Comprendere lo sviluppo dei moderni linguaggi di programmazione in chiave evolutiva

Silvia Crafa

Sommario

La teoria dell'evoluzione è da tempo utilizzata nell'analisi di sistemi non biologici, come lo sviluppo della tecnologia o dei linguaggi umani. Anche la storia dei linguaggi di programmazione, caratterizzata da veri e propri processi di variazione e selezione, può essere riletta in chiave evolutiva. Senza forzare un'interpretazione darwiniana in un contesto così diverso, in questo articolo useremo con precisione il linguaggio evolutivo biologico e i suoi pattern esplicativi per mettere in luce le diverse dinamiche che entrano in gioco nel processo di sviluppo dei linguaggi di programmazione.

Abstract

The Evolutionary Theory has been used to analyze non biological systems, such as the evolution of technology and that of human languages. Even the history of programming languages can be observed in an evolutionary framework, since it displays variation and selection processes. In this paper, rather than casting programming languages evolution into the Darwinian account, we borrow the biological talking to unveil evolutionary patterns and driving forces that unfold behind the development of this rich scientific area.

Keywords: evolutionary theory, evolution of technology, history of programming languages, programming abstractions, mainstream programming models

1. Introduzione

Anche se non sempre ce ne rendiamo conto, i linguaggi di programmazione hanno una notevole influenza sull'ambiente che ci circonda, perché stanno alla base del funzionamento di quasi tutti i dispositivi tecnologici, dai piccoli sistemi embedded inseriti nella lavatrice di casa o nella centralina di controllo dell'automobile, ai sistemi software che gestiscono una banca o un ospedale, fino agli smart-tags oggi inseriti in molti oggetti che danno vita al cosiddetto Internet of Things.

Come i linguaggi umani, anche i linguaggi di programmazione (LP) servono per comunicare: permettono ad un programmatore di istruire un dispositivo hardware, permettono a due o più programmi software di interoperare tra loro, e permettono anche ai programmatori di scambiarsi algoritmi, cioè informazioni in formato preciso e non ambiguo.

I primi linguaggi di programmazione nacquero negli anni '50, ma la loro esplosione avvenne negli anni '80 sulla spinta dell'ampia diffusione dei personal computer. Al giorno d'oggi si tratta ancora di un ambito estremamente vivo e dinamico: nuovi linguaggi nascono continuamente e i linguaggi più diffusi competono tra loro, spesso spinti dalle aziende che li usano all'interno delle tecnologie che vendono. Anche la misurazione della popolarità di un linguaggio è un argomento molto dibattuto e non facile da definire [1].

L'analisi dello sviluppo storico di questi linguaggi può essere fatta secondo diverse prospettive; volendo ad esempio compilare una linea temporale di LP bisogna scegliere un criterio secondo cui ordinare i linguaggi, ma la scelta di tale criterio non è così semplice come appare a prima vista: ad esempio elencare i linguaggi rispetto alla data di invenzione oppure alla data in cui sono diventati popolari produrrebbe risultati molto diversi. Non è raro infatti che il successo di un linguaggio non sia legato all'obiettivo per cui era stato inventato: ad esempio Objective-C è rimasto in sordina quasi vent'anni prima che Apple lo rendesse estremamente diffuso in quanto unico linguaggio utilizzabile per programmare le apps dei dispositivi mobili Apple. Da poco più di un anno inoltre Apple ha optato per un nuovo linguaggio di programmazione, Swift, più moderno e più orientato alle nuove tecnologie, determinando quindi un rapido crollo della diffusione di Objective-C, che si è forse avviato verso l'estinzione (si veda ad esempio l'indice TIOBE di Objective-C in [2]). Molto diverso sembra invece essere il destino del linguaggio Cobol: nato nel 1961, pur essendo passato per una serie di aggiornamenti è senza dubbio un linguaggio obsoleto, ma sembra essere quasi immortale perché gran parte dei sistemi software finanziari sono scritti in questo linguaggio e tradurre del codice stabile in un nuovo linguaggio aprirebbe la porta all'introduzione di errori e problemi di manutenibilità e retro compatibilità che non sarebbero sostenibili. Succede quindi che il nocciolo di questi software rimane scritto in Cobol e viene rivestito di uno strato esterno e di un front-end scritti in linguaggi più moderni.

Le alterne vicende di Objective-C e di Cobol sono un piccolo esempio di come la storia dei LP sia caratterizzata da innovazioni, diversificazioni, trasferimenti orizzontali e influenze sociali, che possono essere letti come veri e propri processi

di variazione e selezione distintivi della teoria dell'evoluzione darwiniana. Lo studio di questa teoria è infatti già da tempo uscito dai confini della biologia per essere applicato all'analisi di svariati sistemi culturali, tra cui l'evoluzione della tecnologia (e.g. [3][4][5]) e l'evoluzione del linguaggio umano (e.g.[6][7]). Per comprendere fino a che punto i sistemi non biologici possono essere visti come un sistema evolutivo, va innanzitutto ricordato che la teoria dell'evoluzione comprende molto più che i soli meccanismi di variazione e selezione, raccogliendo al suo interno una ricca varietà di pattern esplicativi. Alcuni, come le mutazioni casuali, la contingenza e la selezione naturale, fanno parte del nocciolo della teoria, altri invece sono meno fondanti ma comunque ben riconosciuti, come ad esempio la plasticità fenotipica, il gradualismo o gli equilibri punteggiati e l'effetto della costruzione della nicchia (si veda [8] per uno scorcio del dibattito attuale su quali siano i processi fondanti della teoria dell'evoluzione). D'altra parte in letteratura è molto vivo anche il dibattito su quale sia il senso e il limite dell'analogia tra l'evoluzione biologica e quella culturale (e.g., [9][10]): spesso variazione, selezione e soprattutto ereditarietà operano in modo molto diverso nei sistemi biologici ed in quelli culturali, e lo sviluppo di tratti culturali molto diversi tra loro, come ad esempio la diffusione dei brevetti e quella delle lingue parlate, richiedono chiaramente spiegazioni molto diverse e ad hoc.

