cons 7 Vuota)))
```

poiché $5 > 2$, `inserisci` invoca

```
Cons 2 (inserisci 5 (Cons 4 (Cons 7 Vuota)))
```

quindi

```
Cons 2 (Cons 4 (inserisci 5 (Cons 7 Vuota)))
```

che finalmente ritorna "Cons 2 (Cons 4 (Cons 5 (Cons 7 Vuota)))". Sono quindi stati creati ben 3 nuovi valori di tipo `Lista`.

Per capire meglio il comportamento di una lista immutabile ed il perché del suo nome, è utile vedere come essa possa essere definita in un linguaggio che non supporta direttamente tipi di dati ricorsivi, come il linguaggio C. In questo caso, si usano dei puntatori per collegare i vari elementi della lista: in particolare, ogni elemento della lista è rappresentato da una struttura composta da un campo intero (il valore di tale elemento) ed un puntatore al prossimo elemento della lista. Quindi, invece di avere un tipo di dato `lista` definito basandosi su se stesso si ha un tipo di dato `lista` definito usando un puntatore a `lista`. I due costruttori `vuota` e `cons` sono implementati come funzioni che ritornano un puntatore a `lista` (queste funzioni allocano quindi dinamicamente la memoria necessaria a contenere una stuttura `lista`) e le funzioni `car` e `cdr` semplicemente ritornano i valori dei campi della struttura `lista`. Una lista è terminata da un elemento che ha puntatore al prossimo elemento uguale a `NULL` (questo si può considerare l'equivalente della base induttiva). Riassumendo, una semplice implementazione in C può apparire come in Figura 4.7. Si noti che per semplificare l'implementazione il risultato di `malloc()` non viene controllato (un'implementazione più "robusta" dovrebbe invece controllare che `malloc()` non ritorni `NULL`).

La funzione `inserisci` può quindi essere definita analogamente a quanto fatto in Haskell, usando solo le funzioni `vuota()`, `cons()`, `car()` e `cdr()` come mostrato in Figura 4.8. Questa definizione usa il costrutto di *if aritmetico* del linguaggio C, che è equivalente all'espressione `if...then...else...end` di Haskell ma è forse meno leggibile rispetto ad un costrutto di selezione "tradizionale". L'implementazione di Figura 4.9 è equivalente (anche se non strutturata e "meno funzionale").

Sia ora `l` un puntatore alla lista di Figura 4.10, contenente i valori 2, 4, e 7 e si consideri cosa avviene quando si invoca `l = inserisci(5, l);`. Poiché $5 > 2$, `inserisci` invoca ricorsivamente `inserisci(5, cdr(l));`, con `cdr(l)` che è un puntatore al secondo elemento della lista `l` (è quindi un puntatore ad una lista contenente i valori 4 e 7). Il risultato di questa invocazione ricorsiva di `inserisci()` sarà poi passato a `cons()` (che allocherà dinamicamente una nuova struttura di tipo `lista`). Poiché `car(cdr(l))` vale 4 e $5 > 4$, `inserisci(5, cdr(l));` invocherà ancora ricorsivamente `inserici()`, con parametri 5 e `cdr(cdr(l))`. A questo punto, poiché `car(cdr(cdr(l)))` vale 7 e $5 > 7$, `inserisci(5, cdr(cdr(l)));` ritornerà `cons(5, cdr(cdr(l))`, allocando dinamincamente una struttura di tipo `lista`. `cons()` sarà poi invocata altre 2 volte, e come risultato `inserisci(5, l);` ritornerà una lista costituita da 3 nuovi elementi dinamicamente allocati (`cons()` è stata invocata 3 volte), senza modificare alcun elemento della lista di Figura 4.10. Questo è il motivo per cui questo tipo di lista viene detto "immutabile": i campi `val` e `next` della struttura `lista` vengono settati quando la struttura è allocata (da `cons()`) e non vengono mai più variati. Il risultato finale è visibile in Figura 4.11, che mostra in basso i 3 nuovi elementi allocati dinamicamente (si noti che ora il puntatore `l` punta all'elemento di valore 2 più in basso).

```
struct lista {
  int val;
  struct lista *next;
};

struct lista *vuota(void)
{
  return NULL;
}

struct lista *cons(int v, struct lista *l)
{
  struct lista *res;

  res = malloc(sizeof(struct lista));
  res->val = v;
  res->next = l;

  return res;
}

int car(struct lista *l)
{
  return l->val;
}

struct lista *cdr(struct lista *l)
{
  return l->next;
}
```

Figura 4.7: Implementazione di liste immutabili usando il linguaggio C.

E' anche importante notare che nell'esempio precedente il puntatore al primo elemento di `l` può essere stato "perso", lasciando delle zone di memoria allocate dinamicamente ma non più raggiungibili: se un programma invoca

```
l = vuota();
l = inserisci(4, l);
l = inserisci(2, l);
l = inserisci(7, l);
l = inserisci(5, l);
```

le strutture dinamicamente allocate dalla seconda invocazione di `inserisci()` (vale a dire, `inserisci(2, l);`) non hanno più alcun puntatore che "permtta di raggiungerle".

Tornando all'esempio precedente, quando si invoca `l = inserisci(5, l)` le strutture contenenti i primi due elementi della "vecchia lista" possono quindi essere non più raggiungibili, ma la memoria nelle quali sono memorizzate non è stata rilasciata da alcuna chiamata a `free()`. Questo indica chiaramente che un'implementazione delle liste organizzata in questo modo rischia di generare dei *memory leak*, quindi un qualche tipo di meccanismo di garbage collection si rende necessario.

La precedente implementazione di liste immutabili in C può essere confrontata con un'implementazione "più tradizionale", non basata su strutture dati immutabili ma su variabili modificabili, mostrata in Figura 4.12.

Si noti che questa implementazione di `inserisci()` modifica il campo `next` dell'elemento precedente a quello che viene inserito, quindi la lista non è immutabile. D'altra parte, non soffre dei problemi di memory leak evidenziati per l'implementazione precedente (in sostanza, questo ci dice che tecniche di garbage collection sono necessarie solo se si utilizzano strutture dati immutabili). Il risultato dell'operazione

```
struct lista *inserisci(int n, struct lista *l)
{
  return ((l == vuota()) || (n < car(l))) ?
                   cons(n, l) : cons(car(l), inserisci (n, cdr(l)));
}
```

Figura 4.8: Inserimento ordinato in liste immutabili usando il linguaggio C.

```
struct lista *inserisci(int n, struct lista *l)
{
  if ((l == vuota()) || (n < car(l))) {
    return cons(n, l);
  } else {
    return cons(car(l), inserisci (n, cdr(l)));
  }
}
```

Figura 4.9: Inserimento ordinato in liste immutabili usando il linguaggio C senza if aritmetico.

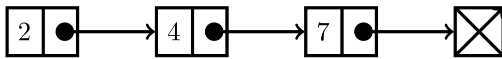

Figura 4.10: Lista di esempio contenente gli interi 2, 4 e 7.

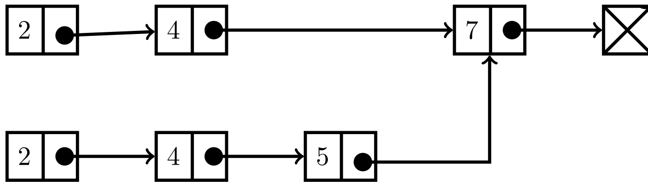

Figura 4.11: Lista della Figura 4.10 dopo aver eseguito inserisci (5, l);.

l = inserisci (5, l); usando questa implementazione delle liste è visibile in Figura 4.13 (si noti che la freccia uscente dall'elemento "4" è cambiata).

Per finire, a titolo di confronto la Figura 4.14 mostra un'implementazione di una lista immutabile in Java (si ricordi che una VM Java implementa un garbage collector di default, quindi i memory leak dovuti all'utilizzo di strutture dati immutabili non sono un problema).

E' interessante notare come in Java il qualificatore `final` permetta di specificare esplicitamente che i campi della classe non verranno mai modificati (la struttura dati è immutabile). Inoltre, si noti che Java (come molti altri linguaggi orientati agli oggetti) obbliga ad usare lo stesso nome per i costruttori e per la classe. Non ci saranno quindi due costruttori `Vuota` e `Cons` per le due varianti, ma due costruttori chiamati entrambi `List` che si distinguono per il numero ed il tipo dei propri argomenti (un costruttore - quello di lista vuota - non ha argomenti, mentre l'altro - quello corrispondente a `Cons` - ha un argomento di tipo `int` ed uno di tipo `List`).

