Supporto a Livello di Linguaggio per la Programmazione Funzionale

Luca Abeni

November 6, 2020

1 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 $f : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$[1].

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 programmazione 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*[2] 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[3].

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 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 2. 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 3, dove `f` è una variabile di tipo puntatore a funzione da intero a intero.

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

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

[3] 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.

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

Figure 1: Funzione C che somma 2 interi.

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

  return s;
}
```

Figure 2: Tentativo di currying della funzione `sommainteri()` (Figura 1) usand il linguaggio C.

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 definiti (undefined behaviour). Sebbene qualche semplice test possa funzionare correttamente, l'esempio di Figura 4 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) è mostrata in Figura 5. 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, $f(a,b) \rightarrow f'(a) = f_a : f_a(b) = f(a,b)$). 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 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 6.

Si provi ora ad implementare una funzione `derivata()`, simile alla precedente, ma che ritorna la

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

  f = somma_c(3);

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

  return 0;
}
```

Figure 3: Utilizzo della funzione `somma_c()` di Figura 2.

```
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
}
```

Figure 4: Problema con la funzione `somma_c()` di Figura 2.

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 $f : \mathcal{R} \to \mathcal{R}$ e ritorna una funzione $f' : \mathcal{R} \to \mathcal{R}$; 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 7. I lettori più attenti si saranno sicuramente accorti del fatto che `derivata()` non è nient'altro che la forma "curryificata" di `calcoladerivata()`[4]!

A titolo di esempio, in Figura 8 si riporta l'implementazione di `derivata()` in C++ (usando le estensioni funzionali fornite da C++11, come le lambda function). 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 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).

2 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

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

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

  return s;
}
```

Figure 5: Currying corretto della funzione `sommainteri()` (Figura 1) 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;
}
```

Figure 6: 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;
}
```

Figure 7: Calcolo della derivata di una funzione.

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...).

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

```
#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;
}
```

Figure 8: Calcolo della derivata di una funzione in C++.

in un ambiente globale. Un linguaggio funzionale fornisce quindi:

- Un insieme di tipi predefiniti e di valori per questi tipi, più un insieme di operatori per combinare valori dei vari tipi costruendo espressioni. Il sistema di tipi utilizzato dal linguaggio può variare, facendo controlli più o meno stretti, a tempo di esecuzione o a tempo di compilazione;
- 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...

E' interessante notare come i linguaggi di programmazione imperativi (a cui il lettore è probabilmente più abituato) forniscono un unico meccanismo di definizione di funzioni 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 function del C++, accennato in precedenza) ed un secondo meccanismo che permette di modificare l'ambiente 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.

Basandosi sui meccanismi descritti qui sopra, la computazione per riduzione può essere 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 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*). Tali valori possono essere valori di tipi riconosciti dal linguaggio o valori funzionali.

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*"[5]. 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...).

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)` definita 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:

[5] 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.

- 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).

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.

Per concludere, i lettori più curiosi si possono chiedere 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)
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".