Fixpoint ed Altre Amenità

Luca Abeni

December 1, 2020

1 Fixed Point Combinator

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

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

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

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

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

$$Fix(g) = g(Fix(g)) \Rightarrow Fix = \lambda g.g(Fix(g)). \tag{1}$$

Come prima considerazione, è interessante notare come se si prova a valutare l'equazione 1 usando una strategia *eager* (valutazione per valore), si ottiene

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

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

Un'altra osservazione importante è che la funzione Fix definita in Equazione 1 permette (per costruzione) di calcolare il punto fisso del suo argomento g, ma non è un combinator: in particolare, la definizione di Fix usa la variabile Fix che è libera, non essendo legata da alcuna λ (mentre la definizione di un combinator dovrebbe contenere solo variabili legate). La cosa è particolarmente rilevante perché Fix è proprio il nome della funzione che si sta definendo, quindi la definizione di Fix è ricorsiva. In altre parole, abbiamo solo spostato la ricorsione dalla definizione di g a quella di Fix.

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

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

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

Il più famoso fra i fixed-point combinator è l'**Y combinator**, sviluppato da Haskell Curry (eh si, i nomi alla fine sono sempre gli stessi...), la cui definizione (usando il λ calcolo) è:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)). \tag{2}$$

Si noti che in letteratura esistono alcune piccole inconsistenze a livello terminologico: mentre in generale l'Y combinator è *un particolare* fixed point combinator, alcuni tendono ad usare il termine "Y combinator" per identificare un generico fixed point combinator (e scrivono quindi che esiste un'infinità di Y combinator).

2 Implementazione in SML

Il Y Combinator è definito come

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Provando una semplice conversione nella sintassi SML ("λ" diventa "`fn`" e "." diventa "`=>`") si ottiene

```
val Y = fn f => (fn x => f (x x))(fn x => f (x x))
```

Questa espressione non è però accettata da un compilatore o interprete SML. Per esempio, Poly/ML genera il seguente errore:

```
> val Y = fn f => (fn x => f (x x))(fn x => f (x x));
poly: : error: Type error in function application.
    Function: x : 'a -> 'b
    Argument: x : 'a -> 'b
    Reason:
       Can't unify 'a to 'a -> 'b (Type variable to be unified occurs in type)
Found near (fn x => f (x x)) (fn x => f (x x))
poly: : error: Type error in function application.
    Function: x : 'a -> 'b
    Argument: x : 'a -> 'b
    Reason:
       Can't unify 'a to 'a -> 'b (Type variable to be unified occurs in type)
Found near (fn x => f (x x)) (fn x => f (x x))
Static Errors
```

Il significato di questo errore diventa chiaro cercando di capire qual'è il tipo di "`x`". Supponendo che il tipo di "`x`" sia "`T`", si ha che:

- Poiché "`(x x)`" indica che "`x`" è *una funzione* applicata ad un argomento, il tipo "`T`" di "`x`" deve essere una funzione. Consideriamo il caso più generico: una funzione $\alpha \to \beta$, quindi $\texttt{T} = \alpha \to \beta$
- D'altra parte, la funzione "`x`" è applicata proprio all'argomento "`x`". Quindi il tipo α dell'argomento della funzione "`x`" deve essere uguale al tipo di "`x`" ("`T`"): $\texttt{T} = \alpha$.

Mettendo tutto assieme si ottiene

$$\texttt{T} = \alpha \to \beta \wedge \texttt{T} = \alpha \Rightarrow \alpha = \alpha \to \beta$$

che definisce il tipo "`T`" in modo ricorsivo. Poiché Standard ML supporta tipi di dati ricorsivi, verrebbe quindi da pensare che dovrebbe essere in grado di supportare il tipo "`T`" di cui stiamo parlando. Invece, una ricorsione come $\alpha = \alpha \to \beta$ non ha molto senso in un linguaggio che fa un controllo stretto dei tipi come Standard ML. Per capire come mai, è necessario entrare un po' più nei dettagli dei meccanismi di definizione di tipi ricorsivi.