```
struct lista {
  int val;
  struct lista *next;
};

struct lista *inserisci(int n, struct lista *l)
{
  struct lista *res, *prev, *new;

  new = malloc(sizeof(struct lista));
  new->val = n;

  res = l;
  prev = NULL;
  while ((l != NULL) && (l->val < n)) {
    prev = l;
    l = l->next;
  }
  new->next = l;
  if (prev) {
    prev->next = new;
  } else {
    res = new;
  }

  return res;
}
```

Figura 4.12: Implementazione "tradizionale" delle liste usando il linguaggio C, senza usare strutture dati immutabili.

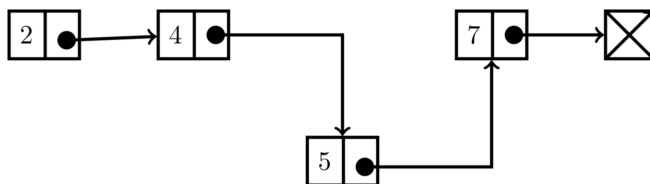

Figura 4.13: Lista della Figura 4.10 dopo aver eseguito inserisci (5, l); usando le liste "tradizionali" (non immutabili).

```
public class List {
    private final Integer val;
    private final List next;

    public List(){
        val=null;
        next=null;
    }
    public List(int v, List l) {
        val = v;
        next = l;
    }
    public int car() {
        return val;
    }
    public List cdr() {
        return next;
    }


    public void stampaLista() {
        if (next!=null) {
            System.out.println(val);
            next.stampaLista();
        }
    }

    public List inserisci(int n){
        if (next==null || n < val)
        return new List(n,this);
        else return new List(car(),cdr().inserisci(n));
    }

    public static void main(String a[]){ // main di test
        List l = new List();
        l = l.inserisci(1);
        l = l.inserisci(5);
        l = l.inserisci(3);
        l = l.inserisci(9);
        l = l.inserisci(7);
        l.stampaLista();
    }
}
```

Figura 4.14: Implementazione di liste immutabili usando il linguaggio Java.

Capitolo 5

Fixpoint ed Altre Amenità

5.1 Fixed Point Combinator

In generale, un combinator è una funzione di ordine superiore (funzione che riceve altre funzioni come argomenti e/o ritorna funzioni come risultato) senza variabili libere (vale a dire, tutte le variabili usate nel combinator sono risolubili nel suo ambiente locale: si tratta quindi di variabili locali o parametri)[1].

Particolarmente importanti nell'ambito della programmazione funzionale (e del suo fondamento teorico, il λ calcolo) sono i *fixed point combinator*. Un fixed point combinator è un combinator che calcola il *punto fisso* della funzione passata come argomento. In altre parole, se g è una funzione, un fixpoint combinator è una funzione Fix senza variabili libere tale che $Fix(g) = g(Fix(g))$.

Si noti che definire una generica funzione di ordine superiore Fix che calcoli il punto fisso della funzione passata come argomento è abbastanza facile: per definizione

$$Fix(g) = g(Fix(g))$$

ma questa espressione può essere vista anche come una definizione di Fix; in altre parole, usando una piccola estensione del λ calcolo che ci permette di associare nomi a λ espressioni

$$Fix(g) = g(Fix(g)) \Rightarrow Fix = \lambda g.g(Fix(g)). \tag{5.1}$$

Come prima considerazione, è interessante notare come se si prova a valutare l'equazione 5.1 usando una strategia *eager* (valutazione per valore), si ottiene

$$Fix(g) = (\lambda g.g(Fix(g)))g \rightarrow_\beta g(Fix(g)) = g((\lambda g.g(Fix(g)))g) \rightarrow_\beta g(g(Fix(g))) = ...$$

e la riduzione diverge. Questo è dovuto al fatto che una strategia di valutazione eager tenderà sempre a valutare l'espressione "$Fix(g)$" più interna espandendola in "$g(Fix(g))$" e così via... Usando invece una strategia *lazy* (valutazione per nome), "$Fix(g)$" viene valutata solo quando g effettivamente la invoca ricorsivamente (quindi, non viene valutata quando si arriva alla base induttiva... Questo garantisce che se se le varie invocazioni ricorsive portano alla base induttiva allora la valutazione di $Fix(g)$ non diverge). Questa osservazione sarà importante quando andremo a provare ad implementare un fixed point combinator in SML.

Un'altra osservazione importante è che la funzione Fix definita in Equazione 5.1 permette (per costruzione) di calcolare il punto fisso del suo argomento g, ma non è un combinator: in particolare, la definizione di Fix usa la variabile Fix che è libera, non essendo legata da alcuna λ (mentre la definizione di un combinator dovrebbe contenere solo variabili legate). La cosa è particolarmente rilevante perché Fix è proprio il nome della funzione che si sta definendo, quindi la definizione di Fix è ricorsiva. In altre parole, abbiamo solo spostato la ricorsione dalla definizione di g a quella di Fix.

Esistono comunque molti diversi fixed point combinator che permettono di calcolare il punto fisso di una funzione senza utilizzare ricorsione esplicita ne' nella definizione della funzione ne' nella definizione del combinator[2]. L'esistenza dei fixed point combinator ha un'importanza teorica notevole (in pratica, mostra che il λ calcolo "puro" - senza ambiente o estensioni che permettano di associare nomi ad espressioni - può implementare la ricorsione ed è Turing completo); dal punto di vista pratico, significa invece che un

[1]Si ricordi che nel caso particolare del λ calcolo un combinator è definito come una λ espressione che non contiene variabili libere, consistentemente con questa definizione più generale.

[2]Più precisamente, ne esiste un'infinità numerabile.

linguaggio funzionale che non implementi "`val rec`" (o l'equivalente "`fun`"), "`let rec`" o simili...) può comunque permettere l'implementazione di funzioni ricorsive!

Il più famoso fra i fixed-point combinator è l'**Y combinator**, sviluppato da Haskell Curry (eh si, i nomi alla fine sono sempre gli stessi...), la cui definizione (usando il λ calcolo) è:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)). \tag{5.2}$$

Si noti che in letteratura esistono alcune piccole inconsistenze a livello terminologico: mentre in generale l'Y combinator è *un particolare* fixed point combinator, alcuni tendono ad usare il termine "Y combinator" per identificare un generico fixed point combinator (e scrivono quindi che esiste un'infinità di Y combinator).

5.2 Implementazione in SML

Il Y Combinator è definito come

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Provando una semplice conversione nella sintassi SML ("λ" diventa "`fn`" e "." diventa "`=>`") si ottiene

```
val Y = fn f => (fn x => f (x x))(fn x => f (x x))
```

Questa espressione non è però accettata da un compilatore o interprete SML. Per esempio, Poly/ML genera il seguente errore:

```
> val Y = fn f => (fn x => f (x x))(fn x => f (x x));
poly: : error: Type error in function application.
    Function: x : 'a -> 'b
    Argument: x : 'a -> 'b
    Reason:
        Can't unify 'a to 'a -> 'b (Type variable to be unified occurs in type)
Found near (fn x => f (x x)) (fn x => f (x x))
poly: : error: Type error in function application.
    Function: x : 'a -> 'b
    Argument: x : 'a -> 'b
    Reason:
        Can't unify 'a to 'a -> 'b (Type variable to be unified occurs in type)
Found near (fn x => f (x x)) (fn x => f (x x))
Static Errors
```

Il significato di questo errore diventa chiaro cercando di capire qual'è il tipo di "`x`". Supponendo che il tipo di "`x`" sia "`T`", si ha che:

- Poiché "`(x x)`" indica che "`x`" è *una funzione* applicata ad un argomento, il tipo "`T`" di "`x`" deve essere una funzione. Consideriamo il caso più generico: una funzione $\alpha \to \beta$, quindi $\mathtt{T} = \alpha \to \beta$
- D'altra parte, la funzione "`x`" è applicata proprio all'argomento "`x`". Quindi il tipo α dell'argomento della funzione "`x`" deve essere uguale al tipo di "`x`" ("`T`"): $\mathtt{T} = \alpha$.

Mettendo tutto assieme si ottiene

$$\mathtt{T} = \alpha \to \beta \wedge \mathtt{T} = \alpha \Rightarrow \alpha = \alpha \to \beta$$

che definisce il tipo "`T`" in modo ricorsivo. Poiché Standard ML supporta tipi di dati ricorsivi, verrebbe quindi da pensare che dovrebbe essere in grado di supportare il tipo "`T`" di cui stiamo parlando. Invece, una ricorsione come $\alpha = \alpha \to \beta$ non ha molto senso in un linguaggio che fa un controllo stretto dei tipi come Standard ML. Per capire come mai, è necessario entrare un po' più nei dettagli dei meccanismi di definizione di tipi ricorsivi.

Di fronte ad una definizione come $\alpha = \alpha \to \beta$, si parla di tipi *equi-ricorsivi* (equi-recursive data types) se α ed $\alpha \to \beta$ rappresentano lo stesso tipo di dati (stessi valori, stesse operazioni sui valori, etc...). Si parla invece di tipi *iso-ricorsivi* (iso-recursive data types) se il tipo α ed il tipo $\alpha \to \beta$ non sono uguali, ma esiste un isomorfismo (funzione $1 \to 1$, invertibile, che ad ogni valore di α associa un valore di $\alpha \to \beta$ e viceversa) dall'uno all'altro. Questo isomorfismo (che stabilisce che i due tipi sono in sostanza equivalenti)

è la funzione che in Standard ML abbiamo chiamato "costruttore". Capendo che Standard ML supporta quindi tipi iso-ricorsivi (un valore di α può essere generato tramite un apposito costruttore a partire da un valore di $\alpha \to \beta$), ma non equi-ricorsivi (α ed $\alpha \to \beta$ **non** possono essere lo stesso tipo!) diventa allora chiaro perché la definizione di `Y` data poco fa generi un errore sintattico (in particolare, un errore di tipo).

Si noti che questo non significa che la definizione dell'Y combinator (Equazione 5.2) sia "sbagliata", ma semplicemente che non è implementabile direttamente in Standard ML. L'Y combinator è definito usando il λ-calcolo, in cui ogni identificatore è legato ad una funzione, il cui tipo non è importante. Utilizzando linguaggi con controlli sui tipi "meno stretti" rispetto a Standard ML (per esempio, Lisp, Scheme, Python, Javascript, etc...) o linguaggi che supportano tipi equi-ricorsivi (per esempio, OCaml con apposite opzioni), l'Equazione 5.2 è implementabile senza problemi.

Per capire meglio come risolvere questo problema, consideriamo la "parte problematica" dell'espressione precedente (l'applicazione di funzione "`(x x)`"), limitandoci per il momento alla funzione (chiaramente di ordine superiore) "`fn x => (x x)`.

Poiché in Standard ML un tipo di dato ricorsivo `T` può essere definito usando il costrutto `datatype` ed (almeno) un costruttore che mappa valori di un tipo che dipende da `T` in valori di `T` (definizione di tipo iso-ricorsivo), si può definire qualcosa dei tipo $T = F(T \to \beta)$ per "simulare" il tipo (equi-ricorsivo) $\alpha = \alpha \to \beta$ di `x`. Il dipo `T` risultante sarà quindi funzione del tipo β ed in Standard ML questo si indica con "`'b T`".

Ricordandosi la sintassi di `datatype` (un tipo di dato dipendente dalla type variable β in Standard ML si definisce usando `datatype 'b <nometipo> = ...`), il tipo di dato che ci interessa è

```
datatype 'b T = F of ('b T -> 'b)
```

dove come detto "`'b T`" è il nome del tipo, mentre "`F`" è il nome del costruttore che mappa valori di `'b T -> 'b` in valori di `'b T`.

A questo punto, è possibile usare "`'b T`" come tipo di "x" facendo si che "x" sia una funzione `'b T -> 'b` da `'b T` a `'b`. L'argomento (parametro attuale) di tale funzione deve quindi essere di tipo `'b T`, generabile da "x" usando il costruttore `F`. Invece di "**fn** x => x x" si scrive allora:

```
fn x => (x (F x))
```

E standard ML è perfettamente in grado di capire e processare questa definizione. Si noti come l'impossibilità di usare tipi equi-ricorsivi ci ha obbligati ad usare il costruttore `F`. Il tipo della funzione definita qui sopra risulta essere `('a T -> 'a) -> 'a` (funzione che mappa valori "x" di tipo "`'a T -> 'a`" in valori di tipo "`'a`").

Ora che abbiamo visto come risolvere il problema del tipo di "x", è possibile tornare all'espressione originaria del Y combinator, che presenta ancora un problema analogo: "`fn f(x (F x))`" va applicata a se stessa, quindi il suo tipo è ricorsivo come il tipo di "x"! Ancora una volta il problema può essere risolto usando il tipo di dato "`'b T`" precedentemente definito: "**fn** f => (**fn** x => f (x (F x)))" ha tipo "`('a -> 'b) -> ('a T -> 'a) -> 'b`", quindi il valore a cui è applicata deve avere tipo "`('a -> 'b) -> ('a T -> 'a)`... Bisogna quindi sostituire un "`('a T -> 'a)`" (tipo di "x") con un "`'a T`" (ottenibile da "x" applicandogli il costruttore "`F`"). L'argomento sarà quindi "**fn** f => (**fn** (F x) => f (x (F x)))"

Come risultato, l'espressione dell'Y combinator è:

```
val Y = fn f => (fn x => f(x (F x)))(fn (F x) => f(x (F x)))
```

e questa espressione è accettata da SML! Riassumendo, la prima parte dell'espressione ("`fn x => f(x (F x))`") ha tipo "`('a T -> 'a) -> 'b`", mentre la seconda parte ("`fn (F x) => f(x (F x))`") ha tipo "`'a T -> 'b`" ed è quindi usabile come argomento per la prima semplicemente ponendo $\beta = \alpha$.

Il tipo della funzione risultante è "`('a -> 'a) -> 'a`", dove "`'a`" è chiaramente un tipo funzione (per il fattoriale, per esempio, è una funzione "`int -> int`").

La discussione di qui sopra, che mostra come sia possibile implementare il fixed point combinator Y in linguaggi (come Standard ML) con tipizzazione stretta che non supportano tipi di dati equi-ricorsivi, ci permette di intuire una cosa importante: Y "elimina la ricorsione" dall'operatore fixed point (l'Equazione 5.1 utilizza ricorsione esplicita) spostando la ricorsione sul tipo di dato (richiede infatti di usare tipi di dati ricorsivi di qualche genere). Questo non è immediatamente visibile nell'Equazione 5.2 o in implementazioni di Y in scheme (o simili), ma diventa più chiaro provando ad implementare Y in (per esempio) Standard ML.

Una volta che ML ha "accettato e capito" la nostra definizione di Y, potrebbe sembrare che tutti i problemi siano risolti. Testando il combinator Y definito sopra, però, si incontrano altre brutte sorprese.

Proviamo per esempio ad utilizzare la funzione Y appena definita per definire la funzione fattoriale. Prima di tutto, definiamo una versione "chiusa" `fact_closed`[3] della funzione fattoriale come funzione come

```
val fact_closed = fn f => fn n => if n = 0 then 1 else n * f (n - 1)
```

A questo punto, si potrebbe pensare di calcolare la funzione fattoriale

```
val fact = Y fact_closed
```

ma questo porta ad una computazione divergente (ricorsione infinita).

Prima di vedere come risolvere questo problema, si noti che il tipo di "`fact_closed`" è "`(int -> int) -> (int -> int)`", quindi, poiché Y ha tipo "`('a -> 'a) -> 'a`" "`Y fact_closed`" ha tipo "`int -> int`".

Veniamo ora a risolvere il problema della non convergenza di "`Y fact_closed`". Ancora una volta, il fatto che la nostra implementazione non fornisca il risultato sperato in Standard ML non significa che l'Y combinator "è sbagliato". Il "problema" è semplicemente dovuto al fatto che l'Y combinator non converge in caso di valutazione *eager* (o *per valore*), che è la strategia usata da Standard ML (si ricordi la prima osservazione sull'Equazione 5.1 nella Sezione 5.1). In altre parole, in un linguaggio che valuta le espressioni per valore dobbiamo usare un altro tipo di fixed point combinator. Fra i vari fixed point combinator che funzionano correttamente usando la valutazione eager (quindi anche in Standard ML), il più famoso è probabilmente il cosiddetto "Z combinator". La sua definizione è

$$Z = \lambda f.(\lambda x.f(\lambda v.((xx)v)))(\lambda x.f(\lambda v.((xx)v)))$$

e come si vede può essere ottenuto dall'Y combinator astraendo l'applicazione "`(x x)`" rispetto ad una variabile "`v`". Informalmente, si può dire che questa nuova astrazione impedisce ad un valutatore eager di entrare in un ciclo infinito di riduzioni.

La traduzione in SML a questo punto è semplice:

```
val Z = fn f =>
        (fn x => f (fn v => (x (F x))v))(fn (F x) => f (fn v => (x (F x))v))
```

ed il tipo di "`Z`" è ora "`(('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)`" (alcune parentesi le ho aggiunte io; Poly/ML stampera' invece "`(('a -> 'b) -> 'a -> 'b) -> 'a -> 'b`") indicando esplicitamente che il tipo prima indicato come "`'a`" (α) è in realtà una funzione "`'a -> 'b`". Stavolta, il combinator funziona correttamente:

```
> val fact_closed = fn f => fn n => if n = 0 then 1 else n * f (n - 1);
val fact_closed = fn: (int -> int) -> int -> int
> val fact = Z fact_closed;
val fact = fn: int -> int
> fact 5;
val it = 120: int
```

5.3 Implementazione in Haskell

Il Y Combinator è definito come

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Provando una semplice conversione nella sintassi Haskell ("λ" diventa "\" e "." diventa "`->`") si ottiene

```
y = \f -> (\x -> f (x x))(\x -> f (x x))
```

Questa espressione non è però accettata da un compilatore o interprete Haskell. Per esempio, `ghci` genera il seguente errore:

```
Prelude> y = \f -> (\x -> f (x x))(\x -> f (x x))

<interactive>:1:23: error:
```

[3]Questa funzione viene detta "chiusa" perché non ha variabili libere. La tradizionale definizione ricorsiva della funzione fattoriale usa invece una variabile libera (il suo nome) per richiamarsi ricorsivamente.

