Paradigma funzionale e fondamenti di λ-calcolo

Leonardo De Faveri

Indice

Capitolo Nr.1

Computazione

I più comuni linguaggi di programmazione sono basati sul modello della *Macchina di von Neumann*, cioè basano la propria strategia di computazione sul concetto di *variabile modificabile*. Una variabile è un'area di memoria identificata da un nome e un tipo, alla quale, durante l'esecuzione del programma, è possibile assegnare dei valori. Quindi, di fatto, una *variabile modificabile* altro non è se non un contenitore al quale, durante la computazione, possono essere assegnati valori diversi attraverso un'operazione di *assegnazione* che ne modifica il contenuto, lasciando però invariata l'associazione tra il nome e la locazione di memoria corrispondente (viene modificato il *r-value*, mentre rimane invariato il *l-value*, che viene fissato al momento della dichiarazione della variabile). Risulta quindi evidente che nei linguaggi di programmazione basati sul modello di *von Neumann* è fondamentale il concetto di *stato*.

Il paradigma funzionale propone invece una strategia alternativa, nella quale i concetti di *stato* e di *variabile modificabile* e l'operazione di *assegnazione* perdono di significato.

Definizione 1.0.1. Il paradigma della *programmazione funzionale* è un paradigma nel quale la computazione avviene per riscrittura di funzioni e non per modifica dello stato.

Gli elementi fondamentali di questo paradigma sono ora funzioni, funzioni di ordine superiore e ricorsione (l'iterazione è diventata irrilevante nel momento in cui abbiamo stabilito che la computazione non prevede modifiche allo stato).

1.1 Espressioni e funzioni

Sfruttando la sintassi del linguaggio *ML* possiamo definire una funzione come qualcosa del tipo:

```
val f = fn x => x * x;
```

La parola chiave `val` introduce una dichiarazione con la quale l'ambiente viene esteso con una nuova associazione tra un nome e un valore; in questo caso il nome `f` è legato alla funzione che trasforma `x` in `x * x`. L'istruzione `fn` introduce la definizione di una funzione con un *parametro formale* `x`, il quale viene modificato dal corpo della funzione, definito a destra dei caratteri `=>`.

Ciò che potrebbe sembrare strano a qualcuno non abituato a lavorare con i linguaggi funzionali è il fatto che la funzione definita da `fn` sia stata associata al nome `f` dopo la definizione. Questo è possibile perché in tutti i linguaggi funzionali le funzioni sono valori *esprimibili*, cioè possono essere il risultato di espressioni complesse. Nel nostro caso, l'espressione alla destra dell'uguale (`=`) è appunto un'espressione che denota una funzione. Sarebbe infatti possibile estendere ulteriormente l'ambiente creando una nuova associazione con l'istruzione:

```
val g = f;
```

Per quanto riguarda l'applicazione di una funzione a un argomento, possiamo usare diverse sintassi: `f(2)`, `(f 2)` e `f 2` producono tutte lo stesso risultato. È anche possibile scrivere e valutare una funzione senza doverle assegnare un nome, ad esempio:

```
(fn x => x + 1) (6);
```

ha valore 7, che risulta dall'applicazione della funzione (*anonima*) `fn x => x + 1` all'argomento 6, detto anche *parametro attuale*.

È possibile definite funzioni all'interno di altre funzioni, ad esempio:

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

definisce una funzione *somma* che preso un parametro `x` restituisce una funzione anonima, la quale, preso un parametro `y` restituisce `x + y`. Una tale funzione può essere applicata in molteplici modi:

```
val tre = somma 1 2;
val somma2 = somma 2;
val cinque = somma2 3;
```

NB. Il linguaggio *ML* mette a disposizione una sintassi più leggera per definire funzioni. La prima funzione `f` può essere definita come:

```
fun f x = x * x;
```

Più in generale, una definizione del tipo:

```
fun F x1 x2 ... xn = corpo;
```

corrisponde a:

```
val F = fn x1 => (fn x2 => ... (fn xn => corpo) ...);
```

1.2 Computazione come riduzione

La valutazione di un'espressione, ossia il procedimento con il quale si passa da un'espressione complessa ad un valore, può essere descritta come un processo di riscrittura nel quale una sotto espressione nella forma di "funzione applicata ad un parametro" viene sostituita, all'interno dell'espressione complessa che la contiene, con il suo corpo, nel quale al posto del parametro formale è stato posto il parametro attuale.

Esempio Sia data la funzione fattoriale con la seguente definizione ricorsiva:

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

Usando questo metodo di computazione, vediamo come è possibile arrivare ad un valore:

```
fatt 3 → (fn n => if n = 0 then 1 else n * fatt(n - 1)) 3
    → if 3 = 0 then 1 else 3 * fatt(3 - 1)
    → 3*fatt(2)
    → 3*((fn n => if n = 0 then 1 else n * fatt(n - 1)) 2)
    → 3*(if 2 = 0 then 1 else 2 * fatt(2 - 1))
    → 3*(2*fatt(1))
    → 3*(2*((fn n => if n = 0 then 1 else n * fatt(n - 1)) 1))
    → 3*(2*(if 1 = 0 then 1 else 1 * fatt(1 - 1)))
    → 3*(2*(1*fatt(0)))
    → 3*(2*(1*((fn n => if n = 0 then 1 else n * fatt(n - 1)) 0)))
    → 3*(2*(1*(if 1 = 0 then 1 else 1 * fatt(0 - 1))))
    → 3*(2*(1*1))
    → 6
```

Si noti come, ad eccezione dei calcoli aritmetici e delle espressioni condizionali, si proceda per manipolazione simbolica di stringhe.

NB. La seguente definizione:

```
fun r x = r(r(x));
```

è un esempio di funzione *divergente* con risultato indefinito. Ogni riscrittura di `r` infatti si risolve in una riscrittura infinita.

Quindi, sintatticamente, questo tipo di linguaggi si basa su espressioni e non comandi. I due costrutti principali che consentono di definire espressioni sono:

- *Astrazione*: data una qualsiasi espressione `exp` ed un identificatore `x`, permette di definire una nuova espressione del tipo `fn x => exp` che denota la funzione che trasforma il parametro formale `x` in `exp`
- *Applicazione*: un'espressione `f_exp` viene applicata ad un'altra espressione `a_exp`, ovvero la funzione denotata da `f_exp` viene applicata all'argomento `a_exp`;

Il processo di riduzione si basa anch'esso su due operazioni principali:

(i) Quando si incontra un identificatore legato all'ambiente, l'identificatore viene sostituito con la sua definizione;

(ii) Nel caso di un'applicazione funzionale applicata ad un argomento, questo viene sostituito a tutte le occorrenze del parametro formale;

La (ii) è una riformulazione della regola di copia, che qui prende il nome di *β-regola*.

Definizione 1.2.1 (Redex)**.** Un *redex* (*reducible expression*) è un'applicazione della forma `((fn x => corpo) arg)`.