Di fronte ad una definizione come $\alpha = \alpha \to \beta$, si parla di tipi *equi-ricorsivi* (equi-recursive data types) se α ed $\alpha \to \beta$ rappresentano lo stesso tipo di dati (stessi valori, stesse operazioni sui valori, etc...). Si parla invece di tipi *iso-ricorsivi* (iso-recursive data types) se il tipo α ed il tipo $\alpha \to \beta$ non sono uguali, ma esiste un isomorfismo (funzione $1 \to 1$, invertibile, che ad ogni valore di α associa un valore di $\alpha \to \beta$ e viceversa) dall'uno all'altro. Questo isomorfismo (che stabilisce che i due tipi sono in sostanza equivalenti) è la funzione che in Standard ML abbiamo chiamato "costruttore". Capendo che Standard ML supporta quindi tipi iso-ricorsivi (un valore di α può essere generato tramite un apposito costruttore a partire da

un valore di $\alpha \to \beta$), ma non equi-ricorsivi (α ed $\alpha \to \beta$ **non** possono essere lo stesso tipo!) diventa allora chiaro perché la definizione di `Y` data poco fa generi un errore sintattico (in particolare, un errore di tipo).

Si noti che questo non significa che la definizione dell'Y combinator (Equazione 2) sia "sbagliata", ma semplicemente che non è implementabile direttamente in Standard ML. L'Y combinator è definito usando il λ-calcolo, in cui ogni identificatore è legato ad una funzione, il cui tipo non è importante. Utilizzando linguaggi con controlli sui tipi "meno stretti" rispetto a Standard ML (per esempio, Lisp, Scheme, Python, Javascript, etc...) o linguaggi che supportano tipi equi-ricorsivi (per esempio, OCaml con apposite opzioni), l'Equazione 2 è implementabile senza problemi.

Per capire meglio come risolvere questo problema, consideriamo la "parte problematica" dell'espressione precedente (l'applicazione di funzione "`(x x)`"), limitandoci per il momento alla funzione (chiaramente di ordine superiore) "`fn x => (x x)`.

Poiché in Standard ML un tipo di dato ricorsivo `T` può essere definito usando il costrutto `datatype` ed (almeno) un costruttore che mappa valori di un tipo che dipende da `T` in valori di `T` (definizione di tipo iso-ricorsivo), si può definire qualcosa dei tipo $T = F(T \to \beta)$ per "simulare" il tipo (equi-ricorsivo) $\alpha = \alpha \to \beta$ di `x`. Il dipo `T` risultante sarà quindi funzione del tipo β ed in Standard ML questo si indica con "`'b T`".

Ricordandosi la sintassi di `datatype` (un tipo di dato dipendente dalla type variable β in Standard ML si definisce usando `datatype 'b <nometipo> = ...`), il tipo di dato che ci interessa è

```
datatype 'b T = F of ('b T -> 'b)
```

dove come detto "`'b T`" è il nome del tipo, mentre "`F`" è il nome del costruttore che mappa valori di `'b T -> 'b` in valori di `'b T`.

A questo punto, è possibile usare "`'b T`" come tipo di "`x`" facendo si che "`x`" sia una funzione `'b T -> 'b` da `'b T` a `'b`. L'argomento (parametro attuale) di tale funzione deve quindi essere di tipo `'b T`, generabile da "`x`" usando il costruttore `F`. Invece di "**fn** x => x x" si scrive allora:

```
fn x => (x (F x))
```

E standard ML è perfettamente in grado di capire e processare questa definizione. Si noti come l'imposisbilità di usare tipi equi-ricorsivi ci ha obbligati ad usare il costruttore `F`. Il tipo della funzione definita qui sopra risulta essere `('a T -> 'a) -> 'a` (funzione che mappa valori "`x`" di tipo "`'a T -> 'a`" in valori di tipo "`'a`").