Ponendoci nel solco di questi dibattiti, in questo articolo ci chiederemo se è possibile guardare alla storia dei linguaggi di programmazione in una prospettiva evolutiva. Senza voler forzare l'analogia con la teoria darwiniana, useremo il linguaggio evolutivo come strumento esplicativo, allo scopo di analizzare le diverse dinamiche in gioco e stimolare lo sviluppo di meta-conoscenza sulla storia dei linguaggi di programmazione.

Più precisamente, nella Sezione 2 vedremo come gli elementi distintivi dell'evoluzione dei sistemi tecnologici si ritrovino anche nell'ambito dei linguaggi di programmazione. Nella Sezione 3 illustreremo i salti evolutivi identificabili nelle fasi più recenti della storia dei LP, mettendo in luce i concetti cruciali che ne hanno modellato lo sviluppo. Sarà esaminato il processo di ricerca di costrutti di programmazione di volta in volta più adatti al contesto storico e, facendo riferimento a linguaggi concreti, saranno evidenziate vere e proprie catene evolutive di linguaggi.

La Sezione 4 sarà dedicata all'analisi di pattern evolutivi sofisticati, come la co-evoluzione, i macro-trend, la costruzione della nicchia e l'exaptation. Concluderemo infine nella Sezione 5 con una discussione sull'uso del framework evolutivo in un contesto così lontano da quello in cui è nato.

2. I linguaggi di programmazione e l'evoluzione tecnologica

Lo studio dei linguaggi di programmazione si inserisce nel panorama più ampio dell'analisi dell'evoluzione della tecnologia, un settore dell'evoluzione culturale che è molto studiato in letteratura. Come l'evoluzione biologica, anche lo sviluppo tecnologico manifesta processi di variazione e selezione, ma anche di convergenza, contingenza ed estinzione. Si riconosce nel progresso tecnologico anche un pattern elaborato come l'esistenza di equilibri punteggiati, cioè la presenza di lunghi periodi di stasi interrotti da periodi di improvvisa e

rapida apparsa di numerosi elementi di novità (e.g. [11]). Allo stesso tempo, sussistono profonde differenze con i sistemi biologici, prima tra tutte il fatto che le innovazioni tecnologiche sono *progettate intenzionalmente*, sia con obiettivi immediati, ad esempio una nuova automobile più sicura della versione precedente, sia con obiettivi più a lungo termine, come quello di rendere sempre più pervasiva nella società la capacità di computazione, inizialmente tramite computer portatili, successivamente tramite smartphone, fino allo scenario dell'Internet of Things attualmente in costruzione. Va osservato che, a differenza della biologia, nel caso della tecnologia la progettazione esplicita delle novità tecnologiche è sostenuta anche dalla possibilità di avere una chiara nozione di *progresso misurabile*, in termini di maggiore efficienza, correttezza, sicurezza, velocità o minor costo economico o energetico.

Un altro aspetto ben noto dell'evoluzione è il cosiddetto *tinkering*, cioè il riuso e la ricombinazioni di elementi preesistenti secondo nuovi schemi. Anche in biologia organismi nuovi riusano elementi preesistenti in strutture nuove, ma mentre le nuove strutture biologiche restano in genere stabili per molto tempo, nel caso della tecnologia anche l'aggiunta di semplici novità può scatenare ricombinazioni di precedenti tecnologie con effetti molto ampi. I linguaggi di programmazione non sfuggono da questi aspetti chiave dell'evoluzione tecnologica, e ne offrono esempi interessanti. Se consideriamo infatti il linguaggio Java, osserviamo che il passaggio dalla versione 1.7 all'attuale versione 1.8 rappresenta un profondo cambiamento, sia tecnico (nuove primitive di programmazione e conseguenti profonde modifiche del sistema runtime) che nello stile di programmazione, integrando aspetti di programmazione funzionale nel modello ad oggetti che era distintivo della programmazione Java. Un tale cambiamento ha richiesto uno sforzo che è stato chiaramente studiato e progettato a fondo, e che è diventato necessario per rendere il linguaggio competitivo nello sviluppo di applicazioni che gestiscono efficientemente strutture dati di enormi dimensioni, come quelle richieste dalle applicazioni per i Big Data (ad esempio il grafo di una rete sociale). In questo cambiamento inoltre si ritrova anche un chiaro esempio di tinkering, infatti alcuni costrutti aggiunti a Java 8 non sono affatto nuovi, ma sono classici costrutti della programmazione funzionale già da tempo integrati nella programmazione ad oggetti in linguaggi come OCaml, C# o Scala. Però la ricombinazione dei nuovi costrutti con quelli distintivi del linguaggio Java permette (auspicabilmente) di estendere il successo delle tecnologie Java ai più moderni contesti applicativi, riguadagnando spazio nei settori attualmente più redditizi del mercato informatico.

