Breve Introduzione alla Programmazione Funzionale per Programmatori Imperativi

Luca Abeni

October 20, 2020

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: 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 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 3.

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

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

  return a;
}
```

Figure 1: Algoritmo di Euclide implementato in C.

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

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

Figure 2: Algoritmo di Euclide implementato in modo ricorsivo.

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

Figure 3: Implementazione puramente funzionale dell'algoritmo di Euclide.

Confrontando l'implementazione di Figura 1 (che verrà chiamata "implementazione imperativa") con quella di Figura 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:

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

  acc = acc + 1;

  return v + acc;
}
```

Figure 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 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 $(3 + 1) * (2 + 5) = 28$.

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);
}
```

Figure 5: Implementazione ricorsiva della funzione fattoriale.

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

Figure 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 5. Una versione più propriamente funzionale (perché utilizza solo espressioni, e non comandi) del fattoriale è mostrata invece in Figura 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 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 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 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).

```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;
}
```

Figure 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);
}
```

Figure 8: Soluzione alternativa del problema delle torri di Hanoi.

Una possibile soluzione (purtroppo non leggibilissima a causa della sintassi del linguaggio C) è mostrata in Figura 9. In questa soluzione, la funzione di utilità `concat()` riceve in ingresso due stringhe

```c
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));
}
```

Figure 9: Soluzione funzionale del problema delle torri di Hanoi.

(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)

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

```
#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;
}
```

Figure 10: Soluzione funzionale in C++ del problema delle torri di Hanoi.

funzioni per la gestione delle stringhe) la Figura 10 mostra una reimplementazione in C++ (che fornisce un supporto per le stringhe molto più avanzato rispetto al C), che appare molto più pulita.

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].

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

[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.

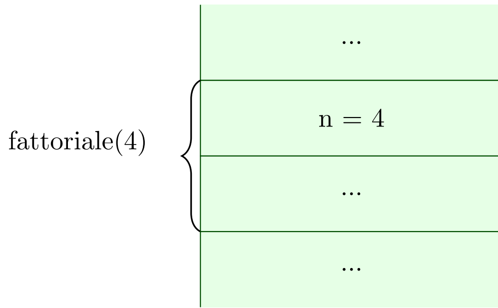

Figure 11: Stack frame per l'invocazione `fattoriale(4)`.

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 11.

Poiché anche "`fattoriale(3)`" si invocherà ricorsivamente, lo stack evolve come mostrato in Figura 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 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 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:

fattoriale (4) $\rightarrow$ fattoriale_tr $(4,\ 1) \rightarrow (4 == 0)\ ?\ 1 :$ fattoriale_tr $(4\ -\ 1, 4 * 1) \rightarrow$
$\rightarrow$ fattoriale_tr $(3,\ 4) \rightarrow (3 == 0)\ ?\ 4 :$ fattoriale_tr $(3\ -\ 1, 3 * 4) \rightarrow$
$\rightarrow$ fattoriale_tr $(2,\ 12) \rightarrow (2 == 0)\ ?\ 12 :$ fattoriale_tr $(2\ -\ 1, 2 * 12) \rightarrow$
$\rightarrow$ fattoriale_tr $(1,\ 24) \rightarrow (1 == 0)\ ?\ 24 :$ fattoriale_tr $(1\ -\ 1, 1 * 24) \rightarrow$
$\rightarrow$ fattoriale_tr $(0,\ 24) \rightarrow (0 == 0)\ ?\ 24 :$ fattoriale_tr $(0\ -\ 1, 0 * 24) \rightarrow 24$

L'osservazione fondamentale qui è che quando "`fattoriale_tr(0, 24)`" ritorna, "`fattoriale_tr(1,`