```
    o Occurs check: cannot construct the infinite type: t0 ~ t0 -> t
      Expected type: t0 -> t
        Actual type: (t0 -> t) -> t
    o In the first argument of 'x', namely 'x'
      In the first argument of 'f', namely '(x x)'
      In the expression: f (x x)
    o Relevant bindings include
        x :: (t0 -> t) -> t (bound at <interactive>:1:13)
        f :: t -> t (bound at <interactive>:1:6)
        y :: (t -> t) -> t (bound at <interactive>:1:1)
...
```

Il significato di questo errore diventa chiaro cercando di capire qual'è il tipo di "`x`". Supponendo che il tipo di "`x`" sia "`t0`", si ha che:

- Poiché "`(x x)`" indica che "`x`" è *una funzione* applicata ad un argomento, il tipo "`t0`" di "`x`" deve essere una funzione. Consideriamo il caso più generico: una funzione $\alpha \to \beta$, quindi $\texttt{t0} = \alpha \to \beta$
- D'altra parte, la funzione "`x`" è applicata proprio all'argomento "`x`". Quindi il tipo α dell'argomento della funzione "`x`" deve essere uguale al tipo di "`x`" ("`t0`"): $\texttt{t0} = \alpha$.

Mettendo tutto assieme si ottiene

$$\texttt{t0} = \alpha \to \beta \wedge \texttt{t0} = \alpha \Rightarrow \alpha = \alpha \to \beta$$

che definisce il tipo "`t0`" in modo ricorsivo (come indicato dal messaggio di errore, "`t0`" deve essere equivalente a "`t0 -> t`"). Poiché Haskell supporta tipi di dati ricorsivi, verrebbe quindi da pensare che dovrebbe essere in grado di supportare il tipo "`t0`" di cui stiamo parlando. Invece, una ricorsione come $\alpha = \alpha \to \beta$ non ha molto senso in un linguaggio che fa un controllo stretto dei tipi come Haskell. Per capire come mai, è necessario entrare un po' più nei dettagli dei meccanismi di definizione di tipi ricorsivi.

Di fronte ad una definizione come $\alpha = \alpha \to \beta$, si parla di tipi *equi-ricorsivi* (equi-recursive data types) se α ed $\alpha \to \beta$ rappresentano lo stesso tipo di dati (stessi valori, stesse operazioni sui valori, etc...). Si parla invece di tipi *iso-ricorsivi* (iso-recursive data types) se il tipo α ed il tipo $\alpha \to \beta$ non sono uguali, ma esiste un isomorfismo (funzione $1 \to 1$, invertibile, che ad ogni valore di α associa un valore di $\alpha \to \beta$ e viceversa) dall'uno all'altro. Questo isomorfismo (che stabilisce che i due tipi sono in sostanza equivalenti) è la funzione che in abbiamo chiamato "costruttore" parlando di tipi di dati algebrici. Capendo che Haskell supporta quindi tipi iso-ricorsivi (un valore di α può essere generato tramite un apposito costruttore a partire da un valore di $\alpha \to \beta$), ma non equi-ricorsivi (α ed $\alpha \to \beta$ **non** possono essere lo stesso tipo!) diventa allora chiaro perché la definizione di `Y` data poco fa generi un errore sintattico (in particolare, un errore di tipo).

Si noti che questo non significa che la definizione dell'Y combinator (Equazione 5.2) sia "sbagliata", ma semplicemente che non è implementabile direttamente in Haskell (come in ogni altro linguaggio che utilizzi una tipizzazione forte). L'Y combinator è definito usando il λ-calcolo, in cui ogni identificatore è legato ad una funzione, il cui tipo non è importante. Utilizzando linguaggi con controlli sui tipi "meno stretti" rispetto a Haskell (per esempio, Lisp, Scheme, Python, Javascript, etc...) o linguaggi che supportano tipi equi-ricorsivi (per esempio, OCaml con apposite opzioni), l'Equazione 5.2 è implementabile senza problemi.

Per capire meglio come risolvere questo problema, consideriamo la "parte problematica" dell'espressione precedente (l'applicazione di funzione "`(x x)`"), limitandoci per il momento alla funzione (chiaramente di ordine superiore) "`\x -> (x x)`".

Poiché in Haskell un tipo di dato ricorsivo `t0` può essere definito usando il costrutto `data` ed (almeno) un costruttore che mappa valori di un tipo che dipende da `t0` in valori di `t0` (definizione di tipo iso-ricorsivo), si può definire qualcosa dei tipo $T = F(T \to \beta)$ per "simulare" il tipo (equi-ricorsivo) $\alpha = \alpha \to \beta$ di `x`. Il tipo `T` risultante sarà quindi funzione del tipo β ed in Haskell questo si indica con "`T b`".

Ricordandosi la sintassi di `data` (un tipo di dato dipendente dalla type variable β in Haskell si definisce usando `data <nometipo> b = ...`), il tipo di dato che ci interessa è

```
data T b = F (T b -> b)
```

dove come detto "`T b`" è il nome del tipo, mentre "`F`" è il nome del costruttore che mappa valori di `T b -> b` in valori di `T b`.

A questo punto, è possibile usare "T b" per tipizzare correttamente "x" facendo si che "x" sia una funzione T b -> b da T b a b. L'argomento (parametro attuale) di tale funzione deve quindi essere di tipo T b, generabile da "x" usando il costruttore F. Invece di "\x —> x x" si scrive allora:

```
\x -> (x (F x))
```

Ed Haskell è perfettamente in grado di capire e processare questa definizione. Si noti come l'impossibilità di usare tipi equi-ricorsivi ci ha obbligati ad usare il costruttore F. Il tipo della funzione definita qui sopra risulta essere (T b -> b) -> b (funzione che mappa valori "x" di tipo "T b-> b" in valori di tipo "b").

Ora che abbiamo visto come risolvere il problema del tipo di "x", è possibile tornare all'espressione originaria del Y combinator, che presenta ancora un problema analogo: "\x -> f(x (F x))" va applicata a se stessa, quindi il suo tipo è ricorsivo come il tipo di "x"! Ancora una volta il problema può essere risolto usando il tipo di dato "T b" precedentemente definito: "\f —> (\x —> f (x (F x)))" ha tipo "(b -> c) -> (T b -> b) -> c", quindi assumendo "b -> c" come tipo per "f" si ha che "\x -> f (x (F x))" ha tipo "(T b -> b) -> c"... Il valore "\x -> f (x (F x))" a cui è applicata deve quindi avere tipo "Tb -> b". Bisogna quindi sostituire un "(T b -> b)" (tipo di "x") con un "T b" (ottenibile da "x" applicandogli il costruttore "F"). L'argomento sarà quindi "\f —> (\ (F x) —> f (x (F x)))"

Come risultato, l'espressione dell'Y combinator dovrebbe essere:

```
y = \f -> (\x -> f(x (F x)))(\ (F x) -> f(x (F x)))
```

ed in teoria questa espressione dovrebbe essere accettata da Haskell! Sfortunatamente, però, alcune versioni di ghc (e quindi di ghci) hanno dei problemi ad inferire correttamente i tipi di dato[4]. Altri programmi, come l'interprete Hugs (https://www.haskell.org/hugs) riescono a parsare e valutare questa definizione senza problemi.

Il problema incontrato da alcune versioni di ghc può essere superato "aiutando" in qualche modo il compilatore ad inferire correttamente i tipi delle varie sottoespressioni. Per esempio, si potrebbe sostituire "F x" con una variabile "z" del giusto tipo (T b). Per fare questo, però, è necessaria una funzione che permetta di estrarre "x" da "F x"; in pratica, la funzione inversa del costruttore "F":