Ora che abbiamo visto come risolvere il problema del tipo di "`x`", è possibile tornare all'espressione originaria del Y combinator, che presenta ancora un problema analogo: "`fn f(x (F x))`" va applicata a se stessa, quindi il suo tipo è ricorsivo come il tipo di "`x`"! Ancora una volta il problema può essere risolto usando il tipo di dato "`'b T`" precedentemente definito: "**fn** f => (**fn** x => f (x (F x)))" ha tipo "`('a -> 'b) -> ('a T -> 'a) -> 'b`", quindi il valore a cui è applicata deve avere tipo "`('a -> 'b) -> ('a T -> 'a)`... Bisogna quindi sostituire un "`('a T -> 'a)`" (tipo di "`x`") con un "`'a T`" (ottenibile da "`x`" applicandogli il costruttore "`F`"). L'argomento sarà quindi "**fn** f => (**fn** (F x) => f (x (F x)))"

Come risultato, l'espressione dell'Y combinator è:

```
val Y = fn f => (fn x => f(x (F x)))(fn (F x) => f(x (F x)))
```

e questa espressione è accettata da SML! Riassumendo, la prima parte dell'espressione ("`fn x => f(x (F x))`") ha tipo "`('a T -> 'a) -> 'b`", mentre la seconda parte ("`fn (F x) => f(x (F x))`") ha tipo "`'a T -> 'b`" ed è quindi usabile come argomento per la prima semplicemente ponendo $\beta = \alpha$.

Il tipo della funzione risultante è "`('a -> 'a) -> 'a`", dove "`'a`" è chiaramente un tipo funzione (per il fattoriale, per esempio, è una funzione "`int -> int`").

La discussione di qui sopra, che mostra come sia possibile implementare il fixed point combinator Y in linguaggi (come Standard ML) con tipizzazione stretta che non supportano tipi di dati equi-ricorsivi, ci permette di intuire una cosa importante: Y "elimina la ricorsione" dall'operatore fixed point (l'Equazione 1 utilizza ricorsione esplicita) spostando la ricorsione sul tipo di dato (richiede infatti di usare tipi di dati ricorsivi di qualche genere). Questo non è immediatamente visibile nell'Equazione 2 o in implementazioni di Y in scheme (o simili), ma diventa più chiaro provando ad implementare Y in (per esempio) Standard ML.

Una volta che ML ha "accettato e capito" la nostra definizione di Y, potrebbe sembrare che tutti i problemi siano risolti. Testando il combinator Y definito sopra, però, si incontrano altre brutte sorprese. Proviamo per esempio ad utilizzare la funzione Y appena definita per definire la funzione fattoriale. Prima

di tutto, definiamo una versione "chiusa" `fact_closed`[3] della funzione fattoriale come funzione come

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

A questo punto, si potrebbe pensare di calcolare la funzione fattoriale

```
val fact = Y fact_closed
```

ma questo porta ad una computazione divergente (ricorsione infinita).

Prima di vedere come risolvere questo problema, si noti che il tipo di "`fact_closed`" è "`(int -> int) -> (int -> int)`", quindi, poiché Y ha tipo "`('a -> 'a) -> 'a`" "`Y fact_closed`" ha tipo "`int -> int`".

Veniamo ora a risolvere il problema della non convergenza di "`Y fact_closed`". Ancora una volta, il fatto che la nostra implementazione non fornisca il risultato sperato in Standard ML non significa che l'Y combinator "è sbagliato". Il "problema" è semplicemente dovuto al fatto che l'Y combinator non converge in caso di valutazione *eager* (o *per valore*), che è la strategia usata da Standard ML (si ricordi la prima osservazione sull'Equazione 1 nella Sezione 1). In altre parole, in un linguaggio che valuta le espressioni per valore dobbiamo usare un altro tipo di fixed point combinator. Fra i vari fixed point combinator che funzionano correttamente usando la valutazione eager (quindi anche in Standard ML), il più famoso è probabilmente il cosiddetto "Z combinator". La sua definizione è

$$Z = \lambda f.(\lambda x.f(\lambda v.((xx)v)))(\lambda x.f(\lambda v.((xx)v)))$$

e come si vede può essere ottenuto dall'Y combinator astraendo l'applicazione "`(x x)`" rispetto ad una variabile "`v`". Informalmente, si può dire che questa nuova astrazione impedisce ad un valutatore eager di entrare in un ciclo infinito di riduzioni.