- *Ridotto*: il *ridotto* di un *redex* `((fn x => corpo) arg)` è l'espressione che si ottiene sostituendo in `corpo` ogni occorrenza del parametro formale `x` con una copia di `arg`;
- *β-regola*: un'espressione `exp` nella quale compaia come sotto espressione un *redex* si riduce in `exp1` (notazione: `exp` $\rightarrow$ `exp1`), dove `exp1` si ottiene da `exp` rimpiazzando il *redex* con il suo ridotto;

Capitolo Nr.2

Valutazione

2.1 Valori

Un *valore* è un'espressione che non deve essere ulteriormente riscritta. In un linguaggio funzionale i valori sono di due tipi:

- *Primitivi*: sono i valori di tipo primitivo forniti dal linguaggio. Ad ogni tipo primitivo è associato un insieme di valori che non necessitano di valutazione (e.g. interi, reali, booleani, caratteri, ...);
- *Funzionali*: sono funzioni o espressioni complesse;

È importante notare che i *valori funzionali* nella forma:

```
fn x => exp;
```

sono di per sé *valori* e quindi eventuali *redex* presenti in `exp` non vengono ridotti finché a una tale espressione non viene applicato un argomento.

Esempio Consideriamo la seguente definizione:

```
val G = fn x => ((fn y => y + 1) 2);
```

Quale valore verrà assegnato a `G`?

Sappiamo che una definizione comporta la creazione di un legame tra il nome a sinistra dell'uguale e l'espressione a destra. In questo caso però non è chiaro quale sia l'espressione da associare. Potrebbe essere:

```
fn x => 3;
```

che è l'espressione ottenuta riducendo il *redex* `((fn y => y + 1) 2)`. Oppure potrebbe essere:

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

nella quale non è avvenuta alcuna riduzione.

Sebbene possa sembrare strano, il valore associato a `G` è il secondo, in quanto, sulla base di ciò che è stato detto in precedenza, espressioni della forma `fn x => exp` sono di per sé valori e quindi non avviene alcuna riduzione del *redex* in esse contenuti.

2.2 Strategie di valutazione

Consideriamo il seguente frammento di codice:

```
fun k x y = x;
fun r z = r(r(z));
fun D u = if u = 0 then 1 else u;
fun succ v = v + 1;

val v = K (D (succ 0)) (r 2);
```

Frammento 2.1: Un'espressione con più redex

Quale valore viene associato a `v` e come viene determinato?

La *β-regola* da sola non basta perché a destra dell'uguale sono presenti 4 *redex*:

```
k (D (succ 0))
D (succ 0)
succ 0
r 2
```

Quale viene ridotto per primo? Generalmente i linguaggi usando una valutazione da sinistra a destra, ma anche dopo aver fissato quest'ordine di valutazione non è chiaro quale sia il *redex* più a sinistra tra:

```
k (D (succ 0))
D (succ 0)
succ 0
```

I 3 *redex* sono infatti sovrapposti ed è quindi necessaria un'ulteriore specifica.

2.2.1 Valutazione per valore

In questa strategia di valutazione, detta anche *in ordine applicativo*, *eager* o *inner most*, un *redex* viene valutato solo se l'espressione che costituisce il suo argomento è già un valore.

Il processo, in particolare, prevede 4 passi:

1. Scandisci l'espressione da valutare a partire da sinistra, selezionando la prima espressione che incontri e sia essa `(f_exp a_exp)`;
2. Valuta per prima (applicando ricorsivamente questo stesso metodo) `f_exp`, fino a ridurla ad un valore (*funzionale*) di tipo `(fn x => ...)`;
3. Valutala la parte argomento `a_exp` fino a ridurla ad un valore `val`;
4. Riduci il *redex* `((fn x => ...)  val)` e riparti dal punto (1);

Nel caso del Frammento 2.1, il punto (1) seleziona l'applicazione `(K (D (succ 0)))`. Ora `K`, `D` e `succ` sono già valori, quindi il primo *redex* ad essere ridotto sarà `(succ 0)`, cioè `((fn v => v + 1) 0)` che viene valutato a `1`. A questo punto viene ridotto il redex `(D 1)`, cioè `((fn u => if u = 0 then 1 else u))` che da, anch'esso, valore `1`. Infine, viene valutato `(K 1)` che produce il valore `(fn y => 1)`. Alla fine, l'espressione ha raggiunto la forma:

```
(fn y => 1) (r 2)
```

Siccome la parte funzionale di questa espressione (`f_exp`) è già un valore, la strategia della valutazione per valore prevede che venga valutata la parte argomento (`a_exp`), ovvero `(r 2)`. Riscrivendo quell'espressione si ottiene `(r (r 2))`, poi `(r (r (r 2)))` e così via.

A `v` dunque, non sarà mai associato alcun valore, in quanto la valutazione diverge.

2.2.2 Valutazione per nome

In questa strategia di valutazione, detta anche *in ordine normale* o *outermost*, un *redex* viene valutato prima della sua parte argomento.

In particolare la valutazione procede come segue:

1. Scandisci l'espressione da valutare a partire da sinistra, selezionando la prima applicazione che incontri e sia essa `(f_exp a_exp)`;
2. Valuta per prima (applicando ricorsivamente questo metodo) `f_exp`, fino a ridurla ad un valore (*funzionale*) di tipo `(fn x => ...)`;
3. Riduci il *redex* `((fn x => ...)  a_exp)` usando la β*-regola* e riparti dal punto (1).

Nel caso del Frammento 2.1, il primo *redex* ad essere ridotto è:

```
K (D (succ 0))
```

che viene riscritto come:

```
fn y => D (succ 0)
```

che è un *valore funzionale* nella forma:

```
(fn y => D (succ 0)) (r 2)
```

Ora, il punto (2) prescrive di ridurre il redex più esterno, cioè:

```
D (succ 0)
```

Dopo la riduzione si ottiene:

```
if (succ 0) = 0 then 1 else (succ 0)
```

Procedendo da sinistra si ottiene prima

```
if 1 = 0 then 1 else (succ 0)
```

e quindi

```
succ 0
```

dal quale si ottiene il valore `1`, che è quello che viene assegnato a `v`.

NB. Si noti come non sia stato necessario valutare `(r 2)`. E che quindi sia stato possibile completare la valutazione senza incappare in una divergenza.

2.2.3 Valutazione lazy

Nella valutazione per *nome*, uno stesso *redex* può dover essere ridotto più volte per effetto di qualche duplicazione avvenuta durante il processo di riscrittura. Nell'esempio precedente `(succ 0)` è stato duplicato nella riscrittura della funzione `D` ed è stato ridotto 2 volte. Questa una diretta conseguenza della scelta di non valutare gli argomenti `a_exp` prima della componente funzionale.

Per ovviare ai problemi di efficienza risultanti, la strategia *lazy* procede come la strategia per *nome*, ma la prima volta che incontra un *redex* duplicato, ne salva il valore, che verrà poi usato in corrispondenza delle altre copie.

Nelle strategie per *nome* e quindi anche nella valutazione *lazy* un *redex* viene ridotto solo se necessario alla computazione.

2.3 Confronto tra strategie

Come abbiamo visto, diverse tecniche di valutazione hanno prodotto risultati diversi. Nella fattispecie, con la tecnica di valutazione per *valore* l'espressione da associare a `v` divergeva, mentre usando la valutazione per *nome* si è arrivati ad un valore *primitivo*. Viene quindi da chiedersi se tecniche diverse, a parità di espressione, possano arrivare a produrre due valori diversi (salvo divergenze).

Definizione 2.3.1. Un'espressione è *chiusa* se tutte le sue variabili sono legate da qualche `fn`.

Per rispondere alla domanda di cui sopra, definiamo il seguente teorema:

Teorema 2.3.1. Sia `exp` un'espressione *chiusa*. Se `exp` si riduce ad un valore primitivo `val` usando una qualsiasi delle 3 strategie viste sopra, allora `exp` si riduce a `val` seguendo la strategia per *nome*. Se `exp` diverge usando la strategia per *nome*, allora diverge anche con le altre due strategie.