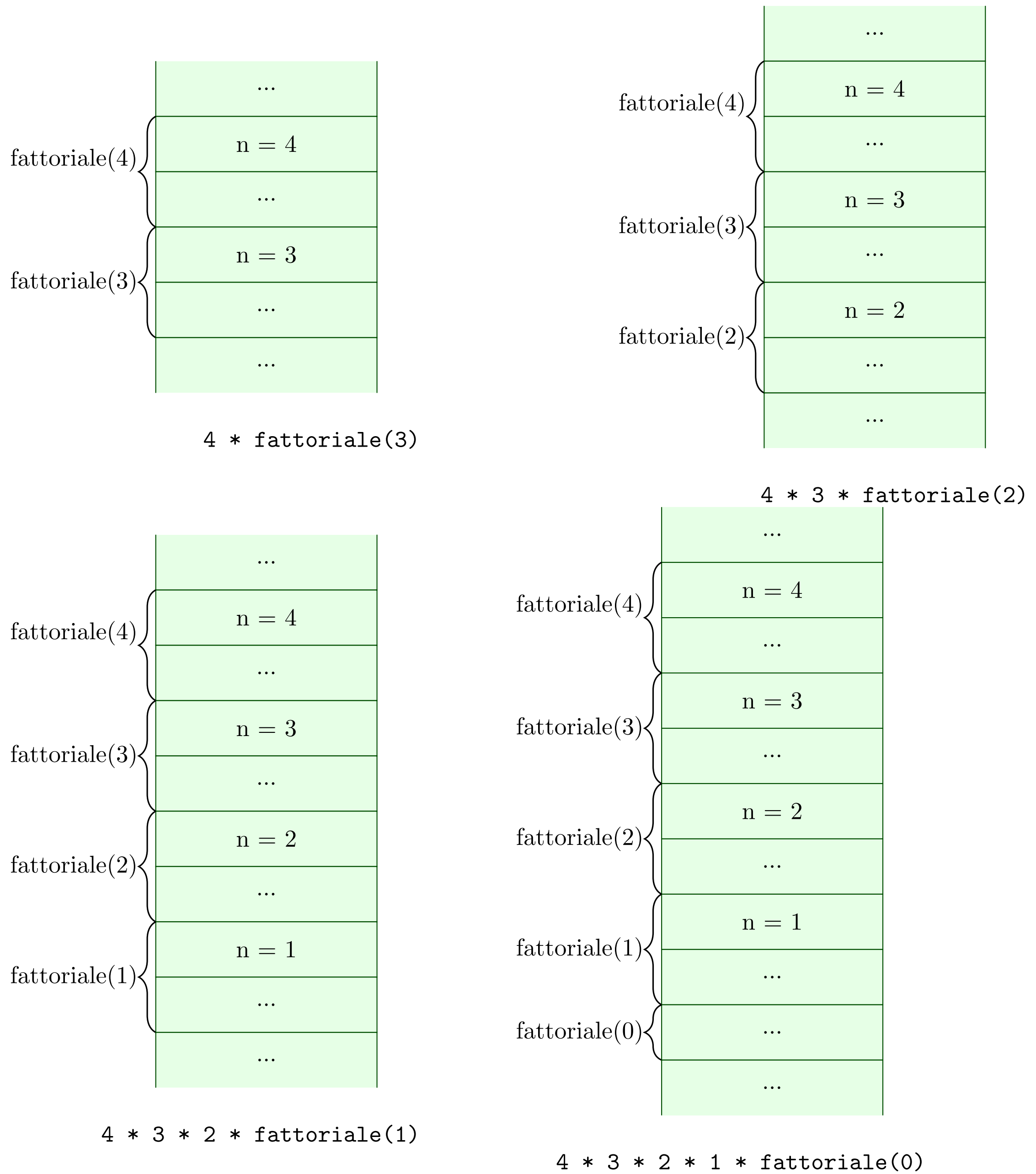

Figure 12: Evoluzione dello stack per l'invocazione `fattoriale(4)`.