La traduzione in SML a questo punto è semplice:

```
val Z = fn f =>
        (fn x => f (fn v => (x (F x))v))(fn (F x) => f (fn v => (x (F x))v))
```

ed il tipo di "`Z`" è ora "`(('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)`" (alcune parentesi le ho aggiunte io; Poly/ML stampera' invece "`(('a -> 'b) -> 'a -> 'b) -> 'a -> 'b`") indicando esplicitamente che il tipo prima indicato come "`'a`" (α) è in realtà una funzione "`'a -> 'b`". Stavolta, il combinator funziona correttamente:

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

3 Implementazione in Haskell

Il Y Combinator è definito come

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Provando una semplice conversione nella sintassi Haskell ("λ" diventa "\" e "." diventa "`->`") si ottiene

```
y = \f -> (\x -> f (x x))(\x -> f (x x))
```

Questa espressione non è però accettata da un compilatore o interprete Haskell. Per esempio, `ghci` genera il seguente errore:

```
Prelude> y = \f -> (\x -> f (x x))(\x -> f (x x))

<interactive>:1:23: error:
    o Occurs check: cannot construct the infinite type: t0 ~ t0 -> t
      Expected type: t0 -> t
```

[3] Questa funzione viene detta "chiusa" perché non ha variabili libere. La tradizionale definizione ricorsiva della funzione fattoriale usa invece una variabile libera (il suo nome) per richiamarsi ricorsivamente.

```
      Actual type: (t0 -> t) -> t
  o In the first argument of 'x', namely 'x'
    In the first argument of 'f', namely '(x x)'
    In the expression: f (x x)
  o Relevant bindings include
      x :: (t0 -> t) -> t (bound at <interactive>:1:13)
      f :: t -> t (bound at <interactive>:1:6)
      y :: (t -> t) -> t (bound at <interactive>:1:1)
...
```

Il significato di questo errore diventa chiaro cercando di capire qual'è il tipo di "`x`". Supponendo che il tipo di "`x`" sia "`t0`", si ha che:

- Poiché "`(x x)`" indica che "`x`" è *una funzione* applicata ad un argomento, il tipo "`t0`" di "`x`" deve essere una funzione. Consideriamo il caso più generico: una funzione $\alpha \to \beta$, quindi $\texttt{t0} = \alpha \to \beta$
- D'altra parte, la funzione "`x`" è applicata proprio all'argomento "`x`". Quindi il tipo α dell'argomento della funzione "`x`" deve essere uguale al tipo di "`x`" ("`t0`"): $\texttt{t0} = \alpha$.

Mettendo tutto assieme si ottiene

$$\texttt{t0} = \alpha \to \beta \wedge \texttt{t0} = \alpha \Rightarrow \alpha = \alpha \to \beta$$

che definisce il tipo "`t0`" in modo ricorsivo (come indicato dal messaggio di errore, "`t0`" deve essere equivalente a "`t0 -> t`"). Poiché Haskell supporta tipi di dati ricorsivi, verrebbe quindi da pensare che dovrebbe essere in grado di supportare il tipo "`t0`" di cui stiamo parlando. Invece, una ricorsione come $\alpha = \alpha \to \beta$ non ha molto senso in un linguaggio che fa un controllo stretto dei tipi come Haskell. Per capire come mai, è necessario entrare un po' più nei dettagli dei meccanismi di definizione di tipi ricorsivi.

Di fronte ad una definizione come $\alpha = \alpha \to \beta$, si parla di tipi *equi-ricorsivi* (equi-recursive data types) se α ed $\alpha \to \beta$ rappresentano lo stesso tipo di dati (stessi valori, stesse operazioni sui valori, etc...). Si parla invece di tipi *iso-ricorsivi* (iso-recursive data types) se il tipo α ed il tipo $\alpha \to \beta$ non sono uguali, ma esiste un isomorfismo (funzione $1 \to 1$, invertibile, che ad ogni valore di α associa un valore di $\alpha \to \beta$ e viceversa) dall'uno all'altro. Questo isomorfismo (che stabilisce che i due tipi sono in sostanza equivalenti) è la funzione che in abbiamo chiamato "costruttore" parlando di tipi di dati algebrici. Capendo che Haskell supporta quindi tipi iso-ricorsivi (un valore di α può essere generato tramite un apposito costruttore a partire da un valore di $\alpha \to \beta$), ma non equi-ricorsivi (α ed $\alpha \to \beta$ **non** possono essere lo stesso tipo!) diventa allora chiaro perché la definizione di `Y` data poco fa generi un errore sintattico (in particolare, un errore di tipo).