Quindi, questo teorema esclude che un'espressione *chiusa* valutata secondo due diverse strategie possa produrre due valori `val1` e `val2` diversi tra loro. Inoltre, la dimostrazione del Teorema si basa sulla seguente proprietà che viene addirittura considerata fondamentale per poter definire funzionale un linguaggio di programmazione.

Prop 2.3.1. Fissata una strategia qualsiasi, nello scope dello stesso ambiente la valutazione di tutte le occorrenze di una stessa espressione produce sempre lo stesso valore.

Capitolo Nr.3

Strumenti per la programmazione

Vediamo ora alcuni strumenti tipici dei linguaggi basati sul paradigma funzionale, che permettono di rendere la programmazione più semplice ed espressiva. Va però fatto notare, che quanto visto finora sarebbe già sufficiente a definire un linguaggio *Turing completo*. Continueremo ad usare come riferimento il linguaggio *ML*.

3.1 Interattività

Tutti i linguaggi funzionali forniscono un ambiente interattivo nel quale la *macchina astratta* valuta ogni espressione nel momento in cui viene inserita. Inoltre, ogni definizione modifica l'ambiente globale introducendo un nuovo legame tra un nome e un valore (*primitivo* o *funzionale*). Ovviamente, è consentito importare definizioni da file esterni o esportare l'ambiente corrente.

3.2 Ambiente locale

È possibile definire riferimenti e associazioni all'interno di *scope* limitati. Ciò è realizzato attraverso costrutti del tipo `let-in-end`:

```
let x = exp in exp1 end;
```

In questo esempio, viene introdotto un legame tra il nome `x` e il valore di `exp` in uno *scope* che include solo `exp1`. Questo costrutto, altro non è se non zucchero sintattico per:

```
(fn x => exp1) exp;
```

Questo è vero perché le funzioni introducono degli ambienti locali costituiti dai legami tra *parametri attuali* e *formali*.

3.3 Pattern matching

Uno strumento utilissimo messo a disposizione dai linguaggi di programmazione funzionali è la possibilità di definire funzioni per casi. Vediamo questo esempio:

```
(* Definizione ricorsiva della funzione di Fibonacci *)
fun fib n = if n = 0 then 1
    else if n = 1 then 1
    else fib(n - 1) + fib(n - 2);
```

```
(* Definizione con pattern matching *)
(* Il carattere | si legge "oppure" *)
fun fib 0 = 1
 | fib 1 = 1
 | fib n = fib(n - 1) + fib (n - 2);
```

Il comportamento delle due funzioni è lo stesso, ma la seconda forma risulta molto più chiara e lineare. Quando si definisce una funzione sfruttando il *pattern matching*, è importante ricordare che all'applicazione i pattern vengono confrontati con il parametro attuale in base all'ordine in cui sono definiti. È quindi necessario definire i pattern con un ordine decrescente di specificità.

Questa feature mostra la sua potenza quando si lavora con tipi di dato strutturati, come ad esempio le liste. In *ML* le liste sono indicate tra parantesi quadre con i valori separati tra parentesi:

```
["uno", "due", "tre"];
```

L'operatore `::` consente di aggiungere un valore in testa ad una lista, mentre `nil` rappresenta la lista vuota `[]`.

```
"zero" :: ["uno", "due", "tre"];(* => ["zero", "uno", "due", "tre"] *)
"quattro" :: [];                (* => ["quattro"] *)
```

Quindi, usando il *pattern matching*, possiamo definire una funzione per il calcolo della lunghezza di una stringa come segue:

```
fun lung nil = 0
 | lung (x::xs) = 1 + lung(xs);
```

Si noti come i nomi usati nel pattern sono poi usati all'interno della funzione per riferirsi a parti del parametro attuale.

Quando si sfrutta questo meccanismo è fondamentale ricordare che i pattern servono per distinguere la *forma* del parametro attuale e non il valore, pertanto la seguente definizione non avrebbe alcun senso:

```
(* La funzione testa l'uguaglianza dei primi due valori di una lista*)
fun testaUguaglianza nil = false
 | testaUguaglianza [x] = false
 | testaUguaglianza [x::x::xs] = true(* il nome x non può ripetersi *)
 | testaUguaglianza [x::y::zs] = false;
```

NB. Se in un pattern non fossimo interessati ad utilizzare uno o più dei parametri della funzione, nella definizione del pattern, potremmo specificare quei parametri con un carattere di underscore _.

```
fun mul (_, 0) = 0
    | mul (0, _) = 0
    | mul (a, b) = a * b;
```

3.4 Tipi di dato

Come detto esistono tipi di dato *primitivi* messi a disposizione direttamente dai linguaggi. Tipicamente questi tipi sono:

- `int`: valori interi;
- `real`: numeri reali (virgola mobile);

- `char`: caratteri;
- `string`: stringhe;
- `bool`: valori booleani `true/false`;

Questi tipi di dato possono poi essere aggregati in liste o tuple:

```
(* Definisce una coppia di interi
    int * int
*)
val coppia1 = (1, 2);

(* Una tupla di 3 elementi
    char * real * bool
*)
val trio = (#"a", 2.4, true);

(* 'a list *)
val lista_vuota = [];

(* int list *)
val lista_interi = [1, 2, 3, ~1, ~2, ~3];
```

Si noti in particolare il tipo della lista vuota `'a list`: `list` indica ovviamente il fatto che si tratti di una lista, mentre `'a` indica una variabile di tipo, cioè un tipo generico non ancora istanziato. In particolare qualunque tipo preceduto da un apostrofo `'` è un tipo generico.

Per quanto riguarda i tipi di dato *funzionali*, le funzioni sono valori denotabili ed esprimibili. Ad esempio:

```
(* val f = int -> int *)
val f = fn x => x + 1;

(* val g = fn: int * string * char list -> char list *)
val g = fn (x, y, z) => if x = 2 then explode(y) else z;

(* val k = fn: ''a * ''a * bool -> bool *)
val k = fn (x, y, z) => if x = y then z else false;

(* val l = fn: 'a * 'b -> 'b * 'a *)
val l = fn (x, y) => (y, x);

(* val h = fn real * int -> int * real *)
val h = fn (x: real, y: int) => (y, x);
```

Da questi esempi emerge chiaramente il fatto che, tipicamente, i linguaggi di tipo funzionale abbiano meccanismi d'identificazione (inferenza) di tipo in base all'espressione in cui compare un certo dato. Nella funzione `f` il tipo del parametro `x` viene identificato essere `int` perché in `f_exp` viene sommato al valore `1` che è intero.

Per il parametro `x` della funzione `g` vale lo stesso ragionamento; `y` viene stabilito essere di tipo `string` perché viene fornito come parametro alla funzione `explode` che accetta una stringa come argomento e siccome il costrutto `if-then-else` è valorizzato, ovvero il tipo del ramo `else` deve essere lo stesso del `then`, `z` deve essere dello stesso tipo del valore di ritorno di `explode(y)`.

È interessante soffermarsi sul tipo della funzione `k`. Come abbiamo visto in precedenza i tipi di preceduti da `'` sono etichette per tipi non ancora istanziati e che quindi vengono stabili

al momento dell'invocazione. In questo caso però abbiamo due apostrofi. Questi indicano il fatto che il tipo dei parametri `x` e `y` deve essere un *equality type*, ovvero un tipo sul quale è applicabile l'operatore di uguaglianza `=`.

