Lambda calcolo

Dispensa per il corso di Paradigmi di Programmazione
Laurea in Informatica, Università di Pisa

(Versione aggiornata al 20 settembre 2023)

1 Introduzione

In questa dispensa sarà presentato il λ-calcolo come formalismo alla base della programmazione funzionale. Si partirà dalle origini storiche del formalismo, per poi passare alla descrizione degli aspetti sintattici e semantici, fino a trattare il rapporto tra il λ-calcolo e i linguaggi funzionali. La trattazione in termini matematici sarà affiancata da riferimenti ed esempi riferiti a linguaggi di programmazione di uso comune, quali JavaScript e C.

1.1 Prerequisiti

Questa dispensa è stata scritta assumento che il lettore abbia una conoscenza di base dei seguenti argomenti:

- definizione di linguaggi formali tramite grammatiche;
- definizione di relazioni tramite regole di inferenza;
- programmazione in un linguaggio C-like come C, C++, Java, JavaScript o simili.

1.2 Le origini nel teoria della calcolabilità

Per ripercorrere le origini del λ-calcolo è necessario tornare all'inizio del secolo scorso, periodo in cui gli esponenti più autorevoli nell'ambito della matematica e della logica hanno iniziato a interrogarsi sulla possibilità di formalizzare tutte le conoscenze matematiche all'interno di un unico sistema logico, che consentisse, ad esempio, di esprimere tutte le dimostrazioni di tutti i teoremi.

In particolare, nel 1928, David Hilbert formulò il quesito noto come *Entscheidungsproblem (problema della decisione)*, come segue:

> E' possibile definire una procedura, eseguibile del tutto meccanicamente, in grado di stabilire, per qualunque formula esprimibile nel linguaggio formale della logica del primo ordine, se tale formula sia

> o meno un teorema? Ossia se possa essere dedotta a partire da dati assiomi (che includano almeno l'Aritmetica di Peano) usando le regole della logica stessa?

Il problema posto da Hilbert trova una prima risposta negativa nei Teoremi di Incompletezza proposti da Kurt Gödel nel 1930. In sostanza, tali teoremi implicano che in qualunque sistema formale sufficientemente espressivo da includere l'Artimetica di Peano è possibile esprimere delle formule che sono vere, ma non sono dimostrabili a partire dagli assiomi che definiscono il sistema formale e usando solo le regole della logica del primo ordine.

Al di là degli aspetti legati alla completezza dei sistemi formali, il quesito formulato da Hilbert aprì la strada anche allo studio di un'altra importante problematica: come si formalizza la nozione di procedura effettiva, eseguibile meccanicamente? In altri termini, come si esprime un algoritmo in modo formale? Questa domanda ha portato a diverse risposte, tra le quali:

- Alonzo Church: Lambda calculus
  An unsolvable problem of elementary number theory, *Bulletin of the American Mathematical Society*, May 1935

- Alan M. Turing: Turing Machines
  On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the Londong Mathematical Society*, May 1935

Questi due formalismi (assieme al formalismo delle *funzioni ricorsive generali*, proposto da Gödel nel 1931) hanno portato contributi fondamentali nell'ambito della teoria della calcolabilità, e hanno originato famiglie di linguaggi di programmazione che si sono poi evolute nei linguaggi di programmazione odierni.

Il *lambda calculus* (o lambda calcolo, o λ-calcolo) è un formalismo che consente di rappresentare procedure di calcolo in termini di composizioni e applicazioni di funzioni minimali. Il λ-calcolo è l'oggetto di questa dispensa e sarà descritto dettagliatamente nel seguito.

Le *Turing Machines* (o macchine di Turing) sono un formalismo in cui le procedure di calcolo sono rappresentabili come delle macchine idealizzate dotate di un nastro infinito su cui possono leggere e scrivere simboli, e in cui la logica della procedura da eseguire è espressa in termini di un automa a stati finiti: partendo da uno stato iniziale, la "macchina" legge il simbolo corrente nel nastro e, sulla base di questo e delle transizioni disponibili, si sposta in un altro stato ed eventualmente aggiorna il simbolo sul nastro (anche scorrendolo). Questo modo di elaborare simboli leggendo e scrivendo astrae il modo con cui comunemente si calcola utilizzando carta e penna. Partendo da una sequenza di simboli di input (e.g., un'espressione da risolvere) si leggono e scrivono simboli sul foglio seguendo regole ben definite (e.g., le regole dell'aritmetica).

Un'importante caratteristica delle macchine di Turing è che il nastro su cui vengono "memorizzati" i simboli da processare può anche essere usato per memorizzare anche una codifica della funzione che si vuole calcolare: in altre

parole, il codice del programma da eseguire. L'automa che guida l'esecuzione dovrà quindi essere in grado di scandire la porziore di nastro che contiene il programma, per poi andare ad eseguire le operazioni corrispondenti sulla porzione di nastro che contiene i dati.

Turing ha dimostrato che qualunque macchina definita per calcolare una specifica funzione (senza memorizzarne il codice nel nastro, ma con un automa di controllo definito ad hoc per quella funzione) può essere codificata come un programma da memorizzare su nastro. Nasce così il concetto di *Macchina di Turing Universale*, ossia un'unica macchina di Turing capace di "simulare" l'esecuzione di qualunque macchina di Turing opportunamente codificata su nastro, e quindi capace di calcolare qualunque funzione che possa essere calcolata tramite questo formalismo.

I lavori di Turing, così come quelli di Church sul λ-calcolo, sono tra i fondamenti principali della teoria della calcolabilità. Essi hanno consentito di comprendere che non tutte le funzioni che la matematica consente di definire possono essere calcolate in modo meccanico. In altri termini, esistono funzioni per le quali non è possibile definire una macchina di Turing, o un'espressione del λ-calcolo, o un programma in un qualunque linguaggio di programmazione moderno, che ne calcoli il risultato. Questi risultati, assieme ai teoremi di incompletezza di Gödel, hanno portato alla conclusione che il problema della decisione inizialmente proposto da Hilber non fosse risolvibile.

Esempi di *funzioni non calcolabili* sono descritti solitamente sfruttando la capacità di autoreferenzialità dei sistemi formali considerati. Ad esempio, è possibile definire una macchina di Turing che verifichi se una qualunque altra macchina di Turing codificata su nastro sia scritta correttamente (dal punto di vista sintattico), mentre non è possibile definire una macchina di Turing che verifichi se un'altra macchina di Turing codificata su nastro, una volta eseguita, termini sempre la propria esecuzione (questo è i noto come il *problema della fermata*). Analogamente, non è possibile definire un'espressione del λ-calcolo che, date altre due espressioni del λ-calcolo, verifichi la loro equivalenza.

Questi risultati ottenuti con formalismi diversi offrono in realtà punti di vista diversi sullo stesso panorama. Lavori successivi che dimostrano le corrispondenze tra macchine di Turing, espressioni del λ-calcolo, funzioni ricorsive generali, ecc... hanno portato gli scienziati a formulare la seguente ipotesi nota come *Tesi di Church-Turing*:

> Se una funzione è calcolabile (tramite un qualunque formalismo), allora esiste una macchina di Turing in grado di calcolarla.

Questa tesi (indimostrabile, ma universalmente accettata) implica che esiste una classe di funzioni calcolabili che coincide con la classe di funzioni per le quali si può definire una macchina di Turing. Inoltre, dalle corrispondenze dimostrate con gli altri formalismi, segue che anche il λ-calcolo, le funzioni ricorsive generali, ma anche i linguaggi di programmazione moderni possono calcolare esattamente la stessa classe di funzioni. Sono tutti *Turing-completi*. Ne consegue che, qualunque sia il linguaggio di programmazione che usiamo,

non è possibile scrivere un programma che controlli se un altro programma termini sempre, e non è possibile scrivere un programma che controlli se altri due programmi siano equivalenti. Queste sarebbero funzionalità che ci piacerebbe poter avere a disposizione nel nostro ambiente di sviluppo preferito, per controllare che il programma che stiamo scrivendo non si pianti all'infinito per colpa di un bug oppure per verificare che una nuova versione del programma che stiamo scrivendo continui a funzionare come la precedente, ma purtroppo la teoria della calcolabilità ci dice che queste funzionalità non saranno mai disponibili.