Si noti che questo non significa che la definizione dell'Y combinator (Equazione 2) sia "sbagliata", ma semplicemente che non è implementabile direttamente in Haskell (come in ogni altro linguaggio che utilizzi una tipizzazione forte). L'Y combinator è definito usando il λ-calcolo, in cui ogni identificatore è legato ad una funzione, il cui tipo non è importante. Utilizzando linguaggi con controlli sui tipi "meno stretti" rispetto a Haskell (per esempio, Lisp, Scheme, Python, Javascript, etc...) o linguaggi che supportano tipi equi-ricorsivi (per esempio, OCaml con apposite opzioni), l'Equazione 2 è implementabile senza problemi.

Per capire meglio come risolvere questo problema, consideriamo la "parte problematica" dell'espressione precedente (l'applicazione di funzione "`(x x)`"), limitandoci per il momento alla funzione (chiaramente di ordine superiore) "`\x -> (x x)`".

Poiché in Haskell un tipo di dato ricorsivo `t0` può essere definito usando il costrutto `data` ed (almeno) un costruttore che mappa valori di un tipo che dipende da `t0` in valori di `t0` (definizione di tipo iso-ricorsivo), si può definire qualcosa dei tipo $T = F(T \to \beta)$ per "simulare" il tipo (equi-ricorsivo) $\alpha = \alpha \to \beta$ di `x`. Il dipo `T` risultante sarà quindi funzione del tipo β ed in Haskell questo si indica con "`T b`".

Ricordandosi la sintassi di `data` (un tipo di dato dipendente dalla type variable β in Haskell si definisce usando `data <nometipo> b = ...`), il tipo di dato che ci interessa è

```
data T b = F (T b -> b)
```

dove come detto "`T b`" è il nome del tipo, mentre "`F`" è il nome del costruttore che mappa valori di `T b -> b` in valori di `T b`.

A questo punto, è possibile usare "`T b`" per tipizzare correttamente "`x`" facendo si che "`x`" sia una funzione `T b -> b` da `T b` a `b`. L'argomento (parametro attuale) di tale funzione deve quindi essere di tipo `T b`, generabile da "`x`" usando il costruttore `F`. Invece di "\x -> x x" si scrive allora:

```
\x -> (x (F x))
```

Ed Haskell è perfettamente in grado di capire e processare questa definizione. Si noti come l'imposisbilità di usare tipi equi-ricorsivi ci ha obbligati ad usare il costruttore `F`. Il tipo della funzione definita qui sopra risulta essere `(T b -> b) -> b` (funzione che mappa valori "`x`" di tipo "`T b-> b`" in valori di tipo "`b`").

Ora che abbiamo visto come risolvere il problema del tipo di "`x`", è possibile tornare all'espressione originaria del Y combinator, che presenta ancora un problema analogo: "`\x -> f(x (F x))`" va applicata a se stessa, quindi il suo tipo è ricorsivo come il tipo di "`x`"! Ancora una volta il problema può essere risolto usando il tipo di dato "`T b`" precedentemente definito: "\f -> (\x -> f (x (F x)))" ha tipo "`(b -> c) -> (T b -> b) -> c`", quindi assumendo "`b -> c`" come tipo per "`f`" si ha che "`\x -> f (x (F x))`" ha tipo "`(T b -> b) -> c`"... Il valore "`\x -> f (x (F x))`" a cui è applicata deve quindi avere tipo "`Tb -> b`". Bisogna quindi sostituire un "`(T b -> b)`" (tipo di "`x`") con un "`T b`" (ottenibile da "`x`" applicandogli il costruttore "`F`"). L'argomento sarà quindi "\f -> (\ (F x) -> f (x (F x)))"

Come risultato, l'espressione dell'Y combinator dovrebbe essere:

```
y = \f -> (\x -> f(x (F x)))(\ (F x) -> f(x (F x)))
```

ed in teoria questa espressione dovrebbe essere accettata da Haskell! Sfortunatamente, però, `ghc` (e quindi `ghci`) ha dei problemi ad inferire correttamente i tipi di dato[4]. Altri programmi, come l'interprete Hugs (`https://www.haskell.org/hugs`) riescono a parsare e valutare questa definizione senza problemi.