Nella funzione `l` invece, abbiamo un esempio di come sia possibile usare una stringa qualsiasi per indicare un tipo generico, in questo caso `'a` e `'b`. Ovviamente, nulla vieta che al momento dell'invocazione della funzione i tipi dei parametri si eguaglino, cioè anche se indicati con etichette diverse i tipi dei parametri *attuali* possono essere uguali.

Si noti come nell'ultima funzione `h`, la definizione sia identica alla precedente, tranne per il fatto che abbiamo specificato esplicitamente quali dovessero essere i tipi.

Si consideri ora la seguente definizione:

```
fun delta x = x x;
```

Il tipo della funzione `delta` non è inferenziabile. L'espressione `(x x)` è illegale in quanto non vi è alcun modo per assegnare un unico tipo consistente ad `x`. A sinistra occorrerebbe qualcosa del tipo `'a -> 'b`, mentre a destra dovrebbe soddisfare il tipo dell'argomento della funzione, cioè `'a`. Quindi, `x` dovrebbe essere contemporaneamente di tipo `'a` e `'a -> 'b`.

3.4.1 Alias

Alcuni linguaggi funzionali mettono a disposizione dell'utente la possibilità di definire *alias* per tipi già esistenti (o altri *alias*). In *ML* ciò è fatto con l'istruzione `type`:

```
(* Definiamo un alias per il tipo int list *)
type signal = int list;

(* val v1 = int list *)
val v1 = [1, 2];

(* val v2 = signal *)
val v2 = [1, 2]: signal;

(* Possiamo confrontare v1 e v2 in quanto, anche se v1 è di tipo
    signal, rimane comunque una int list
*)
v1 = v2; (* true *)
```

Si noti, che è necessario esplicitare il tipo nel caso si voglia usare un alias.

Un caso in cui gli alias possono essere molto utili, è quello in cui si sta lavorando con liste di tuple. Vediamo un esempio:

```
(* Definisco un alias per le tuple composte da 2 valori di un tipo e
    il terzo di un secondo tipo
*)
type ('a, 'b) mapping = ('a * 'a * 'b) list;

(* val it = [(1, 1, 2.0)]: (int * int * real) list *)
[(1, 1, 2.0)]

(* val it = [(1, 1, 2.0)]: (int, real) mapping *)
[(1, 1, 2.0)] : (int, real) mapping;
```

Come nel caso precedente, abbiamo dovuto forzare l'utilizzo dell'alias.

3.4.2 Definire nuovi tipi di dato

Oltre che creare *alias* per tipi di dato esistenti, è possibile anche creare nuovi tipi in grado di modellare le specifiche esigenze di un programma. In *ML* per fare questo abbiamo il comando `datatype`.

```
(* Crea un tipo di dato "frutto" che può assumere solo 3 valori: mela,
    pera, uva
*)
datatype frutto = mela | pera | uva;

(* Possiamo sfruttare questo nuovo tipo di dato come qualsiasi altro
   tipo
*)
fun isPera(x) = (x = pera);
isPera(mela); (* => false *)
isPera(pera); (* => true *)
```

L'esempio precedente definisce un tipo di dato simile a quello che sono gli `enum` in `C++`, inoltre i valori `mela`, `pera`, `uva` sono *costruttori costanti* per il tipo di dato `frutto`. È possibile avere anche *costruttori non costanti*.

```
(* Definisce un tipo "oggetto" con due funzioni (nome, prezzo) che
    restituiscono valori di tipo "oggetto"
*)
datatype oggetto = nome of string | prezzo of real;

(* val ogg1 = nome "stringa": oggetto *)
val ogg1 = nome("stringa");
(* val ogg2 = prezzo 4.2: oggetto *)
val ogg2 = prezzo(4.2);

(* Definisce un tipo di dato con costruttore da tipi generici *)
datatype ('a, 'b) element =
   P of 'a * 'b |
   S of 'a;

(* val it = P ("a", 1): (string, int) element *)
P ("a", 1);
(* val it = P (1.0, 2.0): (real, real) element *)
P (1.0, 2.0);

(* val it = S 4: (int, 'a) element *)
S (4);
(* val it = S ["a", "b"]: (string list, 'a) element *)
S (["a","b"]);

(* Definisce un albero binario di tipo generico ('a) chiamato btree *)
datatype 'a btree =
Empty |                              (* Ramo vuoto *)
Node of 'a * 'a btree * 'a btree;    (* Ramo con due rami *)

(* val zeroNodi = Empty: 'a btree *)
val zeroNodi = Empty;
```

```
(* val unNodo = Node (1, Empty, Empty): int btree *)
val unNodo = Node(1, Empty, Empty);
(* val dueNodi = Node (1, Node (2, Empty, Empty), Empty): int btree *)
val dueNodi = Node(1, Node(2, Empty, Empty), Empty);
```

È anche possibile definire strutture più complesse usando le keyword `signature` e `structure` che, grosso modo, corrispondono rispettivamente ai file `.h` e `.cpp` nel `C++`. `signature` consente infatti, di definire nuovi tipi di dato e le firme di tutte le funzioni su di essi. Tali funzioni saranno poi implementate nel blocco introdotto da `structure`.

```
signature STACK =
sig
    exception StackEmpty
    val empty : 'a list
    val isEmpty : 'a list -> bool
    val push : 'a * 'a list -> 'a list
    val top : 'a list -> 'a
    val pop : 'a list -> 'a list
    (* Specifica il fatto che il tipo deve essere un equality type *)
    eqtype 'a stack
end;
(* All'intero di signature non serve mettere i punti e virgola *)

(* Implementa la struttura appena modellata *)
structure Stack =
struct
    type 'a stack = 'a list;
    exception StackEmpty;
    val empty = [];
    fun isEmpty [] = true
        | isEmpty _ = false;
    fun push (x, elements) = x :: elements;
    fun top [] = raise StackEmpty
        | top (x::xs) = x;
    fun pop [] = raise StackEmpty
        | pop (x::xs) = xs;
end :> STACK;
```

Il simbolo `:>` introduce il nome della `signature` sulla quale si vuole basare una struttura. Avremmo potuto anche specificarlo tra il nome della struttura e l'uguale. Vediamo come usare una struttura:

```
(* Creazione di uno stack di interi*)
(* val stack = [1]: int list *)
stack = Stack.push(1, Stack.empty);
(* val stack = []: int list *)
stack = Stack.pop();
(* val it = true: bool *)
stack = Stack.empty;
```

Si notino, in particolare, due cose:

(i) Il tipo del valore `stack` è `int list` e non `int Stack.stack` come ci saremmo aspettati;

(ii) Il controllo `stack = Stack.empty` restituisce un valore, quindi `Stack` è un *equality type*;

Se volessimo che il linguaggio riconoscesse il valore `stack` come `int Stack.stack` potremmo riscrivere la signature come segue:

```
signature STACK =
sig
    type 'a stack
    val empty : 'a stack
    val isEmpty : 'a stack -> bool
    val push : 'a * 'a stack -> 'a stack
    val top : 'a stack -> 'a
    val pop : 'a stack -> 'a stack
end;
```

Se ora rieseguiamo le istruzioni di prima otteniamo:

```
(* Creazione di uno stack di interi*)
(* val stack = ?: int Stack.stack *)
stack = Stack.push(1, Stack.empty);
(* val stack = ?: int Stack.stack *)
stack = Stack.pop();
(* Errore! *)
stack = Stack.empty;
```

Il tipo di dato adesso è quello che volevamo, ma come possiamo dedurre dal risultato dell'ultima istruzione, il tipo `Stack` non è più un *equality type*.