```
invF (F x) = x
```

A questo punto, è possibile sostituire "\(F x)" con "\z" e la seguente "x" con "invF z", ottenendo

```
y = \f -> (\x -> f(x (F x)))(\z -> f((invF z) z))
```

e questa espressione è accettata anche da ghc/ghci! Riassumendo, la prima parte dell'espressione ("\x -> f(x (F x))") ha tipo "(T b -> b) -> c", mentre la seconda parte ("\z -> f((invF z) z)") ha tipo "T b -> c" ed è quindi utilizzabile come argomento per la prima semplicemente ponendo $\beta = \gamma$.

Il tipo della funzione risultante è "(b -> b) -> b", dove "b" è chiaramente un tipo funzione (per il fattoriale, per esempio, è una funzione "Int -> Int").

La discussione di qui sopra, che mostra come sia possibile implementare il fixed point combinator Y in linguaggi (come Haskell) con tipizzazione stretta che non supportano tipi di dati equi-ricorsivi, ci permette di intuire una cosa importante: Y "elimina la ricorsione" dall'operatore fixed point (l'Equazione 5.1 utilizza ricorsione esplicita) spostando la ricorsione sul tipo di dato (richiede infatti di usare tipi di dati ricorsivi di qualche genere). Questo non è immediatamente visibile nell'Equazione 5.2 o in implementazioni di Y in scheme (o simili), ma diventa più chiaro provando ad implementare Y in (per esempio) Haskell.

Una volta che Haskell ha "accettato e capito" la nostra definizione di Y, si può, per esempio, utilizzare la funzione Y appena definita per definire la funzione fattoriale. Prima di tutto, definiamo una versione "chiusa" fact_closed[5] della funzione fattoriale come

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

A questo punto, si può calcolare la funzione fattoriale come

```
fact = y fact_closed
```

[4] Questo è probabilmente legato ad un bug descritto in https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/bugs.html.

[5] Questa funzione viene detta "chiusa" perché non ha variabili libere. La tradizionale definizione ricorsiva della funzione fattoriale usa invece una variabile libera (il suo nome) per richiamarsi ricorsivamente.

Si noti che il tipo di "`fact_closed`" è "`(Eq p, Num p) => (p -> p) -> p -> p`", quindi, poiché Y ha tipo "`(b -> b) -> b`" "`y fact_closed`" ha tipo "`(Eq p, Num p) => p -> p`".

Riassumendo, una possibile definizione del Y combinator in Haskell (ne esistono molte altre) può essere:

```
data T b = F (T b -> b)
invF (F x) = x
y = \f -> (\x -> f(x (F x)))(\y -> f((invF y) y))
```

Basandosi su questa definizione si può, per esempio, usare `ghci` per fare:

```
Prelude> fact_closed = \f -> \n -> if n == 0 then 1 else n * f (n - 1)
Prelude> :t fact_closed
fact_closed :: (Eq p, Num p) => (p -> p) -> p -> p
Prelude> fact = y fact_closed
Prelude> :t y
y :: (b -> b) -> b
Prelude> :t fact
fact :: (Eq p, Num p) => p -> p
Prelude> fact 3
6
Prelude> fact 4
24
...
```

5.4 Come Derivare l'Y Combinator

Come detto nelle sezioni precedenti, l'Y combinator è definito come $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$; fino ad ora non è però chiaro da cosa derivi tale espressione.

Si consideri una generica funzione ricorsiva f (definita per semplicità sull'insieme dei numeri naturali $\mathcal{N}$) il cui passo induttivo è definito come $f(n+1) = g(f(n), n)$. La classica implementazione ricorsiva di tale funzione andraà ad invocare ricorsivamente $f()$ nel passo induttivo, risultando quindi essere una funzione non chiusa (si ricordi che una funzione si definisce chiusa quando non usa variabili esterne).

Per esempio, si consideri la funzione fattoriale, definita (usando per semplicità un λ-calcolo esteso, che permette di usare numeri, operazioni aritmetiche ed if logici [6]) come

$$f = \lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)$$

Si può immediatamente vedere come l'espressione "if $n = 0$ then 1 else $n \cdot f(n-1)$" utilizzi la variabile libera (non legata) f.

Una generica funzione ricorsiva $f(n+1) = g(f(n), n)$ può essere descritta tramite una sua "versione chiusa" f_{closed} che non usa variabili libere, ma riceve come parametro la funzione da chiamare ricorsivamente. Tale funzione può essere definita come $f_{closed} = \lambda f.\lambda n. < expr >$ (dove "$< expr >$ è un'espressione chiusa, in quanto la "f" è stata legata da un "$\lambda f.$"). Come noto, la funzione ricorsiva originaria "f" è un punto fisso della funzione chiusa f_{closed}: $f = f_{closed} f$ ed assumendo di disporre di un fixed point combinator Y si può calcolare f come

$$f = Y f_{closed}$$

(in altre parole, la funzione chiusa f_{closed} è una versione non ricorsiva di f, che riceve come primo argomento la funzione f da invocare ricorsivamente: $f(n) = (f_{closed} f)n \Leftrightarrow f = f_{closed} f$).

Tornando all'esempio della funzione fattoriale di qui sopra, la sua "versione chiusa" f_{closed} è ovviamente

$$f_{closed} = \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1).$$

Si noti che la definizione di f_{closed} è un primo passo verso l'eliminazione della ricorsione, ma ancora non è sufficiente: f_{closed} riceve infatti come primo argomento la funzione f che stiamo cercando di definire.

[6] Si ricordi che anche se numeri, operazioni aritmetiche, if logici e costrutti simili non fanno tecnicamente parte del λ-calcolo puro, possono essere implementati usando λ-espressioni pure.

Si può allora definire una funzione G che come f_{closed} riceve come primo argomento una funzione da invocare ricorsivamente, ma a differenza di f_{closed} invoca tale funzione passando la funzione stessa come primo argomento. In altre parole, mentre il parametro formale di f_{closed} è una funzione $f : \mathcal{N} \to \mathcal{N}$, il primo parametro di G è una funzione che riceverà 2 argomenti: funzione da richiamare ricorsivamente e numero intero. G si può quindi ottenere da f_{closed} semplicemente sostituendo tutte le invocazioni di f con ff. Tornando all'esempio del fattoriale, si ottiene:

$$G = \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot ff(n-1). \tag{5.3}$$

Riassumendo, f_{closed} è tale che $f = f_{closed}f$, mentre G è tale che $f = GG$!!! I lettori più attenti si saranno sicuramente accorti del fatto che questo è il momento in cui si passa dalla "ricorsione sul flusso di esecuzione" (definizione ricorsiva di f, come funzione non chiusa) alla ricorsione sul tipo di dati (definizione di una funzione G che è chiusa, ma ha necessità di potersi applicare a se stessa - self application). Per capire meglio la cosa, si provi a trovare il tipo della funzione G...

Con un minimo di manipolazione (una "β-riduzione al contrario") si ottiene che G è equivalente a $\lambda f.(f_{closed}(ff))$: infatti, come detto il corpo di G si ottiene dal corpo di f_{closed} sostituendo f con ff (vale a dire, applicando f_{closed} ad ff). Si noti come l'utilizzo di questa "β-riduzione al contrario" implica che (come già noto) il Y combinator che stiamo andando a ricavare lavora per β-equivalenza e non per β-riduzione (in altre parole: $Yf_{closed} \equiv_\beta f_{closed}(Yf_{closed})$, ma $Yf_{closed} \not\to_\beta f_{closed}(Yf_{closed})$).

Tornando all'esempio del fattoriale, si ha

$$G \equiv_\beta \lambda f.(\lambda h.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot h(n-1))(ff)$$

in quanto se si applica $\lambda h.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot h(n-1)$ a (ff) si ottiene l'espressione di G definita in Equazione 5.3. Ora, si può vedere come questa espressione contenga al suo interno la definizione di f_{closed}: l'espressione $\lambda h.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot h(n-1)$ dentro le parentesi che viene applicata ad (ff) è proprio f_{closed}! Riassumendo,

$$G \equiv_\beta \lambda f.(f_{closed}(ff)).$$

Ricordando ora che $f = GG$, ed applicando un'altra "β-riduzione al contrario" si ha $f = GG \leftarrow_\beta (\lambda h.(hh))G$; sostituendo a questo punto G $\lambda f.f_{closed}(ff)$ (che è β equivalente a G) si ottiene

$$f \equiv_\beta (\lambda h.(hh))(\lambda f.f_{closed}(ff))$$

La solita β-riduzione al contrario (astraendo rispetto ad funzione g ed applicando ad f_{closed} in modo da ottenere l'espressione di qui sopra) ci porta a

$$f \equiv_\beta (\lambda g.(\lambda h.(hh))(\lambda f.g(ff)))f_{closed}$$

ed una semplice α-equivalenza porta a

$$f \equiv_\beta (\lambda g.(\lambda h.(hh))(\lambda h.g(hh)))f_{closed}$$

A questo punto, ricordando che per definizione $f \equiv_\beta Yf_{closed}$ si ottiene $Y = \lambda g.(\lambda h.(hh))(\lambda h.g(hh))$!!!

Si noti che l'espressione di Y non è ancora quella vista in precedenza, ma la definizione "tradizionale" si può ottenere tramite un passo di β-riduzione: applicando $\lambda h.(hh)$ a $\lambda h.g(hh)$ si ottiene finalmente $Y = \lambda g.(\lambda h.g(hh))(\lambda h.g(hh))$.

Per capire come si deriva lo Z combinator (che possiamo vedere come una versione dell'Y combinator che funziona con valutazione eager), si consideri il passaggio da

$$G = \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot ff(n-1).$$

a

$$\lambda f.(\lambda h.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot h(n-1)(ff)).$$

Il problema sorge quando "f" è sostituita con una funzione "$\lambda g....$": mentre la prima espressione può essere valutata senza divergere anche da linguaggi eager (perché il redex "$(\lambda g....)(\lambda g....)$" che si viene a creare sta dentro ad un'astrazione "$\lambda n....$"), la seconda cerca di valutare il redex che costituisce l'argomento "$(\lambda g....)(\lambda g....)$" (che non si trova più "protetto") da un'astrazione "$\lambda h....$") prima di applicargli la funzione f_{closed} ("$\lambda h.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot h(n-1)$" nel caso del fattoriale). Ma la riduzione

di questo redex "$(\lambda g....)(\lambda g....)$" non convergerà (perché l'argomento "g" sarà espanso a qualcosa che richiama la funzione stessa), portando ad una ricorsione infinita...

Per evitare questa ricorsione infinita bisogna evitare che la macchina astratta cerchi di ridurre "(ff)" in ogni caso (ma vada a valutare tale espressione solo quando necessario). Questo risultato può essere ottenuto "proteggendo" il termine "(ff)" tramite un'astrazione rispetto ad un argomento v ed applicando poi il risultato a v: $(ff) \to \lambda v.(ff)v$. Da cui si ha:

$$\lambda f.(\lambda h.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot h(n-1))(\lambda v.(ff)v).$$

Ripercorrendo i passi precedenti, questo porta a

$$G \equiv_\beta \lambda f.f_{closed}(\lambda v.(ff)v)$$

e quindi

$$f \equiv_\beta (\lambda h.(hh))(\lambda h.f_{closed}(\lambda v.(hh)v))$$

da cui

$$Z = \lambda f.(\lambda h.(hh))(\lambda h.f_{closed}(\lambda v.(hh)v)).$$

Con un passo di β-riduzione si ottiene

$$Z = \lambda f.(\lambda h.g(\lambda v.(hh)v))(\lambda h.g(\lambda v.(hh)v)).$$

che è l'espressione più tradizionalmente conosciuta.

Appendice A

Un po' di Definizioni

A.1 Identificatori, Legami ed Ambiente

Una caratteristica di quasi tutti i linguaggi di programmazione (da quelli di basso livello come l'Assembly a quelli di più alto livello) è la possibilità di associare nomi alle "entità" che compongono i programmi (siano esse semplicemente valori, locazioni di memoria, variabili, funzioni o altro...).

Più formalmente, ogni linguaggio di programmazione è composto da alcune *entità denotabili*, che è possibile identificare tramite nomi (identificatori) più facilmente gestibili dal programmatore. Esiste quindi una funzione, chiamata *ambiente*, che ha come dominio l'insieme degli identificatori e come codominio l'insieme delle entità denotabili del linguaggio. Tale funzione cambia nel tempo (è possibile creare o distruggere dinamicamente dei legami fra identificatori ed entità denotabili durante l'esecuzione del programma) e nei vari punti del programma (alcuni legami possono essere validi in alcune zone del codice ma non in altre)[1]. Più formalmente, si può quindi dire che:

Definizione 1 (Legame) *Si definisce legame (binding) fra un identificatore I ed un'entità denotabile E una coppia (I, E) che associa l'entità al nome.*

Definizione 2 (Ambiente) *Si definisce ambiente l'insieme di legami esistenti in uno specifico momento dell'esecuzione di un programma (mentre si esegue codice in uno specifico punto del programma).*

(questa definizione di ambiente non deve sorprendere, ricordando che dal punto di vista matematico una funzione è un insieme di coppie - un sottoinsieme del prodotto cartesiano di dominio e codominio).

Come detto, l'ambiente contiene i legami fra identificatori simbolici e generiche "entità denotabili", che dipendono dal linguaggio. Per esempio,

- nel linguaggio Assembly gli identificatori sono associabili solo ad indirizzi di memoria
- in un linguaggio imperativo di più alto livello, gli identificatori possono essere associati a valori, variabili, funzioni o tipi
- nel λ-calcolo, gli identificatori sono associabili a funzioni
- in un linguaggio funzionale, gli identificatori sono associabili solo a valori (in un linguaggio funzionale, una funzione è in effetti un valore)

In generale, quando un linguaggio prevede il concetto di blocco di codice è possibile distinguere i concetti di ambiente locale, ambiente non locale ed ambiente globale.

Definizione 3 (Ambiente Locale) *Si definisce ambiente locale di un blocco di codice il sottoinsieme dell'ambiente composto da legami che sono stati creati nel blocco di codice in questione (e che sono quindi validi solo all'interno di tale blocco di codice).*

Definizione 4 (Ambiente non Locale) *Si definisce ambiente non locale di un blocco di codice il sottoinsieme dell'ambiente che non è composto dall'ambiente locale del blocco in questione (contiene quindi legami che sono stati creati fuori dal blocco e rimarranno validi quando l'esecuzione uscirà dal blocco di codice).*

[1] L'ambiente è modificato, per esempio, da una dichiarazione, o dal primo utilizzo di un'entità denotabile (per linguaggi in cui non è necessario dichiarare un'entità prima di utilizzarla).

Definizione 5 (Ambiente Globale) *Si definisce ambiente globale il sottoinsieme dell'ambiente composto da legami che non sono stati creati all'interno di alcun blocco di codice (o, sono stati creati nel blocco di codice più esterno, che contiene tutti gli altri blocchi).*

A.2 Variabili Modificabili

Nei linguaggi imperativi la computazione procede attraverso la modifica di valori memorizzati in locazioni di memoria, che nei linguaggi di alto livello sono identificate da variabili.

Definizione 6 (Variabile Modificabile) *Una variabile modificabile è un tipo di entità denotabile che rappresenta una zona di memoria che può contenere entità memorizzabili.*

Nella precedente definizione si utilizza il concetto di "entità memorizzabile", il cui significato dipende ancora una volta dal linguaggio, ma che sta ad indicare le entità che possono essere conteneute in una variabile modificabile (per esempio, in un linguaggio funzionale una funzione è un'entità memorizzabile, mentre in alcuni linguaggi imperativi non lo è).

Dal punto di vista concettuale, la presenza di variabili modificabili introduce una nuova funzione, detta "store", che associa ad ogni variabile l'entità memorizzabile in essa contenuta.

Definizione 7 (store) *Si definisce store l'insieme di coppie* (V, E)*, dove* V *è una variabile ed* E *è un'entità memorizzabile che associano ad ogni variabile il suo contenuto.*

Lo store è quindi una funzione (che rappresenta la memoria utilizzabile dal nostro programma per i dati) avente come dominio l'insieme delle variabili e come codominio l'insieme delle entità memorizzabili.

Quando in un linguaggio imperativo si utilizza il valore contenuto nella variabile "`x`", si va in realtà ad applicare la funzione store al valore ritornato dalla funzione ambiente applicata all'identificatore "`x`": $store(env(\texttt{x}))$ (dove "env" è la funzione ambiente).

A.3 Entità Denotabili, Esprimibili e Memorizzabili

Si è visto come un programma sia composto (fra le altre cose) da generiche "entità" (la cui definizione dipende dal linguaggio di programmazione, ma che in generale possono essere tipi, valori, variabili, funzioni, etc...). Tali entità possono essere in generale *denotabili*, *memorizzabili* ed `esprimibili`.

Definizione 8 (Entità Denotabile) *Si definisce entità denotabile un'entità che può essere associata ad un nome / identificatore.*

Definizione 9 (Entità Memorizzabile) *Si definisce entità memorizzabile un'entità che può essere contenuta in una variabile (usando un linguaggio di programmazione imperativo)*

Definizione 10 (Entità Esprimibile) *Si definisce entità esprimibile un'entità che può essere generata come risultato di un'espressione*

Un'entità denotabile è quindi ogni entità che si può indicare con un nome (definito dall'utente o predefinito dal linguaggio) e l'insieme delle entità denotabili è il codominio della funzione ambiente. Un'entità esprimibile è invece qualsiasi entità che può "essere calcolata/allocata" in qualche modo usando i costrutti del linguaggio. Per finire, le entità memorizzabili (che sono una caratteristica dei linguaggi imperativi) costituiscono il codominio della funzione store.

In un linguaggio funzionale, tutte le entità sono denotabili ed esprimibili (non ha senso parlare di entità memorizzabili, in quanto non esiste la funzione store), mentre in un linguaggio di programmazione imperativo va fatta la distinzione fra varie tipologie di entità (possono esistere per esempio entità denotabili ma non esprimibili o memorizzabili, come le funzioni in alcuni linguaggi imperativi).

Si noti che in letteratura le definizioni di denotabile, esprimibile e memorizzabile si associano talvolta solo ai valori.