Il problema incontrato da `ghc` può essere superato "aiutando" in qualche modo il compilatore ad inferire correttamente i tipi delle varie sottoespressioni. Per esempio, si potrebbe sostituire "`F x`" con una variabile "`z`" del giusto tipo (`T b`). Per fare questo, però, è necessaria una funzione che permetta di estrarre "`x`" da "`F x`"; in pratica, la funzione inversa del costruttore "`F`":

```
invF (F x) = x
```

A questo punto, è possibile sostituire "`\(F x)`" con "`\z`" e la seguente "`x`" con "`invF z`", ottenendo

```
y = \f -> (\x -> f(x (F x)))(\z -> f((invF z) z))
```

e questa espressione è accettata anche da `ghc`/`ghci`! Riassumendo, la prima parte dell'espressione ("`\x -> f(x (F x))`") ha tipo "`(T b -> b) -> c`", mentre la seconda parte ("`\z -> f((invF z) z)`") ha tipo "`T b -> c`" ed è quindi utilizzabile come argomento per la prima semplicemente ponendo $\beta = \gamma$.

Il tipo della funzione risultante è "`(b -> b) -> b`", dove "`b`" è chiaramente un tipo funzione (per il fattoriale, per esempio, è una funzione "`Int -> Int`").

La discussione di qui sopra, che mostra come sia possibile implementare il fixed point combinator Y in linguaggi (come Haskell) con tipizzazione stretta che non supportano tipi di dati equi-ricorsivi, ci permette di intuire una cosa importante: Y "elimina la ricorsione" dall'operatore fixed point (l'Equazione 1 utilizza ricorsione esplicita) spostando la ricorsione sul tipo di dato (richiede infatti di usare tipi di dati ricorsivi di qualche genere). Questo non è immediatamente visibile nell'Equazione 2 o in implementazioni di Y in scheme (o simili), ma diventa più chiaro provando ad implementare Y in (per esempio) Haskell.

Una volta che Haskell ha "accettato e capito" la nostra definizione di Y, si può, per esempio, utilizzare la funzione Y appena definita per definire la funzione fattoriale. Prima di tutto, definiamo una versione "chiusa" `fact_closed`[5] della funzione fattoriale come funzione come

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

A questo punto, si può calcolare la funzione fattoriale come

```
fact = y fact_closed
```

Si noti che il tipo di "`fact_closed`" è "`(Eq p, Num p) => (p -> p) -> p -> p`", quindi, poiché Y ha tipo "`(b -> b) -> b`" "`y fact_closed`" ha tipo "`(Eq p, Num p) => p -> p`".

Riassumendo, una possibile definizione del Y combinator in Haskell (ne esistono molte altre) può essere:

```
data T b = F (T b -> b)
invF (F x) = x
y = \f -> (\x -> f(x (F x)))(\y -> f((invF y) y))
```

Basandosi su questa definizione si può, per esempio, usare `ghci` per fare:

[4] Questo è probabilmente legato ad un bug descritto in `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/bugs.html`.

[5] Questa funzione viene detta "chiusa" perché non ha variabili libere. La tradizionale definizione ricorsiva della funzione fattoriale usa invece una variabile libera (il suo nome) per richiamarsi ricorsivamente.

```
Prelude> fact_closed = \f -> \n -> if n == 0 then 1 else n * f (n - 1)
Prelude> :t fact_closed
fact_closed :: (Eq p, Num p) => (p -> p) -> p -> p
Prelude> fact = y fact_closed
Prelude> :t y
y :: (b -> b) -> b
Prelude> :t fact
fact :: (Eq p, Num p) => p -> p
Prelude> fact 3
6
Prelude> fact 4
24
...
```

4 Come Derivare l'Y Combinator

Come detto nelle sezioni precedenti, l'Y combinator è definito come $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$; fino ad ora non è però chiaro da cosa derivi tale espressione.

Si consideri una generica funzione ricorsiva f (definita per semplicità sull'insieme dei numeri naturali $\mathcal{N}$) il cui passo induttivo è definito come $f(n+1) = g(f(n), n)$. La classica implementazione ricorsiva di tale funzione andraà ad invocare ricorsivamente $f()$ nel passo induttivo, risultando quindi essere una funzione non chiusa (si ricordi che una funzione si definisce chiusa quando non usa variabili esterne).