Capitolo Nr.4

Lambda calcolo

All'inizio della trattazione, abbiamo detto che i linguaggi più comuni poggiano sul modello di computazione proposto da Alan Turing, mentre abbiamo detto essere basati su un modello diverso i linguaggi funzionali. Quel modello è il *Lambda Calcolo* ideato da Alonso Church.

Questo modello è strettamente legato al concetto matematico di funzione. Nella matematica moderna una funzione è definita come un sottoinsieme del prodotto cartesiano di due insiemi che sono, rispettivamente, dominio e codominio. Per ogni funzione vengono poi distinti due aspetti:

- *Estensionalità*: Una funzione è caratterizzata dall'associazione tra i valori del dominio e del codominio; non importa come il valore in ingresso sia trasformato per ottenere il risultato;
- *Intensionalità*: Una funzione è caratterizzata dal procedimento matematico che permette di arrivare ad un valore in uscita partendo dall'argomento in ingresso;

Ad esempio, proviamo a considerare le seguenti funzioni:

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

Da un punto di vista *estensionale* queste funzioni sono identiche perché le associazioni realizzate dall prima sono le stesse della seconda. Tuttavia, dal punto di vista dell'*intensionalità* sono due funzione diverse.

In questo secondo aspetto si inserisce il *Lambda calcolo* che, attraverso un sistema di funzioni e nozioni logiche, semplifica la scrittura delle funzioni.

Utilizzando la notazione del *Lambda calcolo* possiamo definire le funzioni dell'esempio precedente come segue:

$$\lambda x.x + 1$$

$$\lambda x.(x - 2) + 1 + 1 + 1$$

NB. In realtà, questa notazione non è veramente corretta, in quanto i simboli $+$, 2 e 1, nell'ambito del λ-*calcolo*, non hanno alcun significato. Tuttavia, per il momento, per non creare confusione useremo una notazione semplificata, ma vedremo più avanti nella trattazione come quei simboli vengano rappresentati veramente.

4.1 Notazione

Nell'ultimo esempio abbiamo introdotto la notazione del *Lambda calcolo*, vediamo quindi che significato hanno i simboli inseriti:

- λ: introduce il parametro x (è l'equivalente di `fn`);
- x: è il parametro della funzione;
- . e (): sono simboli terminali;
- $x + 1$: un'espressione che probabilmente userà il parametro x;

L'espressione nella sua interezza, cioè $\lambda x.x + 1$ nel primo caso, prende il nome di λ-*termine* o λ-*astrazione*. Tanto è semplice, possiamo definire tutta la sintassi del *Lambda calcolo* in una sola riga:

$$M ::= x \mid (MM) \mid \lambda x.M$$

Ciò significa che un'espressione M può essere uno tra:

- x: una variabile;
- (MM): l'applicazione di un'espressione ad un'altra (possono essere due espressioni diverse);
- $\lambda x.M$: un'astrazione, cioè una funzione che data x restituisce M;

Per chiarezza e per migliorare la leggibilità possono essere usate delle parentesi all'interno delle espressioni. Ad esempio, le seguenti espressioni sono equivalenti:

$$(((f_1 f_2) f_3) f_4) \equiv f_1 f_2 f_3 f_4$$

E ciò ci dice anche che, di base, l'applicazione è da sinistra.

Sempre parlando di espressioni, distinguiamo il concetto di *astrazione* da quello di *applicazione*:

- *Astrazione*: $\lambda x.M$;
- *Applicazione*: (MM);

4.2 Variabili legate e libere

Se abbiamo un λ-*termine* $\lambda x.M$, l'operatore di astrazione λ lega la variabile sulla quale agisce, cioè lega x all'espressione M. A questo punto introduciamo i concetti di *variabili libere* e *variabili legate*: data una generica espressione M definiamo l'insieme delle sue *variabili libere*, indicato con $Fv(M)$ (free variables), e delle sue *variabili legate*, $Bv(M)$ (bound variables), come segue:

$$\begin{aligned} Fv(x) &= \{x\} & Bv(x) &= \emptyset \\ Fv(MN) &= Fv(M) \cup Fv(N) & Bv(M\ M) &= Bv(M) \cup Bv(N) \\ Fv(\lambda x.M) &= Fv(M) - \{x\} & Bv(\lambda x.M) &= Bv(M) \cup \{x\} \end{aligned}$$

La nozione di *variabile legata* ha due implicazioni:

- *Semantica*: la ridenominazione consistente della *variabile legata* non modifica la semantica dell'espressione;
- *Sintattica*: eventuali *sostituzioni* non hanno effetto sulle variabili legate;

Sostituzione Diamo ora, una definizione formale di *sostituzione*. Definiamo la notazione *M[N/x]* che leggiamo: sostituzione di N al posto delle occorrenze libere di x in M.

Vediamo alcune casistiche:

$$\begin{aligned} x[N/x] &= N \\ y[N/x] &= y && \text{qualora } x \neq y \\ (M_1M_2)[N/x] &= (M_1[N/x]M_2[N/x]) \\ (\lambda y.M)[N/x] &= (\lambda y.M[N/x]) && \text{qualora } x \neq y \text{ e } y \notin Fv(N) \\ (\lambda y.M)[N/x] &= (\lambda y.M) && \text{qualora } x = y \end{aligned}$$

Esaminiamo i casi singolarmente:

(i) $x[N/x] = N$: sostituendo N al posto di x in x ottengo N;

(ii) $y[N/x] = y$: se $x \neq y$ non ho nessun posto in cui andare a sostituire N, quindi rimane y;

(iii) $(M_1M_2)[N/x] = (M_1[N/x]M_2[N/x])$: sostituire N a x nel risultato dell'applicazione di M_1 a M_2 è equivalente a sostituire N al posto di x in M_1 e M_2 e poi procedere con l'applicazione;

(iv) $(\lambda y.M)[N/x] = (\lambda y.M[N/x])$: ricordando che N va sostituito a tutte le occorrenze libere di x, qui possiamo distinguere 3 casi:

   (a) $x \neq y$ e $y \notin Fv(N)$: siccome $x \neq y$ sono sicuro di non andare a toccare le occorrenze della *variabile legata y*. La seconda condizione invece, mi garantisce che l'espressione N non contiene *variabili libere* chiamate y, le quali, se venissero sostituite dentro M creerebbero una situazioni di conflitto per via dell'omonimia con il parametro della funzione;

   (b) $x \neq y$ e $y \in Fv(N)$: qui, per evitare che si verifichi il problema appena discusso, rinomino la *variabile legata y* prima di effettuare la sostituzione;

   (c) $x = y$: in questo caso sostituire equivarrebbe a cambiare il nome del parametro e quindi non cambierebbe la *semantica* dell'espressione;

Da quanto visto finora sappiamo che due espressioni che differiscono solo per il nome delle *variabili legate* sono equivalenti, ma intuitivamente, possiamo pensare che lo stesso valga anche per il nome delle *variabili libere*.