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

#include <stdio.h>

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);
}

int main()
{
  unsigned int n;

  printf("N = ");
  scanf("%u", &n);

  printf("Pari(%u): %u\n", n, pari(n));
  printf("Dispari(%u): %u\n", n, dispari(n));

  return 0;
}
```

Figure 13: Funzioni (mutuamente) ricorsive per testare se un numero è pari o dispari.

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 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 gen-

```c
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);
}
```

Figure 14: Versione tail recursive del fattoriale.

erato da `gcc -O1` quando compila la funzione `f1()` di Figura 14. Questo codice è mostrato in Figura 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 17.

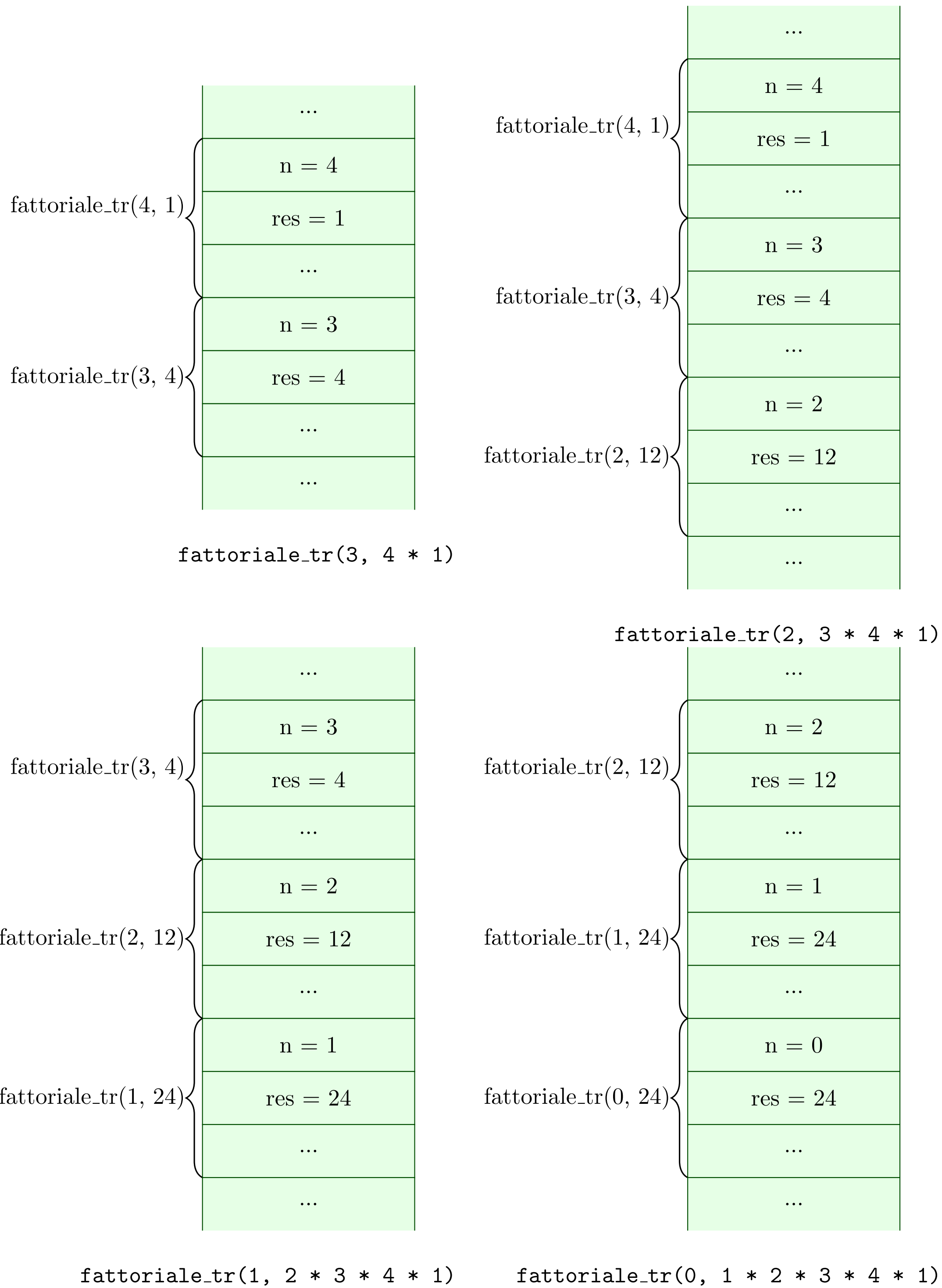

Figure 15: Evoluzione dello stack per l'invocazione `fattoriale_tr(4)`.

```
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
```

Figure 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
```

Figure 17: Versione tail recursive del fattoriale ottimizzata.