Infatti, un elemento distintivo e spesso cruciale nell'evoluzione tecnologica, e in particolare nel caso dell'information technology, è l'impatto di *fattori economici e sociali*. Al giorno d'oggi la tecnologia, l'economia e i sistemi sociali sono profondamente interconnessi ed interdipendenti, ed ognuno di questi ambiti è in grado di incidere sulle prospettive di sviluppo degli altri. Si può dire che non siano solo tre ambiti che co-evolvono e si influenzano reciprocamente, ma si può pensare che costituiscano un vero e proprio *ecosistema*. Ad esempio nel caso dell'informatica esigenze di retro-compatibilità, così come richieste dettate

da mode o da aziende che dominano il mercato, spesso limitano o rendono impossibile la diffusione di nuove soluzioni, mantenendo la tecnologia dominante ad un livello sub-ottimale.

Ad esempio, a volte le applicazioni software vengono sviluppate usando tecnologie e linguaggi legati al web più per una scelta dettata da un trend che da un effettivo vantaggio rispetto all'uso di linguaggi general-purpose più solidi e classiche architetture client-server. Inoltre, innovazioni come Internet, il Cloud computing e i Big Data hanno avuto un impatto talmente forte sulla società che queste parole non corrispondono solo a termini scientifici, ma a vere e proprie keyword di analisi economiche e sociali. Infine, anche la scelta politica di vari governi nazionali di supportare programmi educativi di alfabetizzazione informatica (ad esempio in Italia il progetto ``Programma il Futuro'' [12]) è in grado di incidere sulle prospettive di sviluppo dell'ecosistema informatica-economia-società: si consideri ad esempio come la popolarità di Scratch, un linguaggio di programmazione pensato per i bambini, sia nettamente cresciuta ponendolo al 29-esimo posto nella classifica dei LP più popolari secondo l'indice TIOBE del gennaio 2016 [13].

Quanto detto finora illustra come gli elementi distintivi dell'evoluzione della tecnologia caratterizzino anche il processo di sviluppo dei linguaggi di programmazione. D'altra parte, seguendo la letteratura sull'evoluzione dei sistemi culturali, è possibile studiare in modo preciso fino a che punto i LP costituiscano uno specifico sistema evolutivo darwiniano. Per una discussione dettagliata di questi aspetti si veda ad esempio [14], mentre rimandiamo alla Sezione 4 la descrizione di come alcuni pattern distintivi della teoria dell'evoluzione si rintraccino anche nel caso dei LP offrendo un'ottica nuova per la loro analisi.

3. I salti evolutivi nella programmazione mainstream

Come anticipato nella Sezione 1, se da un lato è facile elencare cronologicamente le date di origine dei vari linguaggi di programmazione, raccontarne il loro processo di sviluppo in termini di una linea temporale è tutt'altro che scontato. Ad esempio, la timeline riportata in [15] racconta la storia di soli cinquanta LP, ma già rappresentare i cambiamenti di questi pochi linguaggi richiede un grafo abbastanza complesso. Lo studio della teoria dell'evoluzione insegna inoltre che la rappresentazione dei processi evolutivi richiede una topologia più complessa di quella lineare: in biologia si usano tipicamente gli alberi per rappresentare l'evoluzione filogenetica delle specie. Anche le analisi filogenetiche delle diverse lingue parlate dagli uomini hanno dimostrato che le loro storie si possono descrivere tramite un albero con diversi gruppi monofiletici [6,16]. Curiosamente, alcune di queste tecniche di ricostruzione filogenetica sono state applicate anche ai linguaggi di programmazione: partendo dai dati forniti da Wikipedia a proposito di quali LP hanno influenzato la creazione di quali altri LP, in [17] gli autori estraggono una filogenesi per un dataset di 347 linguaggi. In particolare, il loro metodo genera due alberi distinti, uno che rappresenta le "speciazioni" dei linguaggi funzionali e un secondo, separato, che rappresenta quelle dei linguaggi imperativi. In

aggiunta a questi alberi, viene prodotto anche un grafo che descrive le influenze orizzontali tra linguaggi. In realtà, per quanto storicamente corretti, due alberi separati per i linguaggi imperativi e i linguaggi funzionali non sono un risultato soddisfacente, soprattutto per i linguaggi moderni che sempre di più integrano questi paradigmi. Rimandiamo a [14] per una discussione dettagliata dei risultati in [17], e osserviamo che già il grafo in [15] citato sopra mostra come la rappresentazione ad albero sia inadeguata: lo sviluppo dei LP rivela infatti una vera e propria *rete* di innovazioni con pattern di diversificazioni e fusioni multiple, che richiama piuttosto le complesse filogenesi reticolari tipiche dei batteri [5].