Prop 4.2.1 (α-equivalenza)**.** Due espressioni che differiscono solo per il nome delle *variabili libere* sono *α-equivalenti*, cioè vale:

$$\lambda x.M \equiv_\alpha \lambda y.M[y/x] \text{ con } y \text{ fresca}$$

Con "fresca" vogliamo dire che si tratta di una variabile non presente in M.

Oss. Due termini che differiscono tra loro per il solo fatto che qualche sotto termine dell'uno è stata rimpiazzato con un sotto termine *α-equivalente*, saranno considerati uguali.

4.3 Computazione

4.3.1 β-riduzione

Prop 4.3.1 (β-riduzione)**.** La regola di *β-riduzione* prescrive che un *λ-termine* $\lambda x.M$ applicato ad un'espressione N, venga ridotto come segue:

$$(\lambda x.M)N \to_\beta M[N/x]$$

In generale, diremo che M *β-riduce* a N, e scriveremo in simboli $M \to N$, quando N è il risultato dell'applicazione di un passo di *β-riduzione* a qualche sottotermine di M. Ad esempio, un sottotermine della forma $(\lambda x.M)N$ è un *redex*, il cui *ridotto* è $M[N/x]$.

Oss. La *β-riduzione* è una relazione non simmetrica, cioè se $M \to N$, non è sempre vero che $N \to M$.

Oss. La *β-riduzione* è una relazione non deterministica, in quanto, se nello stesso termine esistono più *redex*, non definisce una regola che indichi l'ordine in cui ridurli.

Prop 4.3.2 (β-uguaglianza)**.** Tra 2 espressioni M ed N sussiste una relazione di *β-uguaglianza* se M ed N sono collegate da una sequenza di *β-riduzioni*, non necessariamente tutte dello stesso verso. Scriveremo in simboli:

$$M =_\beta N$$

4.3.2 Forme normali

Per ora non abbiamo ancora discusso quando un *λ-termine* sia sufficientemente semplice da non richiedere più alcuna riduzione.

Prop 4.3.3. Un *λ-termine* che non contiene alcun *redex* è detto essere in *forma normale*.

Possiamo affermare che quando un *λ-termine* ha raggiunto una *forma normale* possiamo terminare la *β-riduzione*. In particolare un *λ-termine* in *forma normale*, secondo il paragrafo 2.1, è un *valore*, in quanto non deve essere ulteriormente riscritto. Ovviamente, non vale il contrario, cioè non tutti i *valori* sono *λ-termini* in *forma normale*.

Esempio Vediamo un paio di esempi:

- $\lambda x.(\lambda y.x)$ è una *forma normale* perché non può essere ridotta;
- $\lambda x.(\lambda y.y)x$ non è una *forma normale* perché contiene un sottotermine riducibile $(\lambda y.y)x$. Procediamo allora con la riduzione:

  $$\lambda x.(\lambda y.y)x \to \lambda x.x$$

  $\lambda x.x$ è ora una *forma normale*;

4.3.3 Confluenza

Finora abbiamo ragionato nell'ipotesi che tutti *λ-termini* fossero riducibili ad una *forma normale*, ma in realtà non è così. Ad esempio il seguente *λ-termine* diverge:

$$(\lambda x.xx)(\lambda x.xx) \to_\beta (xx)[(\lambda x.xx)/x] = (\lambda x.xx)(\lambda x.xx) \to_\beta (xx)[(\lambda x.xx)/x] = \dots$$

Un altro problema che era sorto quando abbiamo dato la definizione di *β-riduzione* era il non determinismo, ma fortunatamente, per tutti i *λ-termini* vale la seguente proprietà:

Proprietà (Convergenza)**.** Dato un *λ-termine* M, se M si riduce a N_1 con qualche passo di riduzione, e si riduce anche a N_2 con qualche passo di riduzione, allora esiste un termine P tale che sia N_1 che N_2 si riducono a P con qualche passo di riduzione.

Oss. Se un *λ-termine* può essere ridotto ad una *forma normale*, allora questa è unica e indipendente dall'ordine dei passi di riduzione eseguiti per raggiungerla.

Vediamo ora, alcuni esempi di riduzioni:

Esempio Riduci in *forma normale* la seguente:

$$(\lambda x.x(xy))(\lambda z.zx)$$

Prima d'iniziare, ricordo la β-*regola*:

$$(\lambda x.M)N \to_\beta M[N/x]$$

In questo caso posso porre $M = x(xy)$ e $N = (\lambda z.zx)$ e riscrivere quindi la mia espressione come:

$$(\lambda x.x(xy))(\lambda z.zx) \to_\beta x(xy)[(\lambda z.zx)/x]$$

Andando a sostituire ottengo:

$$x(xy)[(\lambda z.zx)/x] = (\lambda z.zx)((\lambda z.zx)y)$$

Procedo allo stesso modo fino ad arrivare alla *forma normale*:

$$(\lambda z.zx)((\lambda z.zx)y) \to (\lambda z.zx)(yx) \to (yx)x$$

Esempio Riduci in *forma normale* la seguente:

$$(\lambda x.xy)(\lambda x.xz)(\lambda z.zx)$$

Per risolvere questo esercizio è fondamentale ricordare che l'applicazione è da sinistra:

$$(\lambda x.xy)(\lambda x.xz)(\lambda z.zx) \to ((\lambda x.xz)y)(\lambda z.zx) \to yz(\lambda z.zx)$$

Esempio Riduci in *forma normale* la seguente:

$$(\lambda t.tx)((\lambda z.xz)(xz))$$

$$(\lambda t.tx)((\lambda z.xz)(xz)) \to (\lambda t.xt)(x(xz)) \to (x(xz))x$$

Esempio Riduci in *forma normale* le seguenti:

1. $(\lambda x.yx)((\lambda y.\lambda t.yt)zx)$
2. $(\lambda x.xzx)((\lambda y.yyx)z)$
3. $(\lambda x.xy)(\lambda t.tz)((\lambda x.\lambda z.xyz)yx)$

Soluzioni:

1. Per la riduzione posso pensare di partire dalla parte più esterna, o più interna. Vediamo prima questo secondo approccio:

   Tenendo a mente la definizione di β-*regola*, possiamo porre $M = \lambda t.yt$ e $N = zx$. Vado quindi a sostituire N dentro M al posto di tutte le occorrenze libere di y:

   $$(\lambda x.yx)((\lambda y.\lambda t.yt)zx) \to (\lambda x.yx)(\lambda t.zxt)$$

   A questo punto il termine più interno è stato ridotto, quindi vado a ridurre $\lambda x.yx$:

   $$(\lambda x.yx)(\lambda t.zxt) \to y(\lambda t.zxt)$$

   Viceversa, procedendo dall'esterno avremmo:

   $$(\lambda x.yx)((\lambda y.\lambda t.yt)zx) \to y((\lambda y.\lambda t.yt)zx) \to y(\lambda t.zxt)$$

2. Anche qui potremmo procedere dall'interno o dall'esterno. Scegliamo di ridurre partendo dall'interno:
$$(\lambda x.xzx)((\lambda y.yyx)z) \to (\lambda x.xzx)(zzx) \to zzxzzzx$$

3. Procediamo a ridurre dall'esterno:
$$(\lambda x.xy)(\lambda t.tz)((\lambda x.\lambda z.xyz)yx) \to (\lambda t.tz)y((\lambda x.\lambda z.xyz)yx) \to yz((\lambda x.\lambda z.xyz)yx)$$
Giunti a questo punto l'unica cosa che possiamo ridurre è il *redex* interno:
$$yz((\lambda x.\lambda z.xyz)yx) \to yz(\lambda x.yxyz)$$

4.4 Espressività

Per quanto visto finora, il λ-*calcolo* sembra un linguaggio estremamente limitato, con solo 3 tipi di espressioni: variabili, astrazioni e applicazioni. Ad esempio, l'espressione $\lambda x.x + 2$ non è valida in quanto l'operazione + e il simbolo 2 non sono definiti.