```
void wrong_swap(int a, int b)
{
  int tmp;

  tmp = a; a = b; b = tmp;
}

void correct_swap(int *a, int *b)
{
  int tmp;

  tmp = *a; *a = *b; *b = tmp;
}
```

Figura A.1: Esempio di funzione che scambia il contenuto di due variabili, implementata in C. Visto che i parametri vengono passati per valore, la funzione "`wrong_swap()`" non sortirà nessun effetto; la funzione "`correct_swap()`", invece, ricevendo come parametri i *puntatori* alle variabili da scambiare, potà funzionare correttamente.

A.4 Funzioni

Una delle caratteristiche fondamentali dei linguaggi di programmazione di alto livello è quella di permettere di modularizzare il codice, scomponendo il programma in una serie di componenti (sottoprogrammi, subroutine) ognuno dei quali implementa una specifica funzionalità, secondo un'interfaccia ben definita.

Ognuno di questi sottprogrammi è quindi un'entità che rappresenta una parte auto-contenuta del codice che può essere invocata passando dei parametri e ricevento eventualmente in ritorno dei valori. Generalmente, un sottoprogramma che può ritornare un valore viene definito funzione (anche se è una cosa molto differente da una funzione matematica - può avere effetti collaterali!!!) e nella nomenclatura informatica moderna si tende a parlare di funzione anche per sottoprogrammi che non hanno valori di ritorno (perché dal punto di vista concettuale è come se ritornassero un valore di tipo `void`, o `unit`).

Definizione 11 (Funzione) *Si definisce funzione un'entità denotabile composta da un blocco di codice a cui può essere associato un nome utilizzabile per invocarne l'esecuzione. Nel momento in cui viene invocata l'esecuzione di una funzione il codice chiamante può scambiare dei dati con la funzione tramite i suoi parametri, il valore di ritorno o lo stato globale del programma.*

Definizione 12 (Parametro Formale) *Un parametro formale di una funzione, definito nella definizione della funzione, identifica una variabile che può essere usata dal codice chiamante per passare dei dati ad una funzione quando la invoca.*

Definizione 13 (Parametro Attuale) *Un parametro attuale è un'espressione, specificata nel momento in cui si invoca una funzione, che verrà associata al corrispondente parametro formale durante l'esecuzione della funzione.*

Il fatto che una funzione sia definita come un'entità denotabile chiarisce subito che al blocco di codice è possibile (anche se non sempre obbligatorio: esistono anche funzioni anonime!) associare un nome. Inoltre, il fatto che la funzione sia un blocco di codice chiarisce che la funzione è caratterizzata da un ambiente locale (legami fra variabili locali alla funzione e loro nomi, legami fra parametri formali e parametri attuali, etc...).

Il modo in cui i parametri attuali vengono associati ai parametri attuali dipende dal meccanismo di invocazione della funzione e dalla modalità di passaggio dei parametri utilizzata. Per esempio, come detto un parametro formale identifica una variabile; tale variabile può essere creata nel momento in cui si invoca la funzione e distrutta quando l'esecuzione della funzione termina (avendo quindi un tempo di vita che coincide col tempo di esecuzione della funzione), oppure può essere pre-esistente all'esecuzione della funzione (rimanendo in vita quando l'esecuzione della funzione è terminata). Esistono quindi diverse modalità di passaggio dei parametri (dipendenti dal particolare meccanismo di applicazione di funzione che viene utilizzato), le più importanti dei quali sono il passaggio di parametri per valore, il passaggio di parametri per riferimento ed il passaggio di parametri per nome.

```
void reference_swap(int &a, int &b)
{
  int tmp;

  tmp = a; a = b; b = tmp;
}
```

Figura A.2: Esempio di funzione che scambia il contenuto di due variabili, implementata in C++ tramite passaggio di parametri per riferimento.

```
int f(int v)
{
  int a = 666;

  return a + v;
}
```

Figura A.3: Esempio di funzione per il passaggio di parametri per nome.

Nel caso di passaggio parametri *per valore*, una variabile locale viene allocata quando la funzione è invocata e deallocata quando l'esecuzione della funzione termina (questo è facilmente fattibile allocando la variabile sullo stack, nel record di attivazione dell'invocazione di funzione). L'ambiente locale della funzione contiene quindi un binding fra il nome del parametro formale e questa variabile, il cui contenuto viene inizializzato col valore ottenuto valutando l'espressione passata come parametro attuale al momento dell'invocazione (da qui il nome "passaggio di parametri per valore"). Un'importante proprietà del passaggio di parametri per valore è che eventuali modifiche al valore del parametro effettuate durante l'esecuzione della funzione vengono "perse" quando l'esecuzione termina (in altre parole, il passaggio di parametri per valore permette di passare dati dal chiamante alla funzione, ma non viceversa). Questo è l'unico tipo di passaggio di parametri supportato dal linguaggio C; si noti infatti che quando una funzione C deve modificare il valore di una variabile passata come parametro attuale, il "vero parametro" non può essere la variabile, ma deve essere un puntatore ad essa. In altre parole, la funzione "`wrong_swap()`" di figura A.1 non può realmente scambiare il contenuto delle variabili passate come parametri attuali (la funzione "`correct_swap()`", invece, si comporterà in modo corretto).

Nel caso di passaggio di parametri *per riferimento*, invece, all'invocazione della funzione non viene creata alcuna nuova variabile per i parametri formali. L'ambiente locale della funzione è semplicemente modificato aggiungendo un binding fra il nome del parametro formale ed il parametro attuale. Questo significa che i parametri attuali passati per riferimento non possono essere generiche espressioni, ma devono essere variabili, o comunque entità denotabili che possano stare a sinistra dell'operatore di assegnamento (L-value, usando la nomenclatura C/C++ — la "*L*" sta per "*Left*", a *sinistra* dell'operatore). Questo tipo di passaggio di parametri non è (come già accennato) supportato dal linguaggio C, ma è supportato dal linguaggio C++; si veda per esempio la funzione `reference_swap()` di Figura A.2 (confrontando questo codice col codice mostrato in Figura A.1 si possono capire meglio le differenze fra il passaggio di parametri per valore ed il passaggio di parametri per riferimento).

Nel caso di passaggio di parametri *per nome*, infine, all'invocazione della funzione il parametro formale viene sostituito (tramite rimpiazzamento testuale) dall'espressione utilizzata come parametro attuale. Questo è il meccanismo tipicamente utilizzato nella valutazione di programmi funzionali. Sebbene questo meccanismo di passaggio dei parametri possa a prima vista sembrare semplice da implementare, può presentare alcuni subdoli problemi. Per esempio, si consideri la funzione "`f()`" di Figura A.3, che (a parte l'opinabile utilizzo della variabile locale "`a`") ha l'evidente obiettivo di sommare 666 al valore ricevuto in ingresso. In effetti, l'invocazione "`f(n)`" viene valutata per nome come "{ **int** a = 666; **return** a + n;}" che diventa "{ **return** 666 + n;}" ed il valore di ritorno è quindi "`666 + n`". Ma come deve essere valutata l'invocazione "`f(a)`", dove "`a`" è una variabile non locale di "`f()`" (per esempio, una variabile globale)? Sostituendo semplicemente "`v`" con "`a`" nel corpo della funzione, si otterrebbe "{ **int** a = 666; **return** a + a;}" che ritorna "`666 + 666`" (e quindi il valore `1332`). Questo non è ovviamente il risultato atteso. Il problema è ovviamente dovuto al fatto che sostituendo semplicemente "`v`" con "`a`" non si è più in grado di distinguere le due diverse variabili (una locale alla funzione

ed una non locale) che hanno lo stesso nome "a". Questo fenomeno è talvolta noto come "cattura di una variabile libera".

Questo problema viene generalmente risolto nel campo della programmazione funzionale tramite un semplice cambio di nome delle variabili: se la funzione "f()" viene invocata usando come parametro attuale un'espressione che contiene una variabile chiamata "a", allora la variabile locale "a" deve essere rinominata (per esempio in "a1") in modo che la sostituzione risulti in "{ **int** a1 = 666; **return** a1 + a;}" che ritorna correttamente "666 + a". Dal punto di vista implementativo, invece, il passaggio di parametri per nome può essere realizzato passando un "*thunk*" (vale a dire una coppia ambiente, espressione) al posto del parametro. Quindi, una funzione che riceve parametri passati per nome è implementabile come una funzione che riceve come parametri delle coppie composte da espressioni (i parametri attuali) e dagli ambienti in cui tali espressioni vanno valutate.

Come detto, una funzione che non ha valori di ritorno è modellabile come una funzione per cui il tipo del valore di ritorno ha un unico valore possibile (tipo unit, o void). Una considerazione analoga vale anche per i parametri.

A.5 Chiusure

