Due Parole su Lambda Calcolo

Luca Abeni

March 29, 2023

1 Introduzione al Lambda-Calcolo

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

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

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

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

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

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

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

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

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

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

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

Questo è equivalente alla seguente definizione induttiva:

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

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

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

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

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

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

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

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

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

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

2 Semantica del Lambda Calcolo

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

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

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

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

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

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

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

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

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

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

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

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

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

$$(\lambda x.E)E_1 \to_\beta E[x \to E_1]$$

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

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

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

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

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

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

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

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

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

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

$$\Omega = \omega\omega = (\lambda x.xx)(\lambda x.xx) \to_\beta (xx)[x \to (\lambda x.xx)] = (\lambda x.xx)(\lambda x.xx) = \omega\omega = \Omega$$

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

3 Codifica di Linguaggi di Alto Livello

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Analogamente, si può calcolare la codifica di 2:

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

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

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

L'esercizio può essere ripetuto per altri numeri naturali.

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

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

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

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

1. Prima di tutto, si applica n a f e x per rimuovere le astrazioni $\lambda f.\lambda x.$, analogamente a quanto fatto per la codifica di *succ*
2. Poi, si applica m a f per rimuovere l'astrazione $\lambda f.$
3. A questo punto, applicando il risultato di mf (vale a dire, "m" senza "λf") al risultato di $((nf)x)$ (che è "$f^n x$") si aggiungono m "$f($" alla sinistra di "$f^n x$". Come risultato si ottiene "$f^{n+m}x$"
4. Come fatto per *succ*, si astrae di nuovo rispetto a f e x per aggiungere il $\lambda f.\lambda x.$ rimosso al passo 1
5. Per finire, si astrae rispetto a n e m, in modo da ottenere un combinator

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

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

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

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

e

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

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

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

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

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

  return res;
}
```

Figure 1: Implementazione iterativa della funzione `fattoriale()`.

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

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

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

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

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

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

A livello di esempio, vediamo come utilizzare il fixed point combinator Y per calcolare la funzione fattoriale. Ricordiamo che una possibile implementazione imperativa del fattoriale è mostrata in Figura 1, mentre la tradizionale implementazione ricorsiva è mostrata in Figura 2. Riscritta secondo il paradigma funzionale, tale funzione appare come in Figura 3. Un primo tentativo (non troppo riuscito, per il vero) di conversione in λ-espressione potrebbe essere

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

Si noti che questa funzione, che potrebbe apparire strana quando si parla di λ-calcolo puro (in quanto

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

Figure 2: Implementazione ricorsiva della funzione `fattoriale()`.

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

Figure 3: Implementazione funzionale della funzione `fattoriale()`.

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

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

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

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

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

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

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

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

4 Rimozione delle Astrazioni

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

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

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

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

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

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

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

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

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

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

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

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

Si lascia al lettore la dimostrazione che le definizioni dei combinator S e K presentate nella Sezione 3 ($S = \lambda f.\lambda g.\lambda x.fx(gx)$ e $K = \lambda x.\lambda y.x$) godono delle proprietà descritte qui sopra. E' quindi semplice convertire un'espressione del calcolo SK (o del calcolo SKI) in una λ-espressione. Poiché è anche possibile convertire ogni λ-espressione in un'espressione del calcolo SK, il calcolo SK ha lo stesso potere espressivo del λ calcolo ed è quindi Turing-completo.

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

- Un identificatore x, che nell'espressione x del calcolo SK
- Un'applicazione $E_1 E_2$, che si mappa nell'espressione $E_1 E_2$ del calcolo SK
- Un'astrazione $\lambda x.E'$, che va convertita in un'espressione E'' del calcolo SK procedendo ancora per casi:
  - Se E' è un identificatore, può essere x, nel qual caso $E = \lambda x.x$ si converte in $E'' = I = SKK$, oppure un simbolo $y \neq x$, nel qual caso $E = \lambda x.y$ si converte in $E'' = Ky$
  - Se E' è un'applicazione $E'_1 E'_2$, $E = \lambda x.E'_1 E'_2$ va convertita E'' tale che $(\lambda x.E'1E'_2)v = E''v$. Questo comporta che

$$(\lambda x.E'_1 E'_2)v = E''v \Rightarrow E'_1[x \to v]E'_2[x \to v] = E''v \Rightarrow$$

$$(\lambda x.E'_1 v)(\lambda x.E'_2 v) = E''v \Rightarrow S(\lambda x.E'_1)(\lambda x.E'_2)v = E''v \Rightarrow$$

$$E'' = S(\lambda x.E'_1)(\lambda x.E'_2)$$

  - Se E' è un'astrazione $\lambda y.E'_1$, $E = \lambda x.\lambda y.E'_1$ va convertita in E'' applicando ricorsivamente questo procedimento ad E'_1.

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

Un'altra proprietà importante del calcolo SK è che ogni λ-espressione che non contiene variabili libere (vale a dire, un combinator) può essere convertita in un'espressione del calcolo SK che non contiene variabili. In altre parole, per modellare solo combinator è possibile rimuovere la prima clausola (un nome di variabile è un'espressione del calcolo SK) dalla definizione del calcolo SK.

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

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

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

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

5 Lambda Calcolo con Tipi

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

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

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

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

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

Prima di tutto, per definire un λ-calcolo con tipi è necessario introdurre il concetto di tipo; questo può essere fatto introducendo un insieme $\mathcal{P}$ di tipi base, o *tipi primitivi*, ed una regola per definire nuovi tipi di dato a partire da tipi di dato esistenti (questo è analogo a quanto fatto per definire le espressioni del λ-calcolo, create a partire da un insieme di nomi di base e 2 regole che permettono di creare nuove espressioni a partire da espressioni valide). Poiché stiamo definendo tipi per un λ-calcolo, un nuovo tipo γ si potà definire a partire da due tipi α e β come $\gamma = \alpha \to \beta$ (γ è il tipo delle funzioni che hanno α come dominio e β come codominio). In altre parole, l'insieme $\mathcal{T}$ dei possibili tipi è generabile tramite la seguente definizione induttiva:

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

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

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

- Il tipo di una variabile libera x deve essere noto a priori
- Se E_1 ed E_2 sono espressioni con tipi $\alpha \to \beta$ e α, il tipo di E_1E_2 è β: $E_1 : \alpha \to \beta, E_2 : \alpha \Rightarrow E_1E_2 : \beta$
- Se E è un'espressione di tipo β, $\lambda x.E$ ha tipo $\alpha \to \beta$: $E : \beta \Rightarrow \lambda x.E : \alpha \to \beta$

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

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

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

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

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

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

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

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

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

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

$$(\lambda x.E)E_1 \to_\beta E[x \to E_1]$$

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

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

.