1.3 La nascita dei paradigmi imperativo e funzionale

Il modello di calcolo delle macchine di Turing ha ispirato lo scienziato di origini ungheresi John von Neumann nella definizione di un'architettura di elaborazione che è alla base del funzionamento degli attuali computer.

L'*architettura di von Neumann* prevede due componenti principali:

- la **memoria**, dove sono memorizzati sia i programmi da eseguire che i dati da elaborare tramite tali programmi;
- la **unità centrale di elaborazione (CPU)**, che ha il compito di eseguire i programmi immagazzinati in memoria prelevanto le istruzioni (e i dati relativi), interpretandole ed eseguendole una dopo l'altra, aggiornando la memoria.

Le similitudini concettuali tra l'architettura di von Neumann e la macchina di Turing universale sono chiare: la memoria corrisponde al nastro (che contiene anche il codice da eseguire) e la CPU corrisponde all'automa che controlla la macchina. Sebbene i computer moderni aggiungano a questa architettura tutta una serie di funzionalità aggiuntive (ad esempio, l'esecuzione in pipeline, gli aspetti di parallelismo, ecc..) questo è tutt'ora un modello di riferimento valido.

Il fatto che le macchine di Turing imitino in sostanza il modo con cui si calcola usando carta e penna, che l'architettura di von Neumann sia ispirato dalle macchine di Turing, e che i computer moderni di basino sull'architettura di von Neumann, ci porta a concludere che con i calcolatori di oggi quello che possiamo calcolare è in linea di principio ciò che saremmo in grado di calcolare usando carta e penna... Inoltre, la tesi di Church-Turing ci porta a concludere che tanto con carta e penna, quanto con un PC, la classe delle funzioni delle quali possiamo calcolare un risultato è la stessa.

Il fatto che già i primi calcolatori elettronici negli anni '50 si basassero sull'architettura di von Neumann (e quindi sui concetti definiti con le macchine di Turing) ha condizionato la definizione dei primi linguaggi di programmazione. Nascono quindi i linguaggi in stile Turing-von Neumann caratterizzati da:

- **variabili**, come astrazioni delle locazioni di memoria
- **istruzioni di controllo**, come `test-and-jump`, `goto`, `if`, `while`...
- **istruzioni di assegnamento**, per la modifica dello stato della macchina

I linguaggi in questo stlie sono detti *imperativi*, in quanto prevedono in sostanza sequenze di *comandi* da impartire alla macchina. Tra i primi linguaggi di questo tipo possiamo menzionare certamente il Fortran, che è stato il primo vero linaguaggio di programmazione di successo e ancora oggi, essendo un linguaggio poco sofisticato e quindi caratterizzato da una esecuzione efficiente, capita di vederlo utilizzato come linguaggio di implementazione di librerie matematiche o di calcolo ad alte prestazioni. In seguito, grande successo hanno avuto i linguaggi Pascal e C, da cui, per evoluzioni successive, sono nati linguaggi quali C++, Java, C, JavaScript, Python, ecc...

In parallelo, il λ-calcolo ha influenzato la nascita di un'altra classe di linguaggi di programmazione non basati su comandi, ma realizzati principalmente come composizioni di funzioni. In questi linguaggi, detti *funzionali*, il programma è scritto in modo simile a come si definiscono le funzioni matematiche. Esempi notevoli di linguaggi funzionali delle prime generazioni sono LISP ed ML, più recentemente evoluti in linguaggi quali Haskell, OCaml, Scheme, Rust, ecc...

2 Il λ-calcolo

Descriviamo ora il λ-calcolo: formalismo alla base dei linguaggi di programmazione funzionali.

2.1 Il λ-calcolo: Sintassi

La sintassi del λ-calcolo è minimale: consente di scrivere espressioni che contengono solo dichiarazioni di funzioni anonime (tramite un costrutto di astrazione funzionale) e applicazioni di funzioni (tramite un costrutto di giustapposizione funzione-argomento). In aggiunta, il linguaggio consente di introdurre variabili che sono utilizzate solitamente come parametri formali delle funzioni, ma non solo. Questa è la definizione formale.

Sintassi del λ-calcolo. Assumendo un insieme infinito di possibili variabili $V = x, y, z, \ldots$ la sintassi del λ-calcolo consiste di espressioni definite dalla seguente grammatica:

$$\begin{array}{lll} e & ::= x & \text{(variabile)} \\ & \mid \lambda x.e & \text{(astrazione funzionale)} \\ & \mid e\ e & \text{(applicazione)} \end{array}$$

dove x rappresenta in realtà una qualunque variabile in V.

Il costrutto di astrazione funzionale $\lambda x.e$ esprime una funzione anonima con parametro formale x e corpo e. Ad esempio, l'espressione

$$\lambda x.x$$

rappresenta la funzione identità: prevede un parametro x e il suo corpo non è altro che un'espressione che contiene la stessa x.

Come già detto, il costrutto di astrazione funzionale corrisponde ai costrutti per la definizione di funzioni anonime che si trovano in molti linguaggi di programmazione. Ad esempio, in Javascript la funzione identità può essere definita come funzione anonima come segue:

```
function(x) {return x}
```

oppure, più brevemente, come segue:

```
x => x
```

Una funzione anonima del λ-calcolo (detta anche λ-astrazione) può poi essere applicata ad un argomento attraverso il costrutto di applicazione $e\ e$, ossia semplicemente affiancando la definizione della funzione stessa (la prima espressione) alla definizione dell'argomento (la seconda espressione). Ad esempio, possiamo applicare la funzione identità ad una nuova variabile y come segue:

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

Le parentesi in questo esempio sono importanti per far comprendere che la variabile y è al di fuori della definizione della funzione. Omettendo le parentesi (in questo modo: $\lambda x.xy$) si starebbe invece definendo una funzione con un parametro x e con l'applicazione di x a y all'interno del suo corpo. Approfondiremo tra poco l'uso delle parentesi e le convenzioni sintattiche nelle lambda espressioni.

Vediamo alcuni esempi di λ-espressioni sintatticamente corrette:

$$(\lambda x.x)(\lambda y.y) \tag{1}$$
$$((\lambda x.x)(\lambda y.y))z \tag{2}$$
$$((\lambda x.(\lambda y.(xy)))z)k \tag{3}$$
$$((\lambda x.(\lambda y.(xy)))(\lambda z.z))k \tag{4}$$

In questi esempi abbiamo volutamente inserito le parentesi in modo esaustivo. Ricostrundo l'albero degli annidamenti di parentesi si otterrebbe una struttura (detta *albero di sintassi astratta*) che sostanzialmente ricalca l'albero di derivazione dell'espressione che consente di verificare che essa appartiene al linguaggio definito dalla grammatica.

Per evitare di scrivere così tante parentesi, nel λ-calcolo sono previste le seguenti convenzioni sintattiche.

Convenzioni sintattiche. Nelle espressioni del λ-calcolo si assume:

1. che lo *scope* del costruttore λ si estenda il più a destra possibile.
   Ad esempio, $\lambda x.\lambda y.xy$ corrisponde a $\lambda x.(\lambda y.(xy))$, e non a $\lambda x.((\lambda y.x)y)$, in cui lo scope di λy non arriverebbe a includere la y, e neanche $(\lambda x.(\lambda y.x))y$ in cui lo scope di λx non arriverebbe a includere la y.

2. che il costrutto di applicazione sia associativo a sinistra.
   Ad esempio, xyz corrisponde a $(xy)z$, ossia prima si applica x a y, e poi si applica il risultato ottenuto a z. Analogamente, $(\lambda x.x)(\lambda y.y)z$ corrisponde a $((\lambda x.x)(\lambda y.y))z$.

Alla luce di queste convenzioni, i quattro esempi di λ-espressioni illustrati sopra si possono semplificare leggermente come segue:

$$(\lambda x.x)(\lambda y.y) \tag{5}$$
$$(\lambda x.x)(\lambda y.y)z \tag{6}$$
$$(\lambda x.\lambda y.xy)zk \tag{7}$$
$$(\lambda x.\lambda y.xy)(\lambda z.z)k \tag{8}$$

Anche se ancora non abbiamo visto come si "eseguano" (o meglio si *valutino*) le λ-espressioni, analizzando i quattro esempi appena illustrati possiamo descriverli in termini di definizione e applicazione di funzioni. Ad esempio, in (5) stiamo applicando la funzione identità (già vista) ad un'altra definizione della funzione identità (in cui cambia solo il nome del parametro formale). In (6) stiamo applicando la funzione identità alla funzione identità, e il risultato (che sarà a sua volta la funzione identità) viene applicato a z. In (7) stiamo applicando una funzione che prende un primo parametro x e poi un secondo parametro y agli argomenti z e k. Nel suo corpo la funzione applica x a y, ossia applicherà z a k. Infine, l'espressione (8) è analoga a (7), ma il primo argomento che viene passato è la funzione identità (che quindi sarà poi applicata a k, come previsto dal corpo della prima funzione).