Nonostante questo, il λ-*calcolo* è un modello di computazione *Turing completo*, cioè può risolvere ogni problema che ammette soluzione. A livello generale, per poter essere *Turing completo* un linguaggio, o un modello di computazione, deve poter esprimere i numeri naturali, le operazioni aritmetiche e prevedere un meccanismo d'iterazione.

4.4.1 Rappresentazione dei numeri naturali

In matematica i numeri naturali sono definiti a partire dagli *assiomi di Peano* e, partendo dallo zero, tutti i numeri sono determinati come 1 + il precedente:
$$succ(n) = n + 1$$

Per quanto riguarda il λ-*calcolo*, Alonso Church ipotizzò che un numero potesse essere rappresentato come il numero di volte che una funzione viene applicata ad un argomento. Secondo l'idea di Church quindi, lo 0 viene espresso come:
$$0 \coloneqq \lambda f.\lambda x.x$$
e un generico $n \in \mathbb{N}$ è rappresentato applicando n volte la funzione f ad x. Formalmente scriviamo:
$$f^n = \underbrace{f \circ f \circ \cdots \circ f}_{\text{n volte}}$$
che è equivalente alla notazione nf per il λ-*calcolo*.

Vediamo alcuni esempi:

Numero	**Espressione matematica**	**λ-termine**
0	$f\ x = x$	$\lambda f.\lambda x.x$
1	$f\ x = f(x)$	$\lambda f.\lambda x.f\ x$
2	$f\ x = f(f(x))$	$\lambda f.\lambda x.f(f\ x)$
3	$f\ x = f(f(f(x)))$	$\lambda f.\lambda x.f(f(f\ x))$
$\vdots$	$\vdots$	$\vdots$
n	$f\ x = f^n(x)$	$\lambda f.\lambda x.nf\ x$

Riassumendo: un numero n è espresso applicando n volte una funzione ad un dato argomento. Se la funzione è la funzione che restituisce il successivo di un numero, allora n è ottenuta applicando quella funzione n volte all'argomento 0. Ciò che è importante ricordare è che è la funzione stessa la rappresentazione del numero non il suo risultato.

Ma come è definita la funzione per il successivo?

Il valore n è $\lambda f.\lambda x.nf\ x$, quindi definiamo la funzione $succ(n)$ come:

$$succ(n) \coloneqq \lambda n.\lambda f.\lambda x.f(nfx)$$

NB. Potrebbe essere difficile convincersi della correttezza di questa definizione, ma se applicata alla rappresentazione di n, quell'espressione si riduce ad un λ-*termine* che corrisponde al successivo di n.

Esempio Calcoliamo $succ(0)$:

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

Prima di procedere con la riduzione chiariamo un po' il significato di quest'espressione:

- $(\lambda n.\lambda f.\lambda x.f((nf)x))$: definizione della funzione $succ$;
- $(\lambda f.\lambda x.x)$: rappresentazione del numero 0;

Per evitare di creare confusione durante la riduzione rinominiamo, all'interno della definizione di $succ$, f in y e x in g:

$$(\lambda n.\lambda f.\lambda x.f((nf)x)) \equiv_\alpha (\lambda n.\lambda y.\lambda g.y((ny)g))$$

Procediamo ora con la riduzione:

$$(\lambda n.\lambda y.\lambda g.y((ny)g))(\lambda f.\lambda x.x) \to_\beta \lambda y.\lambda g.y((ny)g)[(\lambda f.\lambda x.x)/n] = \lambda y.\lambda g.y(((\lambda f.\lambda x.x)y)g)$$

$$\to \lambda y.\lambda g.y((\lambda x.x)g) \to \lambda y.\lambda g.yg$$

Infine, sostituendo di nuovo i nomi originali otteniamo:

$$\lambda f.\lambda x.fx$$

La forma raggiunta è la rappresentazione del valore 1, cioè il successivo di 0.

Esempio Calcoliamo $succ(1)$:

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

Procediamo come prima, rinominando f e x nel corpo di $succ$:

$$\lambda n.\lambda y.\lambda g.y((ny)g)(\lambda f.\lambda x.fx) \to \lambda y.\lambda g.y(((\lambda f.\lambda x.fx)y)g)$$

$$\to \lambda y.\lambda g.y((\lambda x.yx)g) \to \lambda y.\lambda g.y(yg)$$

A questo punto, cambiamo i nomi ed otteniamo:

$$\lambda f.\lambda x.f(fx)$$

ovvero, la rappresentazione del 2.

4.4.2 Rappresentazione della somma

Partendo dalla rappresentazione che abbiamo dato per un generico $n \in \mathbb{N}$, possiamo derivare facilmente la definizione, almeno a livello concettuale, della funzione di somma. Ad esempio, la somma tra 2 e 3 può essere descritta come:

> "Applicare 2 volte una funzione al risultato dell'applicazione per 3 volte della stessa funzione all'argomento 0"

Supponiamo quindi che, dati due numeri naturali $n, m \in \mathbb{N}$, si voglia calcolare la somma $n+m$. Innanzitutto, vediamo le rappresentazioni in λ-*calcolo* di n ed m:

- $n \coloneqq \lambda f.\lambda x.nfx$;
- $m \coloneqq \lambda f.\lambda x.mfx$;

Idealmente, realizzare la somma tra n e m significa, per quanto detto sopra, applicare n volte una funzione al risultato dell'applicazione per m volte di quella stessa funzione all'argomento 0. Ciò, equivale a mettere al posto di x nella rappresentazione di n, il "corpo" di m:

$$\lambda f.\lambda x.nfx[(mfx)/x] = \lambda f.\lambda x.nf(mfx)$$

Siccome la somma è una funzione in due parametri, la sua definizione in λ-*calcolo* è:

$$\lambda n.\lambda m.\lambda f.\lambda x.nf(mfx)$$

Esempio Vediamo come calcolare la somma $2 + 3$.
La somma, come visto, è definita come:

$$\lambda n.\lambda m.\lambda f.\lambda x.nf(mfx)$$

Gli argomenti 2 e 3 sono rispettivamente $\lambda f.\lambda x.f(fx)$ e $\lambda f.\lambda x.f(f(fx))$. Applichiamo quindi i due argomenti alla funzione:

$$\lambda n.\lambda m.\lambda f.\lambda x.nf(mfx)(\lambda f.\lambda x.f(fx))(\lambda f.\lambda x.f(f(fx)))$$

Prima di procedere con la riduzione, rinominiamo f nella rappresentazione della somma e del 2, rispettivamente come g e h:

$$\lambda n.\lambda m.\lambda g.\lambda x.ng(mgx)(\lambda h.\lambda x.h(hx))(\lambda f.\lambda x.f(f(fx)))$$

A questo punto, siamo pronti a iniziare la riduzione:

$$\lambda n.\lambda m.\lambda g.\lambda x.ng(mgx)(\lambda h.\lambda xh(hx))(\lambda f.\lambda xf(f(fx)))$$
$$\to \lambda g.\lambda x.(\lambda h.\lambda x.h(hx))g((\lambda f.\lambda x.f(f(fx)))gx)$$

Ora, per chiarezza, rinominiamo il parametro x di λg e λh, rispettivamente come y e z:

$$\lambda g.\lambda y.(\lambda h.\lambda z.h(hz))g((\lambda f.\lambda x.f(f(fx)))gy)$$

Prima di riprendere la riduzione, ricordiamo che l'applicazione è sempre da sinistra:

$$\lambda g.\lambda y.(\lambda h.\lambda z.h(hz))g((\lambda f.\lambda x.f(f(fx)))gy) \to \lambda g.\lambda y.(\lambda z.g(gz))((\lambda x.g(g(gx)))y)$$
$$\to \lambda g.\lambda y.(\lambda z.g(gz))(g(g(gy))) \to \lambda g.\lambda y.(g(g(g(g(gy)))))$$

Se ora rimettiamo i nomi originali, otteniamo:

$$\lambda f.\lambda x.(f(f(f(f(fx)))))$$

Questa è proprio la rappresentazione del 5, quindi abbiamo calcolato con successo la somma tra 2 e 3.