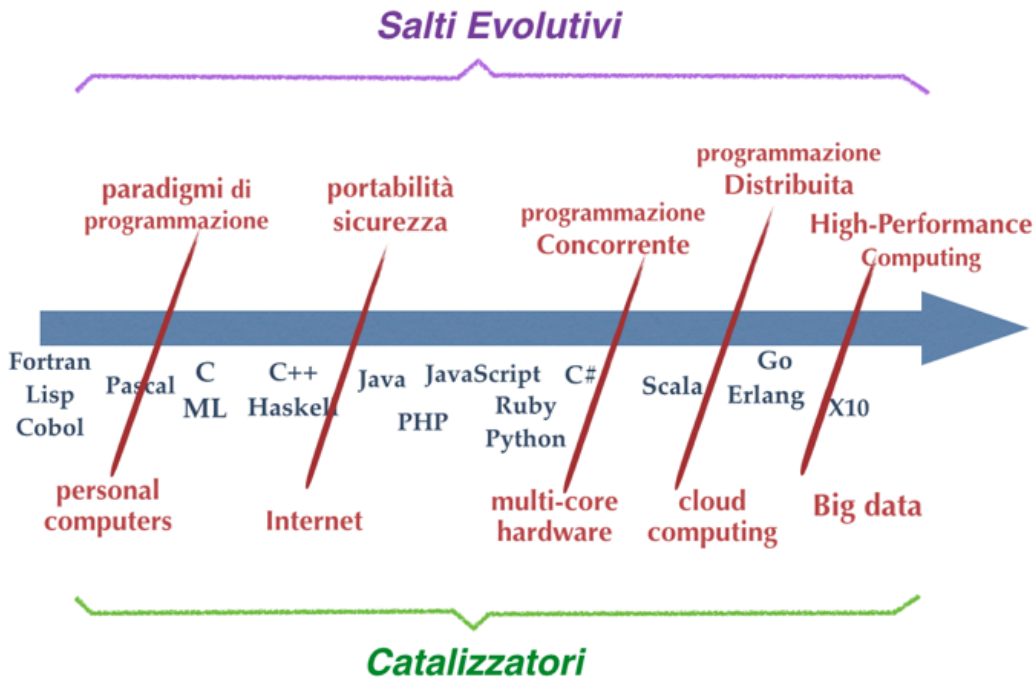

Figure 1 I salti evolutivi nella programmazione mainstream

Se dunque è impraticabile linearizzare l'evoluzione dei linguaggi di programmazione, è invece possibile disporre su una linea temporale i principali salti evolutivi nel loro sviluppo storico. Osserviamo innanzitutto che, come spesso avviene nello sviluppo tecnologico, anche nel caso della programmazione mainstream ogni grande cambiamento richiede un catalizzatore, qualcosa cioè che funga da incentivo forte a cercare nuove soluzioni, magari ripescandole tra (o ricombinandole con) innovazioni rimaste per molto tempo un po' in sordina.

L'excursus storico-evolutivo tracciato in questa sezione è quindi riassumibile nella Figura 1, che illustra la sequenza di salti evoluti riscontrabili nella storia dei

LP con i corrispondenti catalizzatori che hanno effettivamente innescato questi salti. Sulla linea temporale sono anche elencati una serie (molto parziale e incompleta ma comunque indicativa) di linguaggi di programmazione che hanno conosciuto un picco di popolarità in quella fase. Coerentemente con quanto discusso sopra, l'ordinamento della figura non è affatto cronologico: ad esempio Python è un linguaggio introdotto (poco) prima di Java, e la gestione dei problemi di concorrenza è stata chiaramente affrontata prima della nascita di Internet.

3.1 Le prime rivoluzioni

Tra gli anni '50 e '60, con l'introduzione dei linguaggi come Fortran, Lisp, Cobol e Algol, si può iniziare a parlare di LP moderni che, rispetto ai precedenti linguaggi più direttamente legati all'hardware sottostante, introducono primitive più astratte, come `if`, `goto`, `continue`, in grado di dare struttura ai programmi. Negli anni '80 la rivoluzione dei personal computer, che cominciano a diffondersi in maniera capillare, rende la programmazione un'attività sociale, che richiede linguaggi general-purpose e che astraggono maggiormente i dettagli strutturali. In questo contesto, il successo di linguaggi come C, Simula, Smalltalk, ML e Prolog denota l'inizio della diversificazione tra *paradigmi* di programmazione: imperativo, funzionale, logico e orientato agli oggetti. Il caso della programmazione ad oggetti (OOP) è uno di quegli esempi di tecnologie che restano in sordina per un po' prima di conoscere un successo eccezionale: la OOP nasce infatti negli anni '60 in ambiente accademico, ma diventa il vero paradigma dominante nella programmazione mainstream solo negli anni '90, quando i sistemi software industriali si sono fatti così complessi che le capacità di modularità, data hiding e riuso del codice distintivi della OOP, diventano una necessità per poter gestire software complesso in modo economico e affidabile [18].

Un altro cambiamento fondamentale nel panorama dei LP è avvenuto con la comparsa di Internet, e in particolare con la scoperta delle nuove opportunità commerciali che la rete ha portato con sé. Innanzitutto la possibilità di mettere in comunicazione macchine diverse ha spostato l'obiettivo dei LP dalla velocità ed efficienza di esecuzione verso la portabilità (la capacità di far eseguire un programma in un ambiente di esecuzione distinto da quello in cui è stato scritto) e la sicurezza. In questo scenario il linguaggio Java e il bytecode della sua macchina virtuale (JVM) hanno portato al successo concetti di portabilità già noti ad esempio in linguaggi come ML e Smalltalk. La successiva crescita del Web al di sopra della rete Internet ha poi sostenuto la popolarità dei cosiddetti linguaggi di scripting, come PHP, JavaScript, Python o Ruby, che ben si prestano a sviluppare agilmente piccoli programmi (i.e. script) da inserire in pagine web e server web.