In questa prima spiegazione intuitiva abbiamo introdotto una serie di concetti che vedremo in seguito in maggior dettaglio. Ad esempio, il fatto di poter definire funzioni con più di un parametro formale, oppure la possibilità di usare una funzione come argomento da passare ad un'altra funzione.

Prima di poter parlare di questi aspetti, è necessario fare un ragionamento sul ruolo delle variabili nelle λ-espressioni. Negli esempi che abbiamo considerato abbiamo visto che in alcuni casi sostituire una variabile con un'altra non è un problema. Ad esempio, la funzione identità la possiamo definire come $\lambda x.x$, ma anche come $\lambda y.y$. In entrambi i casi stiamo definendo una funzione che ha come corpo il suo parametro, quindi al momento dell'applicazione restituirà esattamente l'argomento che le viene passato. In altre parole, sia $(\lambda x.x)z$ che $(\lambda y.y)z$ dovranno dare come risultato z. Se invece consideriamo le espressioni $(\lambda x.x)z$ e $(\lambda x.x)k$, in cui andiamo ad applicare la stessa funzione a due argomenti che sono variabili diverse, il risultato che otterremo nei due casi sarà diverso: z nel primo e k nel secondo.

Il motivo per cui in $(\lambda x.x)z$ la variabile x può essere sostituita da y, mentre la variabile z non può essere sostituita da k è dovuta al fatto che x è *legata* dal costruttore λx, mentre z è *libera* (non legata da nessun λ). La differenza dovrebbe diventare chiara se pensiamo che il costruttore λx in una astrazione funzionale definisce il parametro formale della funzione. Come in tutti i linguaggi di programmazione più comuni, i nomi che si scelgono per i parametri formali delle proprie funzioni sono ininfluenti e possono essere sempre modificati. Le variabili libere, invece, come z nell'esempio, corrispondono a variabili che non sono usate come parametri formali di nessuna funzione. Tornando al paragone con i linguaggi di programmazione, le variabili libere le possiamo immaginare come variabili globali, oppure come variabili dichiarate in un blocco più esterno

rispetto a quello corrente. Se modificassimo tali variabili, rischieremmo di fare riferimento, ad esempio, ad una variabile globale diversa, cambiando il senso e l'esecuzione del programma.

E' bene quindi essere sempre in grado di identificare le variabili libere all'interno di una λ-espressione. A tal fine, diamo la definizione di *insieme delle variabili libere (free variables) di una λ-espressione e*. Tale insieme è definito in modo costruttivo, per ricorsione sulla struttura dell'espressione.

Insieme delle variabili libere. L'insieme delle variabili libere di una λ-espressione e, denotato $FV(e)$, è definito ricorsivamente sulla struttura di e come segue:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.e) &= FV(e) \setminus \{x\} \\ FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \end{aligned}$$

In sostanza, il calcolo di $FV(e)$ scende ricorsivamente nell'espressione accumulando (tramite l'unione) le variabili che incontra nelle sottespressioni. Nel momento in cui, risalendo, si incontra una λ-astrazione, la variable legata dal λ viene rimossa dall'insieme delle variabili libere calcolato nel corpo della funzione che essa definisce.

Riprendendo alcune espressioni già viste:

- $FV(\lambda x.x) = \emptyset$, poiché $FV(\lambda x.x) = FV(x) \setminus \{x\} = \{x\} \setminus \{x\} = \emptyset$;
- $FV((\lambda x.\lambda y.xy)(\lambda z.z)k) = \{k\}$, poiché

$$\begin{aligned} & FV((\lambda x.\lambda y.xy)(\lambda z.z)k) \\ = \;& FV((\lambda x.\lambda y.xy)(\lambda z.z)) \cup FV(k) && \text{(ricordare associatività)} \\ = \;& (FV(\lambda x.\lambda y.xy) \cup FV(\lambda z.z)) \cup FV(k) \\ = \;& (FV(\lambda x.\lambda y.xy) \cup \emptyset) \cup \{k\} && \text{(saltati passi visti prima)} \\ = \;& (FV(\lambda y.xy) \setminus \{x\}) \cup \{k\} \\ = \;& ((FV(xy) \setminus \{y\}) \setminus \{x\}) \cup \{k\} \\ = \;& (((FV(x) \cup FV(y)) \setminus \{y\}) \setminus \{x\}) \cup \{k\} \\ = \;& ((((\{x\} \cup \{y\}) \setminus \{y\}) \setminus \{x\}) \cup \{k\} \\ = \;& \{k\} \end{aligned}$$

Inoltre, è interessante considerare i seguenti esempi (il cui calcolo è lasciato al lettore):

- $FV(\lambda x.xz) = \{z\}$
- $FV((\lambda x.xk)x) = \{kx\}$

Il primo esempio mostra semplicemente che una variabile libera può occorrere anche nel corpo di una definizione di funzione (che non abbia quella variabile come parametro formale, ovviamente). Il secondo invece mostra che una variabile legata non interferisce con eventuali variabili libere omonime che si trovino

al di fuori dello suo scope. Nell'esempio, la x in $\lambda x.xk$ e la x che si trova alla fine dell'espressione sono due variabili diverse: la prima è legata, mentre la seconda è libera (e viene raccolta in FV). Questo è esattamente ciò che accade anche nella maggior parte dei linguaggi di programmazione: in ambiti diversi del programma si può usare lo stesso nome per due variabili distinte.

Per concludere la trattazione della sintassi del λ-calcolo, introduciamo ora il fatto che espressioni che differiscono solo nel nome scelto per le variabili legate sono da considerarsi equivalenti (le espressioni sono dette in questo caso α-*equivalenti*). Questo ci consentirà, in alcuni casi in cui potrebbere insorgere ambiguità sulle variabili, di rinominare le variabili legate ottenendo un'espressione equivalente a quella che stiamo considerando. Questa operazione di ridenominazione delle variabili legate è detta α-*conversione*.

Nozioni di α-equivalenza e α-conversione. Due espressioni e_1 ed e_2 sono α-*equivalenti* (e si denota $e_1 \equiv_\alpha e_2$) se l'una può essere ottenuta dall'altra rinominando una o più variabili legate, senza interferire con le variabili libere, ossia senza che alcuna occorrenza di una variabile libera diventi legata a seguito della ridenominazione. Data una espressione e_1, l'operazione di ridenominazione delle variabili legate che consente di trasformarla in una espressione e_2 tale che $e_1 \equiv_\alpha e_2$ è detta α-*conversione*.

La nozione di α-equivalenza, qui introdotta basandosi sul concetto intutivo di "ridenominazione delle variabili legate", sarà formalizzato più avanti, quando introdurremo l'operazione più generale di "capture avoiding substitution".

Facciamo qualche esempio di espressioni α-equivalenti ottenibili tramite α-conversioni in cui le istanze legate di x e y sono ridenominate:

- $\lambda x.x \equiv_\alpha \lambda y.y$
- $\lambda x.\lambda y.xyz \equiv_\alpha \lambda y.\lambda x.yxz$
- $(\lambda x.x)x \equiv_\alpha (\lambda y.y)x$

Vediamo inoltre qualche interessante esempio di espressioni che non sono α-equivalenti (e scriveremo $\not\equiv_\alpha$ per indicarlo)

- $\lambda x.\lambda y.xyz \not\equiv_\alpha \lambda y.\lambda x.xyz$
- $\lambda x.xy \not\equiv_\alpha \lambda y.yy$
- $(\lambda x.xy)y \not\equiv_\alpha (\lambda y.yy)y$

Nel primo caso, le due espressioni non sono equivalenti perchè le occorrenze di x e y nel corpo della λ-astrazione non sono state sostituite. Scambiare i nomi di due variabili in due costruttori λ richiede che anche tutte le occorrenze delle variabili ad essi legati siano rinominate. Il secondo esempio è invece più interessante. Quello che accade è che a seguito della ridenominazione della variabile legata x in y, l'occorrenza di y nel corpo della λ astrazione venga "catturata" dal nuovo λy, trasformando y in una variabile legata. Infatti, $FV(\lambda x.xy) = y$, mentre

$FV(\lambda y.yy) = \emptyset$. Il terzo esempio soffre esattamente dello stesso problema, sebbene l'insieme delle variabili libere sia uguale nelle due espressioni (ed è $\{y\}$). Quello che cambia è che comunque la prima occorrenza di y viene catturata, quindi nell'espressione a sinistra entrambe le occorrenze di y sono libere, mentre nell'espressione a destra solo l'ultima occorrenza di y è tale. Le α-conversioni che portano a questo tipo di interfernze tra le variabili libere e quelle legate non sono ammissibili.