Per esempio, si consideri la funzione fattoriale, definita (usando per semplicità un λ-calcolo esteso, che permette di usare numeri, operazioni aritmetiche ed if logici [6]) come

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

Si può immediatamente vedere come l'espressione "if $n = 0$ then 1 else $n \cdot f(n-1)$" utilizzi la variabile libera (non legata) f.

Una generica funzione ricorsiva $f(n+1) = g(f(n), n)$ può essere descritta tramite una sua "versione chiusa" f_{closed} che non usa variabili libere, ma riceve come parametro la funzione da chiamare ricorsivamente. Tale funzione può essere definita come $f_{closed} = \lambda f.\lambda n. < expr >$ (dove "$< expr >$ è un'espressione chiusa, in quanto la "f" è stata legata da un "$\lambda f.$"). Come noto, la funzione ricorsiva originaria "f" è un punto fisso della funzione chiusa f_{closed}: $f = f_{closed}f$ ed assumendo di disporre di un fixed point combinator Y si può calcolare f come

$$f = Y f_{closed}$$

(in altre parole, la funzione chiusa f_{closed} è una versione non ricorsiva di f, che riceve come primo argomento la funzione f da invocare ricorsivamente: $f(n) = (f_{closed}f)n \Leftrightarrow f = f_{closed}f$).

Tornando all'esempio della funzione fattoriale di qui sopra, la sua "versione chiusa" f_{closed} è ovviamente

$$f_{closed} = \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1).$$

Si noti che la definizione di f_{closed} è un primo passo verso l'eliminazione della ricorsione, ma ancora non è sufficiente: f_{closed} riceve infatti come primo argomento la funzione f che stiamo cercando di definire. Si può allora definire una funzione G che come f_{closed} riceve come primo argomento una funzione da invocare ricorsivamente, ma a differenza di f_{closed} invoca tale funzione passando la funzione stessa come primo argomento. In altre parole, mentre il parametro formale di f_{closed} è una funzione $f : \mathcal{N} \to \mathcal{N}$, il primo parametro di G è una funzione che riceverà 2 argomenti: funzione da richiamare ricorsivamente e numero intero. G si può quindi ottenere da f_{closed} semplicemente sostituendo tutte le invocazioni di f con ff. Tornando all'esempio del fattoriale, si ottiene:

$$G = \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot ff(n-1). \tag{3}$$

Riassumendo, f_{closed} è tale che $f = f_{closed}f$, mentre G è tale che $f = GG$!!! I lettori più attenti si saranno sicuramente accorti del fatto che questo è il momento in cui si passa dalla "ricorsione sul

[6] Si ricordi che anche se numeri, operazioni aritmetiche, if logici e costrutti simili non fanno tecnicamente parte del λ-calcolo puro, possono essere implementati usando λ-espressioni pure.

flusso di esecuzione" (definizione ricorsiva di f, come funzione non chiusa) alla ricorsione sul tipo di dati (definizione di una funzione G che è chiusa, ma ha necessità di potersi applicare a se stessa - self application). Per capire meglio la cosa, si provi a trovare il tipo della funzione G...

Con un minimo di manipolazione (una "β-riduzione al contrario") si ottiene che G è equivalente a $\lambda f.(f_{closed}(ff))$: infatti, come detto il corpo di G si ottiene dal corpo di f_{closed} sostituendo f con ff (vale a dire, applicando f_{closed} ad ff). Si noti come l'utilizzo di questa "β-riduzione al contrario" implica che (come già noto) il Y combinator che stiamo andando a ricavare lavora per β-equivalenza e non per β-riduzione (in altre parole: $Yf_{closed} \equiv_\beta f_{closed}(Yf_{closed})$, ma $Yf_{closed} \not\to_\beta f_{closed}(Yf_{closed})$).