Il successivo salto evolutivo nella timeline corrisponde alla popolarità della programmazione concorrente, che offre cioè la capacità di programmare più flussi di controllo che eseguono contemporaneamente. Come per l'OOP, anche in questo caso LP concorrenti esistono fin dagli anni '60, ma la loro diffusione è rimasta limitata perché programmare correttamente la gestione di più attività che procedono insieme è molto più complesso che scrivere codice che esegue in

sequenza. La situazione è però cambiata all'inizio del 2000, quando il costante avanzamento tecnologico, che fino ad allora aveva garantito la continua costruzione di processori hardware sempre più veloci, ha raggiunto il suo limite fisico: per avere delle prestazioni migliori non era più possibile ottenere una CPU più efficiente, ma bisognava cambiare il modello di architettura dei processori, aggiungendo più CPU nella stessa macchina e facendole cooperare sempre meglio. Nacquero così i processori multicore, e da allora l'hardware parallelo è diventato a tutti gli effetti lo standard attuale. Questo cambiamento si è rivelato dunque il vero catalizzatore per la programmazione concorrente, perché solo i programmi che definiscono e coordinano (in modo corretto ed efficiente) le loro sottoattività concorrenti sono in grado di beneficiare a pieno del parallelismo dell'hardware sottostante.

3.2 L'evoluzione di alcuni costrutti di programmazione

Prima di parlare della fase più recente dello sviluppo dei linguaggi, ci addentriamo in questa sezione un po' più nei dettagli concreti dell'evoluzione di alcuni costrutti di programmazione, un po' come, in biologia, oltre a ripercorrere i cambiamenti delle specie, si può osservare l'evoluzione (attraverso varie specie) di specifici tratti, come le ali o la coda.

Ogni paradigma, o stile, di programmazione ha infatti dei tratti fondamentali, che si traducono in precisi costrutti di programmazione, come le funzioni e i dati immutabili nella programmazione funzionale o gli oggetti e l'ereditarietà nella OOP. Questi costrutti non sono univoci, ed evolvono nel tempo con l'evolvere dei linguaggi. Ad esempio si nota una chiara evoluzione dei costrutti di programmazione ad oggetti guardando la linea evolutiva che porta dal linguaggio C++ a Java fino al linguaggio Scala. Innanzitutto, l'incapsulamento dei dati, principio cardine della OOP, era sostanzialmente violato dalle `friend`-functions di C++, che sono state eliminate in Java, per arrivare poi ad adottare in Scala lo Uniform Access Principle che prevede l'uso di una sintassi talmente uniforme da mantenere totale l'incapsulamento dell'implementazione degli oggetti. Inoltre, il classico problema dell'ereditarietà multipla nella OOP era stato affrontato in C++ introducendo l'ereditarietà virtuale nelle gerarchie a diamante, per passare poi ad una soluzione molto diversa in Java usando il costrutto delle Interfacce, fino alla scelta, in qualche modo intermedia, di Scala basata sui mixins.

Nel caso della programmazione concorrente la scelta dei costrutti è maggiormente critica perché la concorrenza gioca un ruolo cruciale a diversi livelli di astrazione: c'è concorrenza a livello di architettura hardware, di sistema operativo (che gestisce contemporaneamente diversi processi), di supporto runtime del linguaggi (ad esempio la JVM consiste di diversi thread concorrenti), di sintassi del LP, fino al livello logico di progettazione di un algoritmo concorrente. I costrutti di concorrenza di un linguaggio devono quindi essere allo stesso tempo sufficientemente astratti per permettere di tradurre in un programma la logica di un algoritmo concorrente, e sufficientemente snelli da poter essere implementati in modo efficiente e scalabile. Ad esempio, il modello a thread originale di Java prevedeva di usare la classe `Thread` per gestire

contemporaneamente sia il nuovo flusso di esecuzione concorrente sia l'attività logica da fargli eseguire. Dalla versione 1.5 di Java, con le classi di tipo `Executor`, è possibile disaccoppiare la definizione delle attività logiche da quella dei meccanismi della loro esecuzione, e decidere politiche di esecuzione molto più efficienti utilizzando ad esempio pool fatti di pochi thread della JVM per eseguire concorrentemente numerose attività logiche.

L'intrinseca difficoltà del ragionamento e della programmazione concorrente fanno sì che al giorno d'oggi siamo ancora essenzialmente in una fase di "sperimentazione": nuovi costrutti o nuovi mix di costrutti esistenti vengono adottati dai linguaggi sulla base dei loro ambiti di applicazione. Rimandiamo a [19] per una discussione più dettagliata, ricordando qui brevemente i tre principali modelli di concorrenza a cui si rifanno maggiormente gli attuali linguaggi mainstream. Il modello più diffuso, utilizzato per esempio in Java, C# e C++11, è quello a *memoria condivisa*, in cui le diverse attività concorrenti comunicano sincronizzandosi su uno stato condiviso. Questo modello ben si armonizza con lo stile imperativo di questi linguaggi, ed è molto naturale in un approccio data-centrico, in cui un controllo centralizzato coordina le diverse attività che operano su dati condivisi. D'altra parte la gestione corretta degli accessi a questi dati condivisi si dimostra molto difficile, ed errori come interferenze, datarace, deadlock, inversioni di priorità sono difficili da eliminare completamente anche ricorrendo a test estensivi. Il modello a *scambio di messaggi* è invece naturale nei linguaggi funzionali: lo stato della computazione non è più condiviso e modificabile, ma viene trasformato e comunicato tra le diverse attività concorrenti. Questo modello permette una programmazione più dichiarativa e con meno rischi di interferenze, ma richiede una carico aggiuntivo per la gestione delle comunicazioni. Si inseriscono in questo modello i cosiddetti sistemi di Attori, tipici ad esempio dei linguaggi Erlang e Scala usati rispettivamente in progetti come Facebook Chat o WhatsApp e LinkedIn o Ebay.