2.2 Esecuzione: valutare le λ-espressioni

Dal momento che il λ-calcolo è un linguaggio di espressioni, per descrivere i processi di calcolo che portano a calcolare un risultato, più che di "esecuzione" è opportuno parlare di "valutazione". La *valutazione delle λ-espressioni* è un processo che consiste di una serie di passi di calcolo che possono portare a un risultato finale. Dal momento che il λ-calcolo prevede in sostanza solo funzioni (anonime) e un costrutto per la loro applicazione, i passi di calcolo da svolgere durante la valutazione saranno in sostanza tutti passi di applicazione di funzione.

Se pensiamo alle espressioni aritmetiche, un passo di calcolo corrisponde ad

1. individuare un'operazione all'interno dell'espressione che è pronta per essere calcolata (ad esempio perchè i suoi operandi non contengono ulteriori operazioni);
2. sostituire tale operazione con il suo risultato.

Ad esempio, se consideriamo la seguente espressione artimetica:

$$3 + (5 - 2) + 3 - 1$$

possiamo individuare l'operazione $5 - 2$ come "pronta per essere calcolata" e la possiamo sostituire con il suo risultato ottenendo

$$3 + 3 + 3 - 1$$

La scelta dell'operazione da processare può non essere unica (potevamo scegliere $3 - 1$) e il procedimento va ripetuto finchè non otteniamo un'espressione non ulteriormente processabile (ad es., un singolo numero). Quello sarà il risultato della valutazione dell'espressione.

Nel λ-calcolo il procedimento è analogo, solo che l'unica operazione di calcolo che possiamo eseguire è l'applicazione funzionale. Anche nel caso del λ-calcolo dovremo eseguire gli stessi due passi che abbiamo visto sopra: individuare l'operazione da eseguire e sostituirla con il suo risultato.

L'operazione di applicazione funzionale del λ-calcolo prende il nome di *β-riduzione*, e la porzione di λ-espressione in cui si esegue tale operazione e che viene sostituita dal suo risultato prende il nome di *redex*. Vediamo un esempio con la funzione identità:

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

In questo esempio stiamo applicando la funzione identità alla variabile libera y. Abbiamo quindi una funzione e un argomento: l'intera λ-espressione è un redex pronto per essere sostituito con il risultato dell'applicazione.

L'applicazione funziona come segue: si prende l'argomento e lo sostituisce sintatticamente a tutte le occorrenze del parametro formale che si trovano nel corpo della funzione. Ciò che si ottiene dopo dalla sostituzione sarà il risultato dell'applicazione.

Nell'esempio della funzione identità, il corpo della funzione è x, che è un'occorrenza del parametro formale. Sostituendolo sintatticamente all'argomento (che è y) ottenuamo come risultato y. Quindi $(\lambda x.x)y$ dopo un passo risulta in y, e la valutazione si ferma in quanto y non è un redex (non contiene funzioni pronte per essere applicate). Si dice quindi che il risultato della valutazione della λ-espressione è y, e questo è anche intuitivamente corretto: la funzione identità ha restituito esattamente l'argomento che le era stato passato.

D'ora in avanti descriveremo un passo di β-riduzione tramite una freccia $\rightarrow$, quindi il nostro esempio può essere rappresentato come segue:

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

Vediamo alcuni altri esempi, che prevedono anche più di un passo di calcolo e in cui per chiarezza ad ogni passo sottolineamo il redex che viene processato. Partiamo da questo:

$$\underline{(\lambda x.x)(\lambda y.y)}z \rightarrow \underline{(\lambda y.y)z} \rightarrow z$$

Qui stiamo applicando la funzione identità alla funzione identità, e il risultato dovrà essere applicato alla variabile libera z. Il risultato del primo passo (in cui si sostituisce $\lambda y.y$ a x nel corpo della prima funzione) è la funzione identità applicata a z. Il secondo passo, quindi, risulta in z.

In questo esempio è bene notare che, per via delle convenzioni sintattiche che prevedono l'associatività a sinistra dell'applicazione funzionale, le funzioni vanno applicate esattamente in quell'ordine. Sarebbe sbagliato applicare prima $(\lambda y.y)$ a z (anche se il risultato in questo caso sarebbe lo stesso) e questo diventa chiaro se esplicitiamo le parentesi che sono sottointese per via dell'associatività: $\big((\lambda x.x)(\lambda y.y)\big)z$.

Consideriamo ora questi due esempi:

$$\underline{(\lambda x.\lambda y.xy)(\lambda z.z)}k \rightarrow (\lambda y.(\underline{(\lambda z.z)y}))k \rightarrow \underline{(\lambda y.y)k} \rightarrow k \tag{9}$$

$$\underline{(\lambda x.\lambda y.xy)(\lambda z.z)}k \rightarrow \underline{(\lambda y.((\lambda z.z)y))k} \rightarrow \underline{(\lambda z.z)k} \rightarrow k \tag{10}$$

In entrambi, la λ-espressione da valutare è la stessa, ma durante l'esecuzione vengono fatti passi diversi in quanto (al secondo passo) ci sono due redex possibili tra i quali si può scegliere.

Nel primo caso viene scelto il redex più interno, $(\lambda z.z)y$, mentre nel secondo il redex più esterno, $(\lambda y.((\lambda z.z)y))k$. In entrambi i casi abbiamo una λ-astrazione (nel secondo caso un po' più complicata) applicata ad un argomento. Il risultato che otteniamo alla fine è lo stesso: k.

Visti questi esempi possiamo definire un redex come una struttura $(\lambda x.e_1)e_2$ che deve essere individuata nell'espressione per poter applicare la β-riduzione. Nell'individuare il redex bisogna far attenzione a che non vi siano parentesi aperte o chiuse (anche implicite) tra la λ-astrazione $(\lambda x.e_1)$ e l'argomento e_2. E' possibile che nell'espressione questa struttura ricorra più volte (anche l'una dentro l'altra) e questo apre alla possibilità di scegliere tra diversi modi di valutare l'espressione.

La sostituzione sintattica che viene svolta in ogni β-riduzione può causare problemi. Ad esempio in

$$(\lambda x.\lambda y.xy)y \tag{11}$$

Se andassimo a sostituire l'argomento y al parametro formale x come abbiamo fatto negli esempi precedenti, otterremmo $\lambda y.yy$. Purtroppo, però, questo non va bene, perchè l'occorrenza di y che era l'argomento passato alla funzione era, nell'espressione iniziale, una variabile libera. Dopo la sostituzione, invece, la y inserita nel corpo della funzione viene catturata dal costruttore λy rendendola legata!

Questo modo di eseguire la sostituzione non è quindi corretto, e va modificato in modo da tenere conto delle variabili libere presenti nell'argomento passato alla funzione. Per questo definiamo la nozione di *capture avoiding substitution*.

Capture avoiding substitution. Date due espressioni e_1 ed e_2, e una variabile x, definiamo la *capture avoiding substitution* di x in e_1 con e_2, denotata $e_1\{x := e_2\}$ ricorsivamente sulla struttura di e_1 come segue:

$$\begin{aligned}
x\{x := e_2\} &= e_2 \\
y\{x := e_2\} &= y \qquad \text{se } x \neq y \\
(e'e'')\{x := e_2\} &= (e'\{x := e_2\})(e''\{x := e_2\}) \\
(\lambda y.e)\{x := e_2\} &= \lambda y.(e\{x := e_2\}) && \text{se } x \neq y \text{ e } y \notin FV(e_2) \\
(\lambda y.e)\{x := e_2\} &= \lambda z.((e\{y := z\})\{x := e_2\}) && \text{se } x \neq y, y \in FV(e_2) \\
& && \text{e } z \text{ fresca}
\end{aligned}$$

I primi due casi della definizione in sostanza dicono che se applichiamo la sostituzione $\{x := e_2\}$ ad un espressione costituita solo da una variabile, la sostituzione avrà luogo solo se tale variabile è proprio x. Questo è il caso base della ricorsione sulla struttura della λ-espressione. Se invece (terzo caso) l'espressione a cui applichiamo al sostituzione è un'applicazione $e'e''$, allora la sostituzione dovrà semplicemente propagarsi ricorsivamente sulle due sottoespressioni.