Tornando all'esempio del fattoriale, si ha

$$G \equiv_\beta \lambda f.(\lambda h.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot h(n-1))(ff)$$

in quanto se si applica $\lambda h.\lambda n.$if $n = 0$ then 1 else $n \cdot h(n-1)$ a (ff) si ottiene l'espressione di G definita in Equazione 3. Ora, si può vedere come questa espressione contenga al suo interno la definizione di f_{closed}: l'espressione $\lambda h.\lambda n.$if $n = 0$ then 1 else $n \cdot h(n-1)$ dentro le parentesi che viene applicata ad (ff) è proprio f_{closed}! Riassumendo,

$$G \equiv_\beta \lambda f.(f_{closed}(ff)).$$

Ricordando ora che $f = GG$, ed applicando un'altra "β-riduzione al contrario" si ha $f = GG \leftarrow_\beta (\lambda h.(hh))G$; sostituendo a questo punto G $\lambda f.f_{closed}(ff)$ (che è β equivalente a G) si ottiene

$$f \equiv_\beta (\lambda h.(hh))(\lambda f.f_{closed}(ff))$$

La solita β-riduzione al contrario (astraendo rispetto ad funzione g ed applicando ad f_{closed} in modo da ottenere l'espressione di qui sopra) ci porta a

$$f \equiv_\beta (\lambda g.(\lambda h.(hh))(\lambda f.g(ff)))f_{closed}$$

ed una semplice α-equivalenza porta a

$$f \equiv_\beta (\lambda g.(\lambda h.(hh))(\lambda h.g(hh)))f_{closed}$$

A questo punto, ricordando che per definizione $f \equiv_\beta Yf_{closed}$ si ottiene $Y = \lambda g.(\lambda h.(hh))(\lambda h.g(hh))$!!!

Si noti che l'espressione di Y non è ancora quella vista in precedenza, ma la definizione "tradizionale" si può ottenere tramite un passo di β-riduzione: applicando $\lambda h.(hh)$ a $\lambda h.g(hh)$ si ottiene finalmente $Y = \lambda g.(\lambda h.g(hh))(\lambda h.g(hh))$.

Per capire come si deriva lo Z combinator (che possiamo vedere come una versione dell'Y combinator che funziona con valutazione eager), si consideri il passaggio da

$$G = \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot ff(n-1).$$

a

$$\lambda f.(\lambda h.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot h(n-1)(ff)).$$

Il problema sorge quando "f" è sostituita con una funzione "$\lambda g....$": mentre la prima espressione può essere valutata senza divergere anche da linguaggi eager (perché il redex "$(\lambda g....)(\lambda g....)$" che si viene a creare sta dentro ad un'astrazione "$\lambda n....$"), la seconda cerca di valutare il redex che costituisce l'argomento "$(\lambda g....)(\lambda g....)$" (che non si trova più "protetto") da un'astrazione "$\lambda h....$") prima di applicargli la funzione f_{closed} ("$\lambda h.\lambda n.$if $n = 0$ then 1 else $n \cdot h(n-1)$" nel caso del fattoriale). Ma la riduzione di questo redex "$(\lambda g....)(\lambda g....)$" non convergerà (perché l'argomento "g" sarà espanso a qualcosa che richiama la funzione stessa), portando ad una ricorsione infinita...

Per evitare questa ricorsione infinita bisogna evitare che la macchina astratta cerchi di ridurre "(ff)" in ogni caso (ma vada a valutare tale espressione solo quando necessario). Questo risultato può essere ottenuto "proteggendo" il termine "(ff)" tramite un'astrazione rispetto ad un argomento v ed applicando poi il risultato a v: $(ff) \to \lambda v.(ff)v$. Da cui si ha:

$$\lambda f.(\lambda h.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot h(n-1))(\lambda v.(ff)v).$$

Ripercorrendo i passi precedenti, questo porta a

$$G \equiv_\beta \lambda f.f_{closed}(\lambda v.(ff)v)$$

e quindi

$$f \equiv_\beta (\lambda h.(hh))(\lambda h.f_{closed}(\lambda v.(hh)v))$$

da cui

$$Z = \lambda f.(\lambda h.(hh))(\lambda h.f_{closed}(\lambda v.(hh)v)).$$

Con un passo di β-riduzione si ottiene

$$Z = \lambda f.(\lambda h.g(\lambda v.(hh)v))(\lambda h.g(\lambda v.(hh)v)).$$

che è l'espressione più tradizionalmente conosciuta.