Il terzo modello di concorrenza, attorno a cui è di recente esploso l'interesse sia della comunità di ricerca che dell'industria, si basa sull'uso dei processori grafici (GPU) che offrono un'architettura massicciamente parallela tale da supportare con grande efficienza classi di applicazioni che lavorano su grandi moli di dati, come simulazioni fisiche, ray tracing, applicazioni bioinformatiche o di intelligenza artificiale, ricostruzioni 3D fino alle previsioni degli andamenti delle azioni finanziarie. Questo tipo di programmazione, detta General Purpose GPU-Programming richiede però specifici algoritmi e specifici linguaggi di programmazione, attualmente ancora piuttosto legati al tipo di architettura su cui sono implementati. Il crescente interesse su questi temi sta però facendo nascere sempre più progetti di sviluppo di LP di più alto livello di astrazione al di sopra di queste tecnologie, e.g. CUDA 6, OpenAcc, Copperhead.

3.3. Le sfide più recenti

Se lo sviluppo di hardware come i processori multicore e le GPU ha catalizzato la popolarità della programmazione concorrente, più di recente le nuove tecnologie di cloud computing hanno fatto da catalizzatore per la diffusione della *programmazione distribuita*. Anche in questo caso i sistemi distribuiti non

sono certo una novità, insieme ad un gran numero di tecniche di programmazione correlate, come la comunicazione tramite socket, le Remote Method Invocation, o i più recenti Grid computing e Service Oriented Architectures. D'altra parte la comparsa sul mercato di risorse cloud, cioè grandi spazi di memoria o risorse di calcolo come cluster di computer, a prezzi accessibili ha determinato un cambiamento importante, arrivando a modificare addirittura il modello di business delle aziende. Invece di un service provider acceduto secondo il modello client-server, si è passati al cosiddetto Software As A Service model, in cui le applicazioni sono realizzate come servizi disponibili su un ampio spettro di piattaforme che va dai piccoli dispositivi mobili fino a grandi cluster che offrono migliaia di processori multicore.

Questo scenario richiede soluzioni specifiche per gestire aspetti chiave come la scalabilità (cioè l'aumento o la diminuzione delle risorse hardware impegnate a seconda dell'aumento o diminuzione del carico di lavoro da effettuare), l'eterogeneità dell'hardware impiegato, la tolleranza ai guasti (sia software che hardware), la responsività, la sicurezza e la privacy. Tutti questi aspetti di programmazione distribuita non sono nuovi, ma con il cloud computing riemergono ad un più critico ordine di grandezza e richiedono certamente nuove soluzioni sia a basso livello, di architettura e di sistema, ma anche linguaggi di programmazione con adeguati costrutti per la gestione elastica della distribuzione. In questo contesto sembra essere particolarmente attrattivo il paradigma della *programmazione reattiva* [20], che promuove in particolare l'uso di costrutti asincroni, cioè di istruzioni che disaccoppiano la richiesta di effettuare un'operazione dalla sua esecuzione. In questo modo quindi la componente software che ha chiesto un'operazione non viene bloccata in attesa che l'operazione si concluda, determinando una maggiore efficienza e scalabilità delle prestazioni del sistema.

Infine, l'ultimo cambiamento riportato in Figura 1 rappresenta la sfida informatica più attuale, cioè la capacità di operare con le enormi moli di dati prodotti continuamente dalle reti sociali e dai dispositivi smart, sempre più pervasivi, come gli smatphone, i tablet, le smart-TV, i dispositivi di domotica e ogni oggetto dotato di tecnologia NFC (Near Field Communication) capace di attivarsi e scambiare dati con dispositivi di computazione presenti nelle vicinanze. La capacità di analizzare questi dati, in inglese "big data analytic", rappresenta la vera killer application per quello che si chiama *High Performance Computing* (HPC), cioè quel modello di programmazione fortemente orientato alla scalabilità su hardware massicciamente parallelo, inclusi i supercomputer con migliaia di processori.

È ancora troppo presto perché l'HPC sia considerato mainstream, ma citiamo come esempio X10: un linguaggio di programmazione open-source sviluppato da IBM Research che estende i costrutti tipici della OOP con nuove primitive specifiche. In particolare, X10 usa il costrutto fondamentale di *place*: un'astrazione che rappresenta un nodo virtuale di computazione che può essere mappato indifferentemente su un computer all'interno di un cluster o su un core all'interno di un supercomputer. Un programma X10 gestisce quindi in modo centralizzato un gran numero di attività di calcolo distribuite in un insieme di

place, ed è in grado di lavorare con collezioni di dati così grandi da dover essere memorizzate a pezzi su più place. Si arriva così a gestire in modo agevole codice che gira su migliaia di nodi multicore.