```
void->int contatore(void)
{
  int n = 0;

  int f(void) {
    return n++;
  }

  return f;
}
```

Figura A.4: Esempio di funzione che ritorna una chiusura.

In alcuni linguaggi le funzioni sono considerate entità memorizzabili (esistono variabili che possono memorizzare funzioni) o esprimibili (esistono funzioni che possono ritornare funzioni come valore di ritorno) o possono essere usate come parametri per altre funzioni. In questo caso i valori memorizzati, ritornati o passati come parametri non possono essere semplicemente funzioni, ma devono essere *chiusure*.

Definizione 14 (Chiusura) *Si definisce chiusura una coppia composta da una funzione e dal suo ambiente non locale.*

Sostanzialmente, una chiusura serve per sapere a quali entità denotabili sono associati gli identificatori per cui non esiste un legame nell'ambiente locale della funzione.

Si consideri per esempio la funzione void->int contatore(void) descritta in Figura A.4, scritta in uno pseudo-linguaggio simile al C in cui "void->int" indica il tipo delle funzioni che non hanno argomenti e ritornano un valore di tipo int. La funzione contatore() non riceve argomenti e ritorna una funzione che ritorna progressivamente tutti i numeri interi da 0 in poi. Come si può vedere, "n" è una variabile locale della funzione contatore() e non una variabe o argomento della funzione "f()" che viene ritornata. Quindi, nel momento in cui viene invocata (per esempio) "prossimo = contatore()" nell'ambiente locale di contatore() esiste un legame fra l'identificatore "n" ed una variabile che contiene inizialmente il valore "0" (e viene incrementata da f() ogni volta che la si invoca). Ma tale legame non è nell'ambiente locale di f() (è nel suo ambiente non locale). Quando quindi contatore() termina ed il suo ambiente locale viene distrutto, non esiste più tale legame e non è chiaro come l'identificatore "n" debba essere risolto se invoco "prossimo()". Peggio, anche la variabile legata ad "n" viene distrutta quando contatore() termina. Per rendere quindi utilizzabile questo codice, devono essere fatte 2 cose distinte:

1. La variabile legata all'identificatore "n", che contiene valore iniziale "0" non deve essere distrutta quando contatore() termina. Questo significa che la variabile non deve essere allocata sullo stack, ma nello heap

```
#include <stdio.h>

int (*counter(void))(void)
{
  int n = 0;

  int f(void) {
    return n++;
  }

  return f;
}

int main()
{
  int (*c1)(void) = counter();
  int (*c2)(void) = counter();
  int (*c3)(void) = counter();

  printf("   %d %d\n", c1(), c1());
  printf("%d %d %d\n", c2(), c2(), c2());
  printf("   %d %d\n", c3(), c3());

  return 0;
}
```

Figura A.5: Esempio che mostra che i puntatori a funzione del C non implementa una chiusura (l'esempio utilizza un'estensione non-standard del compilatore `gcc`).

2. Il legame fra "`n`" e tale variabile non deve essere creato solo nell'ambiente locale di `contatore()`, ma va copiato in un apposito ambiente che farà parte della chiusura ritornata da `contatore()` (e memorizzata in "`prossimo`")

Per capire meglio la differenza fra una chiusura ed una funzione (o un puntatore a funzione, come usato dal linguaggio C), si consideri il programma di Figura A.5, che utilizza un'estensione supportata dal compilatore `gcc` per implementare in C la funzione della Figura A.4. In questo caso la variabile "`int n`" viene distrutta quando la funzione "`contatore()`" ritorna, quindi il comportamento del programma è indefinito (come facilmente verificabile compilando il programma con `gcc`, eventualmente con vari livelli di ottimizzazione.

A.6 Chiusure e Classi

Il concetto di chiusura, che permette di associare un ambiente non locale ad una funzione, è stato originariamente introdotto per supportare funzioni di ordine superiore (funzioni che ritornano altre funzioni come risultato, o che le accettano come argomento). Ma le chiusure sono costrutti che vanno oltre a questo e permettendo di associare dati (lo stato contenuto nell'ambiente non locale) a codice (il codice che implementa il corpo della funzione) presentano interessanti similitudini e legami col concetto di oggetto.

Per capire meglio la relazione fra chiusure e classi, si può considerare l'implementazione in C++ di un contatore come chiusura e confrontarla con un'implementazione "più tradizionale" di contatore come classe. La Figura A.6 mostra come utilizzare il costrutto delle lambda expression C++ (che sono sostanzialmente delle funzioni anonime associate ad un ambiente non locale, quindi implementano il concetto di chiusura) per realizzare un contatore, mentre la Figura A.7 rappresenta un contatore implementato come classe (qui l'unica cosa "particolare" è la ridefinizione dell'operatore "`()`"). Come si può vedere confrontando le figure, la variabile intera "`n`" in un caso è una variabile locale della funzione "`contatore()`", che viene poi incapsulata nell'ambiente non locale della funzione ritornata, mentre nell'altro caso è un membro dati privato della classe.

```
#include <functional>
#include <iostream>

auto counter(void)
{
  int n = 0;

  return [n](void) mutable {
    return n++;
  };
}

int main()
{
  auto c1 = counter();
  auto c2 = counter();
  auto c3 = counter();

  std::cout << c1() << " " << c1() << std::endl;
  std::cout << c2() << " " << c2() << " " << c2() << std::endl;
  std::cout << c3() << " " << c3() << std::endl;

  return 0;
}
```

Figura A.6: Esempio che mostra come le lambda expression del C++ implementano il concetto di chiusura.

```
#include <iostream>

class Counter {
  private:
    int n;
  public:
    Counter(void) : n(0) {
    }
    int operator() (void) {
      return n++;
    }
};

int main()
{
  auto c1 = Counter();
  auto c2 = Counter();
  auto c3 = Counter();

  std::cout << c1() << " " << c1() << std::endl;
  std::cout << c2() << " " << c2() << " " << c2() << std::endl;
  std::cout << c3() << " " << c3() << std::endl;

  return 0;
}
```

Figura A.7: Esempio di contatore implementato come classe C++ (da confrontarsi con l'implementazione basata su chiusure).

Sostanzialmente quindi sia chiusure che classi permettono di affrontare il problema dell'accesso incontrollato ai dati, anche se lo fanno in modo diverso: le classi permettono di far si che solo il codice "autorizzato" (i metodi della classe) possa accedere allo stato incapsulato nella classe, mentre le chiusure permettono di far si che lo stato accessibile da una funzione stia nell'ambiente contenuto nella sua chiusura e sia quindi inaccessible da altre funzioni perché non referenziabile in esse.

Sebbene questo esempio (progettato per evidenziare la relazione fra chiusure e classi/oggetti) sembri mostrare che una chiusura possa servire semplicemente ad "associare uno stato" ad una funzione, esistono esempi forse più interessanti riguardo all'utilità delle chiusure nella programmazione funzionale: si pensi alla versione curryificata di una funzione che somma due numeri, o ad una funzione che riceve come argomento una funzione che accede a variabili non locali.