I casi interessanti sono quelli relativi alla λ-astrazione. Qui possiamo trovarci in due situazioni distinte. La prima (più semplice) è quella in cui y, parametro formale della λ-astrazione, non appartiene alle variabili libere di e_2. In questo caso si può procedere con la sostituzione nel corpo della λ-astrazione. Se

invece y, oltre ad essere il parametro formale, fosse anche una variabile libera di e_2, rischieremmo di trovarci nella situazione erronea descritta prima, in cui una variabile libera (di e_2, in questo caso) venga catturata a un costruttore λ, diventando legata. La soluzione adottata in questo caso è di operare una doppia sostituzione. Ossia, prima di applicare la sostituzione $\{x := e_2\}$ nel corpo della λ-astrazione, è necessario rinominare il parametro formale y utilizzando una variabile z che sia *fresca*, cioè che non compaia da nessun'altra parte nella λ-espressione. Va notato che la sostituzione va fatta anche nel costruttore λ, cioè λy diventa λz. In questo modo si ottiene una espressione α-equivalente alla precedente in cui viene meno il confiltto di nomi tra il parametro formale e la variabile libera, e la sostituzione può avvenire.

Prima di fare qualche esempio, è bene notare un'altra cosa: in entrambi gli ultimi due casi della definizione (quelli relativi alle λ-astrazioni) è previsto che la variabile x che si vuole sostituire sia diversa dalla variabile y che è parametro formale della λ-astrazione. Questo significa che se vogliamo sostituire x con e_2 le sole occorrenze di x che andremo a sostituire sono quelle libere. La ricorsione si arresta quando incontra una sottoespressione in cui x è legata.

Vediamo qualche esempio:

- $((\lambda x.yx)w)\{x := \lambda k.kx\} = (\lambda x.yx)w$
- $((\lambda x.yx)w)\{y := \lambda k.kx\} \equiv_\alpha ((\lambda z.yz)w)\{y := \lambda k.kx\} = \lambda z.(\lambda k.kx)z)w$
- $((\lambda x.yx)w)\{w := \lambda k.kx\} = (\lambda x.yx)(\lambda k.kx)$

Nel primo esempio la sostituzione non ha luogo in quanto x non è libera. Nel secondo esempio l'espressione $\lambda k.kx$ contiene la variabile libera x che rischia di essere catturata. Per questo motivo, prima di operare la sostituzione vera e propria è necessario rinominare il parametro formale x usando una variabile fresca z. Nel terzo caso, la sostituzione avviene senza problemi.

L'operazione di capture avoiding substitution ci consente ora di definire la β-riduzione in modo da evitare il problema con le variabili libere descritto in precedenza, e di valutare correttamente anche espressioni come quella presentata in (11).

β-riduzione La regola di β-riduzione descrivere un passo di calcolo di una λ-espressione. Identificato un *redex* $(\lambda x.e_1)e_2$ all'interno dell'espressione, questo viene processato come indicato dalla seguente regola:

$$(\lambda x.e_1)e_2 \to e_1\{x := e_2\}$$

il risultato ottenuto sostituisce il redex, mentre tutto il resto dell'espressione rimane inalterato.

Quindi, riprendendo la λ-espressione in (11), la sua valutazione sarà la seguente:

$$(\lambda x.\lambda y.xy)y \equiv_\alpha \underline{(\lambda x.\lambda z.xz)y} \to \lambda z.yz$$

In questo esempio, il primo passo di α-conversione è esplicitato per mostrare l'intervento della capture avoiding substitution nell'introdurre la variabile fresca

z per evitare di catturare y. Come in precedenza, il redex a cui si applica la riduzione (in questo caso l'intera espressione) è sottolineato per chiarezza.

La valutazione completa di una λ-espressione è quindi una sequenza di passi di applicazione della β-riduzione che porta ad una espressione non ulteriormente riducibile. Tali espressioni sono dette in *forma normale beta*, e rappresentano di fatto i *valori* che si possono ottenere come risultato. Non potendo contenere redex (altrimenti sarebbero ulteriormente riducibili) queste particolari espressioni/valori possono sostanzialmente assumere due forme: una astrazione funzionale, o una espressione che contiene variabili libere che "ostacolano" la riduzione ulteriore. Alcuni esempi di valori di questo tipo:

- $\lambda x.x$
- $\lambda x.xy$
- x
- xy

I primi due casi rappresentano funzioni. Quindi possiamo dire che *nel λ-calcolo le funzioni sono valori*, e questo è forse la principale caratteristica del paradigma di programmazione funzionale, come si riscontra in molti linguaggi moderni, e di cui il λ-calcolo rappresenta il fondamento teorico. Gli ultimi due casi invece rappresentano valori che, per via delle variabili libere, sono intuitivamente non completamente definitivi. Se sapessimo come istanziare tali variabili, potremmo in certi casi continuare con il calcolo. Ma questo è puramente un ragionamento ideologico. All'atto pratico la valutazione dell'espressione si arresta raggiungendo quei risultati.

La β-riduzione fornisce le basi per poter definire una nozione di equivalenza tra λ-espressioni, detta *β-equivalenza*, che tiene conto della valutazione delle espressioni, e non solo della loro sintassi.

β-equivalenza. Due λ-espressioni e_1 ed e_2 si dicono β-equivalenti, indicato $e_1 \equiv_\beta e_2$, se vale una delle seguenti condizioni:

- $e_1 \equiv_\alpha e_2$
- $e_1 \Rightarrow e_1' \equiv_\alpha e_2$ oppure $e_2 \Rightarrow e_2' \equiv_\alpha e_1$
- $e_1 \Rightarrow e_1'$ e $e_2 \Rightarrow e_2'$ con $e_1' \equiv_\alpha e_2'$

dove $\Rightarrow$ rappresenta una sequenza di β-riduzioni consecutive (formalmente, $\Rightarrow$ è la chiusura transitiva della relazione $\rightarrow$).

Due espressioni sono β-equivalenti se (e solo se), in sostanza, porteranno allo stesso risultato. Questo accade se le due espressioni sono uguali a meno di rinominare le variabili legate (cioè sono α-equivalenti), se una si riduce nell'altra (di nuovo, a meno di α-equivalenza), o se entrambe si riducono in una stessa

espressione (sempre modulo α-equivalenza). Quindi, ad esempio, valgono la seguenti equivalenze:

$$(\lambda x.x) \equiv_\beta (\lambda y.y)$$
$$(\lambda x.\lambda y.xy)(\lambda x.x)z \equiv_\beta (\lambda k.k)z$$
$$(\lambda x.\lambda y.xy)(\lambda x.x)(\lambda x.x)z \equiv_\beta (\lambda x.x)((\lambda x.\lambda y.x)zk)$$

Si lascia al lettore verificare quale caso della β-equivalenza si applichi, e che la valutazione delle λ-espressioni equivalenti porti allo stesso risultato.

3 Confluenza e non terminazione

Abbiamo visto che la β-riduzione consente di valutare le λ-espressioni ottenendo un risultato. Abbiamo anche visto, però, che durante la valutazione ci si può trovare in un situazione in cui ci siano più redex pronti per essere processati, con la conseguenza di dover operare una scelta tra essi. Questo accadeva negli esempi in (9) e (10) visti in precedenza. In quegli esempi, entrambe le scelte operate al secondo passo conducevano comunque allo stesso risultato finale k.

Il fatto che comunque si scelga di processare una λ-espressione si ottenga sempre lo stesso risultato non è scontato. Esistono formalismi (o linguaggi) in cui questa proprietà non vale.