Per concludere, osserviamo che se già la programmazione concorrente e distribuita coinvolgevano diversi livelli di astrazione, nell'HPC lo scarto tra la logica delle applicazioni per big data e l'infrastruttura di esecuzione è particolarmente ampio e richiede dunque degli specifici costrutti che rendano possibile allo stesso tempo una programmazione dichiarativa, cioè orientata al problema da risolvere, e un'implementazione estremamente efficiente. Due pattern di programmazione particolarmente efficaci in questo contesto sono il modello *map-reduce* e il modello *bulk synchronous parallel* (BSP). Il primo, implementato ad esempio dalle tecnologie Google's MapReduce, Apache Hadoop e Apache Spark, riusa i classici combinatori map e reduce della programmazione funzionale per esprimere algoritmi di ordinamento e ricerca in modo massicciamente scalabile (e.g. [21,22]). Il modello BSP, implementato ad esempio da Google's Pregel e Apache Giraph, si dimostra particolarmente interessante per la sua capacità di lavorare con grafi di dimensioni enormi, come i grafi sociali o geografici, sfruttando al massimo la distribuzione del carico di lavoro tra diverse macchine (e.g. [23,24]).

4. I pattern evolutivi rintracciati nella storia dei linguaggi di programmazione

Questa sezione illustra come alcuni dei pattern esplicativi sviluppati all'interno della teoria dell'evoluzione si possano riconoscere anche nello sviluppo storico dei linguaggi di programmazione. Senza voler forzare l'analogia con l'evoluzione biologica, è interessante osservare come questi pattern, pur in un contesto molto diverso da quello in cui sono nati, riescano a mettere in luce alcune delle dinamiche che entrano in gioco nel mondo dei LP.

Co-evoluzione

Analogamente alla co-evoluzione del linguaggio umano e della struttura del cervello dell'uomo, i linguaggi di programmazione hanno indubbiamente una storia di co-evoluzione con lo sviluppo delle tecnologie hardware. Infatti, molti dei salti evolutivi che abbiamo visto nella sezione precedente hanno come catalizzatore proprio degli avanzamenti delle architetture hardware. Nel caso dei LP esiste però una seconda importante linea di co-evoluzione, che non ha un corrispondente biologico: gli avanzamenti della ricerca teorica. Infatti le tecniche di programmazione mainstream e la ricerca teorica sui LP si sono mutuamente influenzate, spesso con contaminazioni tra soluzioni teoriche e pratiche nate a distanza di tempo. Nella sua Turing lecture [25], Robin Milner osserva che le astrazioni di programmazione più appropriate nascono proprio da una *dialettica* tra i test sperimentali condotti dalla programmazione pratica e gli approfonditi test matematici portati avanti dall'approccio teorico. Il ruolo della ricerca teorica nell'evoluzione dei LP è quindi essenziale per il metodo dialettico identificato da Robin Milner. I linguaggi formali studiati nell'approccio teorico sono infatti adatti a testare nuove primitive di programmazione e nuovi modi di comporre diverse

primitive, il tutto in un modello conciso ed espressivo. In altri termini, permettono di fare *sperimentazione in un ambiente controllato*, testando nuovi "tratti" di un linguaggio e promuovendo dunque specifiche "mutazioni" nei linguaggi. Questa capacità di sperimentare, progettare e testare possibili mutazioni non ha eguale né in biologia né nell'evoluzione del linguaggio umano.

Trend Macro-evolutivi

In biologia un trend macro-evolutivo è uno sviluppo che avviene in modo trasversale in diverse specie; ad esempio le dimensioni fisiche del cervello sono aumentate parallelamente nell'evoluzione di specie distinte di ominidi. Nel contesto dei linguaggi di programmazione si assiste ad un progressivo innalzamento del livello di astrazione dei linguaggi mainstream. Nuovi linguaggi, così come le nuove release dei linguaggi più classici, supportano un stile di programmazione più dichiarativo, che si focalizza cioè più sul "cosa fare" che sul "come fare" [19]. I dettagli sull'implementazione del "come fare" vengono progressivamente rimossi dal controllo del programmatore (o mascherati) e affidati ai supporti runtime dei linguaggi, che infatti si fanno via via più complessi. Si pensi ad esempio alla gestione della memoria e a come la possibilità del C++ di controllare direttamente l'allocazione e la deallocazione della memoria sia stata abbandonata in Java per affidarla alla gestione automatica, meno precisa ma più sicura, del garbage collector offerto dal supporto runtime. Come ulteriore esempio si consideri come è cambiato il modo di operare sulle collezioni di dati: supponiamo ad esempio di voler identificare all'interno di una lista di persone l'età dell'uomo più anziano. L'approccio mainstream di un tempo, illustrato dal seguente codice Java, consisteva nel programmare esplicitamente lo scorrere di ogni persona nella lista verificando se si tratta di un maschio e aggiornando il massimo corrente ogni volta che si trova un uomo con un'età maggiore di quella incontrata in precedenza:

```
Persona[] elenco = ... // inizializzazione dell'elenco
int max = -1;
for (int i=0; i< elenco.length; i++)
   if(elenco[i].getGenere()==MASCHIO && elenco[i].getAnni() >max)
     max = elenco[i].getAnni();
```

L'approccio offerto invece dai costrutti più dichiarativi dei linguaggi moderni permette di manipolare direttamente la lista di persone tramite ad esempio operazioni di filtraggio, chiedendo quindi di indicare nel programma unicamente il criterio secondo cui filtrare la lista. Utilizzando i nuovi costrutti di Java 8 il codice precedente si può riscrivere come segue, dove l'elenco di persone viene filtrato in un elenco di soli uomini, trasformato nell'elenco delle età di tali uomini ed infine ne viene estratto il massimo:

```
Collection<Persona> elenco = ... // inizializzazione dell'elenco
int max = elenco.stream().filter( p-> p.getGenere()==MASCHIO )
                         .mapToInt( p-> p.getAnni())
                         .max();
```

Lo spostamento di molti linguaggi verso uno stile più dichiarativo dipende dal fatto che l'uso di costrutti di programmazione più astratti permette di scrivere programmi più chiari e compatti quindi più corretti e più facilmente mantenibili. Ma è importante ricordare che ciò è stato possibile grazie al contemporaneo sviluppo di hardware sempre più efficiente, che ha permesso di supportare la crescita della complessità degli strati software di cui sono composti i supporti runtime dei linguaggi, a cui sono stati progressivamente lasciati i dettagli che al programmatore non si chiede più di indicare.