Notazione estesa Con modalità simili possiamo rappresentare valori booleani, le altre operazioni aritmetiche, condizioni, etc., tuttavia la notazione risulterebbe estremamente complicata. Per semplificare la trattazione, introduciamo, con quello che di fatto è un abuso di notazione, la possibilità di utilizzare numeri, operatori ed espressioni.

Ad esempio, con la nuova notazione, espressioni tipo $\lambda x.(x + 2)$ o $\lambda x.$`if x = 0 then 1 else x + 2`, risulteranno valide.

4.4.3 Iterazione e ricorsione

L'ultima cosa che ci resta da definire per dimostrare che il *λ-calcolo* è un modello di computazione *Turing completo*, è un sistema che consenta l'iterazione. Trattandosi di un modello basato sulle funzioni, la soluzione più ovvia è la ricorsione. Tuttavia, siccome alle funzioni non sono associati nomi, non possiamo semplicemente invocare una funzione all'interno di un'altra, ma siamo costretti a sfruttare i soli concetti di *astrazione* e *applicazione*.

Consideriamo, ad esempio, la seguente definizione ricorsiva della funzione identità:

```
fun f n = if n = 0 then 0 else 1 + f(n);
```

Come si implementa una tale funzione col *λ-calcolo*?

Proviamo a esaminare la definizione sottostante:

$$f = \lambda n.\texttt{if } n \texttt{ = 0 then 0 else 1 + } f(n \texttt{ - 1})$$

A questo punto, abbiamo che $f = G(f)$ dove G è una funzione di ordine superiore che prende una funzione come argomento e restituisce la stessa come risultato.

Valutando la funzione G otteniamo f, tuttavia non sappiamo ancora come è definita G. Non è difficile convincersi la seguente sia una definizione valida:

$$G = \lambda f.\lambda n.\texttt{if } n \texttt{ = 0 then 0 else 1 + } f(n \texttt{ - 1})$$

In questo modo abbiamo reso f un parametro e, di fatto, eliminato la ricorsione.

Ciò che abbiamo fatto corrisponde, in *ML*, a sostituire la definizione data in precedenza con la seguente:

```
fun G f n = if n = 0 then 0 else f(n - 1);
```

Operatori di punto fisso Dalla definizione di G di cui sopra, possiamo derivare la seguente *β-uguaglianza*:

$$f =_\beta Gf$$

Siamo quindi riusciti a ridurre il nostro problema di definire un meccanismo che consenta la ricorsione, alla sola ricerca di una funzione f tale per cui valga quella *β-uguaglianza*.

Una funzione f siffatta è detta essere un *punto fisso* (in inglese *fixpoint*) dell'operatore G e il *λ-calcolo* mette a disposizione dei costrutti che consentono di calcolare il *punto fisso* di un qualunque termine arbitrario. Questi costrutti sono detti *operatori di punto fisso* o *combinatori* e non sono altro che espressioni prive di *variabili libere*.

I seguenti sono 3 esempi di *operatori di punto fisso*:

- *Y-combinator*: usato nei linguaggi funzionali con *valutazione per nome*:

$$Y \stackrel{def}{=} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

- *Z-combinator*: usato nei linguaggi funzionali con *valutazione per valore*:

$$Z \stackrel{def}{=} \lambda f.((\lambda x.(f(\lambda y.(xx)y)))(\lambda x.(f(\lambda y.(xx)y))))$$

- *H-combinator*: usato nei linguaggi funzionali con *valutazione per valore*:

$$H \stackrel{def}{=} \lambda f.((\lambda x.xx)(\lambda x.(f(\lambda y.(xx)y))))$$

Nel resto della trattazione ci concentreremo solo sul *combinatore* Y, ciò che diremo ha però una valenza generale.

Da ciò che abbiamo appena enunciato segue che, se G è un *operatore*, allora $YG = G(YG)$ e quindi YG è un *punto fisso* di G. Vogliamo dimostrare che ciò è valido per qualsiasi espressione G. Dalla definizione abbiamo:

$$Y \stackrel{def}{=} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Applichiamo una generica espressione G ad Y:

$$YG = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))G \rightarrow (\lambda x.G(xx))(\lambda x.G(xx)$$

$$\rightarrow G(\lambda x.G(xx))(\lambda x.G(xx) = G(YG)$$

NB. Il λ-*termine* Y non può essere ridotto ad una *forma normale* e la riduzioni potrebbe continuare all'infinito. Questo è proprio il motivo per cui è usato soltanto dai linguaggi che sfruttano un meccanismo di *valutazione per nome* che riduce il *redex* prima della sua parte argomento. In questo caso, dopo il secondo passaggio di riduzione il *redex* diventa G e quindi, la riduzione di Y termina.

Ora che abbiamo definito dei costrutti che consentono di realizzare la ricorsione, possiamo implementare in λ-*calcolo* la funzione di partenza, che a questo punto possiamo esprimere come:

$$id \stackrel{def}{=} YG$$

Proviamo quindi ad applicare id a un qualche valore:

$$\begin{aligned}
id\ 2 \stackrel{def}{=}\ & (YG)2 \\
\rightarrow\ & G(YG)2 \\
\rightarrow\ & \texttt{if 2 = 0 then 0 else 1 + ((}YG\texttt{)(2 - 1))} \\
\rightarrow\ & \texttt{1 + ((}YG\texttt{) 1)} \\
\rightarrow\ & \texttt{1 + (}G\texttt{(}YG\texttt{) 1)} \\
\rightarrow\ & \texttt{1 + ((if 1 = 0 then 0 else 1 + ((}YG\texttt{)(1 - 1))))} \\
\rightarrow\ & \texttt{1 + (1 + ((}YG\texttt{) 0))} \\
\rightarrow\ & \texttt{1 + (1 + (if 0 = 0 then 0 else 1 + ((}YG\texttt{)(0 - 1))))} \\
\rightarrow\ & \texttt{1 + (1 + 0)} \\
\rightarrow\ & \texttt{2}
\end{aligned}$$

2 è proprio il valore che ci aspettavamo, perciò, siamo riusciti con successo ad implementare una funzione ricorsiva in λ-*calcolo*.

Giunti a questo punto siamo anche riusciti a giustificare quanto affermato in precedenza sulla sua espressività, ovvero abbiamo dimostrato che il λ-*calcolo* descrive un modello di computazione *Turing completo*.