Pensiamo ad esempio alla valutazione delle espressioni Booleane in molti linguaggi di programmazione. Ad esempio, in linguaggi C-like (come C, C++, Java, JavaScript, ecc..), si può scrivere un `if` simile al seguente:

```
if (den!=0 && num/den>0) {
    ...
}
```

Se gli operandi del `&&` si potessero valutare in ordine arbitrario (come accade in logica con l'operazione di congiunzione), a seconda che si scelga di valutare prima `den!=0` o `num/den>0` si potrebbero ottenere due risultati diversi quando `den` è zero: rispettivamente `false` e un errore di esecuzione.

Altri esempi possono essere trovati facilmente nella programmazione concorrente: quando si eseguono due thread concorrenti che lavorano sulla stessa area di memoria, può accadare che a seconda di quale thread viene messo in esecuzione per primo, i valori finali che si troveranno nella memoria condivisa siano diversi.

Nel λ-calcolo fortunatamente queste problematiche non si verificano. Vale una proprietà di *confluenza* delle computazioni che è sancito dal Teorema di Church-Rosser.

Teorema di Church-Rosser. Data una λ-espressione e, se esistono due sequenze di β-riduzioni diverse che consentono di ridurre e in e_1 o in e_2 non α-equivalenti, allora esiste anche una λ-espressione e' tale che entrambe e_1 ed e_2 potranno essere ridotte in e' (o a qualcosa di α-equivalente).

Il teorema quindi implica che qualunque scelta si operi nell'applicare la β-riduzione, le espressioni che si ottengono potranno poi sempre convergere verso lo stesso risultato. Questa proprietà prende il nome di *confluenza* proprio richiamando il fatto che i diversi flussi di esecuzione della valutazione vanno a riunirsi (a confluire) nello stesso risultato finale.

E' importante sottolineare che il teorema dice che e_1 ed e_2 *possono* essere entrambi ridotti ad e', ossia, esistono una sequenze di β-riduzioni che trasformano entrambe in e'. La domanda che ci si può porre ora è: *la valutazione di una λ-espressione potrebbe non terminare?*

La risposta alla domanda è affermativa: è possibile definire λ-espressioni non terminanti. Un esempio classico è la λ-espressione nota come *combinatore* Ω, definito come segue:

$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

E' facile accorgersi che applicando la β-riduzione a Ω si ottiene nuovamente Ω, e questo porta ad una computazione infinita:

$$\Omega = (\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx) \rightarrow \dots$$

Inoltre, può succedere che a seconda delle scelte che si fanno quando ci sono più redex disponibili, la valutazione della stessa λ-espressione possa terminare o meno, come in questi casi, in cui il redex scelto ad ogni passo è sottolineato:

- $\underline{(\lambda x.k)\Omega} \rightarrow k$
- $(\lambda x.k)\underline{\Omega} \rightarrow \underline{(\lambda x.k)\Omega} \rightarrow k$
- $(\lambda x.k)\underline{\Omega} \rightarrow (\lambda x.k)\underline{\Omega} \rightarrow \dots (\lambda x.k)\underline{\Omega} \rightarrow \dots$

Quello che ci dice il Teorema di Church-Rosser è che, anche in questi casi, tutte le computazioni *che terminano* portano allo stesso risultato di valutazione.

4 λ-calcolo e programmazione funzionale

Vediamo ora perchè un formalismo così minimale come il λ-calcolo è considerato il fondamento teorico della programmazione funzionale.

4.1 Funzioni Curryed e higher-order

Come già detto in precedenza, uno degli elementi chiave della programmazione funzionale è il fatto che *le funzioni sono valori*, e come tali possono essere il risultato di una computazione, ma possono anche essere passate come argomenti ad altre funzioni. Quest'ultimo aspetto consente di definire *funzioni di ordine superiore (o higher-order)*, ossia funzioni che prendono funzioni e le utilizzano nel loro corpo. Un esempio banale è la funzione *Apply*, definita come segue:

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

La funzione *Apply* prende una funzione f, un argomento x e poi applica f ad x. Quindi, assumendo di dare il nome Id alla funzione identità, definita come $Id = (\lambda x.x)$, avremo:

$$Apply\ Id\ k \rightarrow (\lambda x.Id\ x)k \rightarrow Id\ k \rightarrow k$$

Si noti che *Apply* è apparentemente una funzione con due parametri, mentre il λ-calcolo prevede astrazioni funzionali con un solo parametro. Il modo con cui si può definire una funzione che lavora su più parametri è annidando le astrazioni funzionali (cioè scrivendo $\lambda x.\lambda y.\ldots$).

Le funzioni con più parametri definite in nel modo del λ-calcolo vengono dette *funzioni Curryed* (dal matematico Haskell Curry). Esse sono un altro aspetto caratteristico della programmazione funzionale e hanno la particolarità di poter essere applicate parzialmente. Ad esempio, proviamo a passare solo il primo argomento alla funzione *Apply*:

$$Apply\ Id$$

quello che otteniamo con un passo di β-riduzione tenendo conto di come è definita la funzione *Apply*, è:

$$\lambda x.Id\ x$$

ossia una funzione con un parametro formale x a cui sarà applicata la funzione Id. Abbiamo applicato la funzione *Apply* parzialmente (passandole solo il primo argomento) e questo ha portato ad ottenere come risultato una funzione pronta a ricevere il secondo argomento e completare il lavoro.

Per fare un esempio più concreto, assumiamo per un attimo di disporre anche di una operazione di somma + e di poter usare valori interi nelle λ-espressioni. La funzione che fa la somma di due numeri potrebbe quindi essere definita come segue:

$$Somma = (\lambda x.\lambda y.x + y)$$

tale che

$$Somma\ 10\ 5 \rightarrow \lambda y.10 + y \rightarrow 10 + 5 = 15$$

Applicando parzialmente la funzione si otterrebbe:

$$Somma\ 10 \rightarrow \lambda y.10 + y$$

che è una funzione pronta a sommare 10 ad un valore che le venga passato.

A questo punto, continuando a definire funzioni, potremmo immaginarci di dare un nome a tale funzione ottenuta come risultato:

$$SommaDieci = Somma\ 10$$

potendo a questo punto scrivere:

$SommaDieci\ 5 = 15$ $\quad$ $SommaDieci\ 3 = 13$ $\quad$ $SommaDieci\ 10 = 20$ $\quad$ *ecc...*

4.2 Ricorsione: il combinatore Y

Nel λ-calcolo non è previsto un meccansimo nativo (ad es., un costrutto sintattico specifico) per definire funzioni ricorsive. Le funzioni sono anonime, quindi non è possibile definire funzioni ricorsive richiamando la funzione stessa nel suo corpo[1] e non esistono altri operatori che consentano di iterare o eseguire ricorsivamente una data funzione. Ciò nonostante, un meccanismo di ricorsione può essere *implementato* usanto i pochi costrutti sintattici che il formalismo mette a disposizione.

Una funzione può essere resa ricorsiva passandola ad un'altra particolare funzione definita negli anni '40 dal matematico e logico Haskell Curry, e che prende il nome di Y *combinator* (o combinatore Y), definito come segue:

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

Data una qualunque λ-espressione F (che in pratica dovrà definire una funzione affinché la cosa abbia senso) il combinatore Y soddisfa la seguente proprietà di *punto fisso*:

$$YF \equiv_\beta F(YF)$$

Questa proprietà può essere verificata mostrando che sia YF che $F(YF)$ si riducono dopo pochi passi di β-riduzione alla stessa λ-espressione, come segue:

$$\begin{aligned} YF &= \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))F \\ &\to (\lambda x.F(xx))(\lambda x.F(xx)) \\ &\to F((\lambda x.F(xx))(\lambda x.F(xx))) \end{aligned}$$

$$\begin{aligned} F(YF) &= F(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))F) \\ &\to F((\lambda x.F(xx))(\lambda x.F(xx)) \end{aligned}$$

da cui, per il terzo caso della definizione di β-equivalenza, si può concludere che $YF \equiv_\beta F(YF)$.

Ora, come si utilizza il combinatore Y per definire funzioni ricorsive? Il modo è il seguente:

- si definisce una λ-espressione e al cui interno si usa un nome f per effettuare le chiamate ricorsive. In sostanza, e rappresenta il corpo della funzione ricorsiva f che si intende definire;
- si definisce la λ-astrazione $G = \lambda f.e$
- si definisce $F = YG$

[1] Anche negli esempi precedenti in cui abbiamo dato nomi di convenienza alle funzioni *Apply*, *Id*, *Somma*, ecc.. non stavamo effettivamente dichiarando funzioni. Non è quindi possibile, nel λ-calcolo, scrivere qualcosa tipo $R = \lambda x.R$ in quanto $=$ non è nella sintassi del linguaggio.

. Così facendo, F sarà la funzione ricorsiva che si voleva definire.

Ad esempio, consideriamo l'espressione

$$e = \lambda y.(fy)y$$

che rappresenta una funzione che applica "ricorsivamente" f al suo parametro y, aggiungendo una istanza di y in fondo. Definiaimo la λ-astrazione:

$$G = \lambda f.e$$

e applichiamo il combinatore Y:

$$F = YG$$

Applicando F ad un argomento k, la sua esecuzione sarà la seguente:

$$\begin{aligned} & YGk \\ \Rightarrow\ & G(\lambda x.G(xx))(\lambda x.G(xx))k \\ \rightarrow\equiv_\beta\ & (\lambda y.((YG)y)y)k \\ \rightarrow\ & (YG)kk \\ \Rightarrow\ & (YG)kkk \\ \rightarrow\ & \ldots \end{aligned}$$

dove $\Rightarrow$, come in precendenza, rappresenta la chiusura transitiva di $\rightarrow$, quindi una sequenza di passi di β-riduzioni non illustrati in dettaglio per semplicità. Inoltre, la β-equivalenza $\equiv_\beta$ nello sviluppo è usata per poter indicare più semplicemente (YG) nell'espressione anzichè l'espressione più complessa (ma equivalente) che in realtà si otterrebbe.

Un esempio più concreto lo si può osservare in un linguaggio di programmazione che preveda funzioni anonime in stile λ-calcolo. Vediamo come si può implementare il combinatore Y in JavaScript e utilizzarlo per implementare una funzione per il calcolo del fattoriale usando solo funzioni anonime[2]:

```
const Y = f => (x => x(x))(x => f(y => x(x)(y)))
const factorial = f => (x => (x===1 ? 1 : x * f(x-1)))
const ris = Y(factorial)(10)
```

4.3 Rappresentare dati e controllo del flusso di esecuzione come funzioni

Brevemente, illustriamo come altri costrutti comuni nei linguaggi di programmazione funzionali possano essere codificati in λ-calcolo. Questo ci consentirà di considerare il λ-calcolo come un buon modello di questa famiglia di linguaggi di programmazione.

[2]Il combinatore Y implementato in questo esempio è leggermente diverso da quello definito sopra come λ-espressione per evitare che l'interprete JavaScript vada in stack overflow.

Letterali Booleani. Le costanti $True$ e $False$ possono essere descritti come le seguenti λ-astrazioni:

$$True = \lambda t.\lambda f.t \qquad\qquad False = \lambda t.\lambda f.f$$

ossia come funzioni con due parametri formali che restituiscono rispettivamente il primo o il secondo parametro.

Letterali numerici (naturali). Il numero naturale n è codificato come la λ-espressione C_n definita come segue:

$$C_n = \lambda z.\lambda s.s(s(\ldots(s\ z))\ldots)$$

dove s è ripetuto n volte. L'idea è che il parametro z rappresenti lo zero e il parametro s la funzione successore. Quindi $C_0 = \lambda z.\lambda s.z$, $C_1 = \lambda z.\lambda s.s(z)$, $C_2 = \lambda z.\lambda s.s(s(z))$, ecc...

Espressioni condizionali. Tenendo conto della codifica dei Booleani data prima, un costrutto di *espressione* condizionale[3] può essere definito come segue:

$$IF \quad = \quad \lambda c.\ \lambda then.\ \lambda else.\ c\ then\ else$$

quindi, ad esempio, l'espressione $IF\ True\ e_1\ e_2$ avrà il seguente sviluppo:

$$\begin{aligned} IF\ True\ e_1\ e_2 =& (\lambda c.\ \lambda then.\ \lambda else.\ c\ then\ else)\ True\ e_1\ e_2 \\ \rightarrow\ & (\lambda then.\ \lambda else.\ True\ then\ else)\ e_1\ e_2 \\ \rightarrow\ & \lambda else.\ True\ e_1\ else)\ e_2 \\ \rightarrow\ & True\ e_1\ e_2 \\ \rightarrow\ & (\lambda t.\lambda f.t)\ e_1\ e_2 \\ \rightarrow\ & (\lambda f.e_1)\ e_2 \\ \rightarrow\ & e_1 \end{aligned}$$

Operazioni su naturali. Tenendo conto della rappresentazione dei numeri naturali vista sopra, è possibile definire le operazioni aritmetiche come λ-astrazioni che prendono come parametri gli operandi n ed m. Queste le definizioni, ad esempio, delle operazioni di somma e moltiplicazione:

$$\begin{aligned} Plus &= \lambda m.\lambda n.\lambda z.\lambda s.(m\ (n\ z\ s)\ s) \\ Times &= \lambda m.\lambda n.(m\, C_0\, (Plus\ n)) \end{aligned}$$

Anche se non è semplice, lasciamo al lettore di verificare l'esecuzione di operazioni quali:

$$Plus\ C_3\ C_2 \qquad \text{e} \qquad Times\ C_3\ C_2$$

Oltre a quanto visto in questa sezione, è possibile codificare in λ-calcolo anche costrutti per la dichiarazione di variabili locali, strutture dati (coppie, liste, ecc...), ecc. Il concetto importante da comprendere è che nella sua semplicità il λ-calcolo è un valido modello dei linguaggi di programmazione funzionale in quanto consente di codificarne tutte i principali costrutti linguistici.

[3] Da non confontere con il *comando* di scelta condizionale `if` tipico dei linguaggi imperativi

5 Strategie per il passaggio dei parametri: call-by-name (lazy) e call-by-value (eager)

Nelle sezioni precedenti abbiamo visto che ci possono essere più modi per valutare una λ-espressione, e che il Teorema di Church-Rosser garantisce che tutte le riduzioni che terminano porteranno allo stesso valore. Ma quando ci si trova nella situazione di dover scegliere tra due o più redex pronti per essere ridotti, su quali criteri è conveniente basare la propria scelta? In altre parole, è possibile definire una *strategia di valutazione* che risolva le scelte e renda il processo di valutazione deterministico?

Chiaramente è possibile definire diverse strategie di valutazione, ma in particolare è interessante considerarne due che trovano riscontri nei linguaggi di programmazione attuali: la strategia *call-by-name* (che porta ad una valutazione *lazy*) e la strategia *call-by-value* (che porta ad una valutazione *eager*). Entrambe sono strategie che riguardano nello specifico il passaggio dei parametri.

Consideriamo questo esempio, in cui bisogna notare l'uso delle parentesi:

$$(\lambda x.xx)\big((\lambda y.yyy)k\big)$$

i redex tra cui scegliere per applicare la β-riduzione sono due:

$$\underline{(\lambda x.xx)\big((\lambda y.yyy)k\big)} \qquad \text{e} \qquad (\lambda x.xx)\big(\underline{(\lambda y.yyy)k}\big)$$

E' facile notare che entrambe le scelte portano in due passi allo stesso risultato: $(kkk)(kkk)$.

In questo esempio ci troviamo in pratica di fronte ad una funzione pronta per essere applicata ad un argomento che potrebbe a sua volta essere immediatamente processato. Dobbiamo quindi scegliere se applicare prima la funzione più esterna (passandole l'argomento così com'è definito, posticipando la valutazione di quest'ultimo), oppure se prima valutare l'argomento e poi applicare la funzione più esterna al risultato ottenuto.

La prima scelta corrisponde alla strategia *call-by-name*, in cui l'argomento viene passato così com'è scritto, che corrisponde alla valutazione *lazy* (pigra, poichè si posticipa la valutazione dell'argomento). La seconda scelta corrisponde invece alla strategia *call-by-value*, in cui la funzione viene applicata al valore ottenuto dalla valutazione dell'argomento, che corrisponde corrisponde alla valutazione *eager* (avida, impaziente di valutare l'argomento).

La scelta di quando valutare gli argomenti passati a una funzione è critica anche nei linguaggi di programmazione. La maggior parte dei linguaggi utilizza un approccio eager. Ad esempio, usando la sintassi di C (e derivati), la chiamata di funzione

```
f(3+4);
```

porta prima a calcolare la somma, e poi a passare il risultato `7` alla funzione.

Questa scelta porta a delle conseguenze. Ad esempio, se la chiamata fosse la seguente

```
f(g(0)+1);
```

avremmo che viene prima eseguita la funzione `g` e poi la funzione `f`. Se entrambe le funzioni, ad esempio, lavorassero su una stessa variabile globale, potremmo avere un'interferenza di `g` sul calcolo che dovrà effettuare `f` di cui dobbiamo essere consapevoli.

Esistono linguaggi che prevedono la valutazione lazy (l'esempio classico è il linguaggio Haskell), oppure che prevedono operatori per specificare che una certa espressione debba essere valutata in modo lazy (come OCaml). La valutazione lazy, se ben implementata, offre vantaggi prestazionali (si valuta l'argomento solo se necessario) e di espressività (ad es., per lavorare con strutture dati infinite). Tuttavia, la valutazione eager è nella stragrande maggioranza dei casi preferita in quanto molto più semplice per il programmatore.

Studiamo però le differenze tra le due strategie nel λ-calcolo. Per fare questo è opportuno formalizzare la β-riduzione tramite regole di inferenza. Questo ci consentirà di esprimere le strategie.

Per cominciare, la relazione di β-riduzione standard (senza strategie) può essere formalizzata come segue:

$$(\lambda x.e_1)e_2 \to e_1\{x := e_2\}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{e_1\ e_2 \to e_1\ e_2'} \qquad \frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

L'assioma di questa definizione con regole di inferenza è quello già visto come definizione della β-riduzione: nel corpo dell'astrazione funzionale $(\lambda x.e_1)$ si sostituisce ogni occorrenza libera di x con l'argomento e_2. Le tre regole di inferenza, invece, esprimono formalmente il fatto che la riscrittura descritta dall'assioma si possa applicare ad una qualunque sotto-espressione della λ-espressione che stiamo processando, cosa che in precedenza era stata specificata nel testo della definizione di β-riduzione, ma che ora è formalizzata in termini matematici.

Le tre regole di inferenza, quindi, consentono di ridurre sotto-espressioni senza vincolare la scelta su quale di esse processare.

Riconsiderando l'esempio dell'espressione $(\lambda x.xx)\big((\lambda y.yyy)k\big)$ abbiamo infatti che la definizione ci consente di generare entrambi i seguenti alberi di derivazione, per il primo passo di β-riduzione:

$$\frac{}{(\lambda x.xx)\big((\lambda y.yyy)k\big) \to \big((\lambda y.yyy)k\big)\big((\lambda y.yyy)k\big)} \tag{12}$$

e

$$\frac{(\lambda y.yyy)k \to kkk}{(\lambda x.xx)\big((\lambda y.yyy)k\big) \to (\lambda x.xx)(kkk)} \tag{13}$$

Consideriamo ora la valutazione *lazy* (strategia *call-by-name*), che può essere definita come segue:

$$(\lambda x.e_1)e_2 \to e_1\{x := e_2\}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}$$

In questo caso abbiamo che la precedenza nella scelta tra i diversi redex abilitati debba essere data all'applicazione della funzione a livello sintatticamente più alto (ossia, più esterna). Questa strategia la otteniamo rimuovendo dalla definizione le regole di inferenza che consentono di ridurre l'argomento di una applicazione funzionale e il corpo di una funzione. Quindi, la strategia impone di portare, usando l'unica regola di inferenza disponibile, l'espressione in una forma in cui $(\lambda x.e_1)e_2$ è a top-level, e poi di applicare la sostituzione di x con e_2 senza poter prima ridurre e_1 o e_2. In questo modo, tornando all'esempio, l'albero di derivazione in (13) non è più valido, mentre rimane possibile effettuare la derivazione in (12).

Infine, consideriamo la valutazione *eager* (strategia *call-by-value*), che definiamo come segue:

$$(\lambda x.e_1)v \to e_1\{x := v\}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

Qui abbiamo che la precedenza debba essere data alla riduzione dell'argomento della funzione, prima che all'applicazione. Per questo modifichiamo l'assioma della definizione utilizzando il simbolo v per esprimere una λ-espressione non ulteriormente riducibile (ossia un valore). In questo modo, in un'applicazione funzionale, la funzione può essere applicata solo dopo aver ridotto completamente l'argomento. A questo punto, rispetto alla strategia call-by-name, reintroduciamo la regola di inferenza che consente di ridurre l'argomento di una applicazione funzionale, ma con un accorgimento: anziché consentire di ridurre liberamente l'argomento e_2, consentiamo a questa regola di applicarsi solo dopo che la funzione da applicare è stata completamente ridotta, ossia quando abbiamo ve_2. In questo modo la strategia richiederà di ridurre la funzione da applicare (per portarla in forma $\lambda x.e_1$, ma senza scendere nel corpo e_1) per poi procedere con la riduzione dell'argomento e_2 prima di poter applicare.

Tornando ancora all'esempio, abbiamo che con questa strategia l'albero di derivazione in (12) non sarà più valido, mentre rimane possibile effettuare la derivazione in (13).

Attenzione! Nelle due strategie appena descritte, gioca un ruolo importante l'assenza della seguente regola di inferenza, che consente di eseguire parte del corpo di una funzione prima di applicarla ad un argomento:

$$\frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

senza questa regola abbiamo che, ad esempio, la seguente espressione non possa essere ridotta:

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

quando invece usando la β-riduzione standard (senza strategie), tale espressione farebbe un passo di riduzione in $\lambda x.z$.

Questo che significa che le strategie di call-by-name e call-by-value **non sono equivalenti** alla β-riduzione standard in quanto possono portare a risulati diversi ($\lambda x.((\lambda y.y)z)$ invece che $\lambda x.z$, nell'esempio). D'altronde, il modo di procedere delle strategie call-by-name e call-by-value è quello adottato anche in tutti i linguaggi di programmazione: il corpo di una funzione non viene mai eseguito (nemmeno parzialmente) prima dell'invocazione della funzione stessa. Ad esempio, se in C abbiamo la funzione:

```
int foo(int x) {
  int y = 5;
  int z = 6+y;
  return x+z;
}
```

sebbene prima ancora di chiamare la funzione si potrebbe calcolare il valore di `z`, questo non viene fatto. Il corpo viene (completamente) eseguito solo nel momento in cui da qualche parte viene effettuata, ad esempio, la chiamata `foo(5)`.

E' bene notare, a questo punto, che entrambe le strategie call-by-name e call-by-value sono *determinstiche*. Per qualunque λ-espressione abbiamo sempre una sola possibile regola applicabile. Questo fa si che queste regole possano essere la base su cui definire un interprete (cioè un esecutore) di un linguaggio di programmazione funzionale che, chiaramente, deve essere in grado di eseguire i programmi in modo ben determinato, non ambiguo.

L'ultima questione da chiarire ora è: le strategie call-by-name e call-by-value sono del tutto equivalenti? Il Teorema di Church-Rosser risponde in parte a questa domanda: certamente le due strategie non potranno portare la stessa espressione ad essere valutata con due risultati completamente diversi (nonostante il discorso fatto sopra sulla non equivalenza di queste strategie con la β-riduzione standar). Consideriamo però il seguente esempio:

$$IF\ True\ True\ (\Omega\ \Omega)$$

dove IF, $True$ e Ω sono definiti come nelle sezioni precedenti. In questo caso è facile osservare che le due strategie portano a comportamenti diversi. Nel caso della call-by-name i tre argomenti dell'IF vengono passati così come sono, quindi IF riconosce immediatamente che la guardia è $True$ e resituisce il valore nel primo ramo: $True$. Nel caso della call-by-value, invece, i tre argomenti devono essere ridotti prima di essere passati all'IF. Tra questi, però, c'è $\Omega\ \Omega$ la cui valutazione non termina, portando l'intera espressione a non poter essere valutata. Quindi, questo esempio illustra che le due strategie possono differire dal punto di vista della terminazione e, in particolare, la strategia call-by-value

può in certi casi non terminare quando quella call-by-name terminerebbe. Ciò nonostante, come già detto, la maggior parte dei linguaggi di programmazione segue un approccio call-by-value (eager) per la sua maggiore semplicità di comprensione per il programmatore. Sarà compito del programmatore fare attenzione a non passare alle funzioni argomenti la cui valutazione potrebbe non terminare.