Costruzione della nicchia

I moderni sistemi software, come ad esempio quelli delle grandi aziende Internet-driven come Google, Facebook o LinkedIn, non sono scritti usando un solo linguaggio di programmazione, ma sono formati da diversi strati e moduli software scritti usando diversi linguaggi, che interoperano tra loro a vari livelli di astrazione. In alcuni casi, tra cui in particolare le applicazioni Web, i diversi linguaggi coinvolti sono appositamente pensati per essere usati insieme, formando così quello che potremmo chiamare un *ecosistema di linguaggi*. Ad esempio, siti web ricchi e dinamici come quelli di Twitter, Amazon o eBay, richiedono lo sviluppo di un back-end, che effettua il grosso del lavoro tipicamente consultando un database, e un front-end che si occupa dell'interazione dinamica con l'utente. Avendo scopi molto diversi, questi due lati dell'applicazione sono spesso scritti usando linguaggi distinti. Ma anche solo lo sviluppo del lato front-end generalmente coinvolge tre diversi linguaggi di programmazione: HTML5 per gestire il contenuto delle pagine web, CSS per gestire l'aspetto delle pagine, e JavaScript per gestire il comportamento logico delle pagine. L'uso di questi tre linguaggi è talmente interconnesso da poter riconoscere un effetto di costruzione della nicchia, cioè la modifica/evoluzione di uno dei tre provoca un cambiamento nel contesto d'uso degli altri che retroagisce e impatta sulla loro evoluzione.

Exaptation

Con il termine exaptation si intende il processo evolutivo in cui un tratto evolutosi con una certa funzione viene successivamente riusato con una funzionalità diversa. Ad esempio, alcune specie di dinosauri erano coperte di piume probabilmente per riscaldarsi, ma nell'evoluzione in uccelli le piume sono state "riusate" per volare.

Un interessante esempio di exaptation si può riscontrare nel mondo della programmazione osservando che dopo circa cinquant'anni in cui i linguaggi di programmazione funzionali sono rimasti in secondo piano, al giorno d'oggi il loro tratto distintivo, cioè l'uso delle funzioni come valori first-class, rappresenta la novità più interessante dei linguaggi mainstream. In questa nuova fase infatti le funzioni diventano il fulcro su cui si poggiano alcuni meccanismi di programmazione concorrente e asincrona, come ad esempio i futures, le callback o la programmazione reattiva. Per capire come sia avvenuta questa exaptation è utile osservare che il concetto di *funzione* può essere riletto come un'astrazione che rappresenta un *comportamento*: come una funzione un comportamento è componibile con altri comportamenti per ottenere

comportamenti complessi, e può essere comunicato tra diversi moduli software e condiviso tra diversi flussi logici concorrenti. Inoltre, mentre la programmazione imperativa richiede di pensare in termini di *tempo*, la programmazione funzionale si basa sul concetto di *spazio*, descrivendo cioè l'avanzamento della computazione in termini di trasformazioni di valori tramite (composizioni di) funzioni, al posto del susseguirsi temporale di comandi che modificano lo stato della memoria. Progettare i sistemi concorrenti in termini di cambiamenti spaziali invece che temporali rende più facile gestire il nondeterminismo intrinseco a questi sistemi, generando quindi software più corretto. Inoltre questa visione spaziale si armonizza bene con la capacità di strutturare il software tipica della programmazione orientata agli oggetti. Come abbiamo visto nella sezione precedente, la capacità di supportare efficacemente la programmazione concorrente, in particolare per operare efficientemente sulle enormi collezioni di dati, è un fattore critico per il successo di un linguaggio, che è riuscito infatti a catalizzare dei cambiamenti profondi in linguaggi come C++ e Java, passati ai nuovi standard C++11 e Java 8 anche per introdurre le funzioni come valori first-class (chiamandole lambda-expression).

5. Conclusione

In questo articolo abbiamo visto come il linguaggio della teoria dell'evoluzione possa essere usato efficacemente per mettere in luce diversi aspetti dello sviluppo storico dei linguaggi di programmazione. Per capire però fino a che punto questa teoria rappresenti un framework esplicativo corretto per questo ambito è necessario approfondire l'analisi di come i processi di variazione, selezione ed ereditarietà operano nel contesto biologico e in quello culturale/ tecnologico. Se da un lato esiste molta letteratura sull'evoluzione della tecnologia in generale, o su ambiti specifici come l'evoluzione di specifici manufatti o dei brevetti tecnologici, ancora poco è stato studiato il caso dei sistemi software e dei linguaggi di programmazione. Sarebbe invece stimolante rispondere a domande come cos'è una mutazione in un LP o esiste una lotta per la sopravvivenza (nel mercato del software) tra linguaggi? Che ruolo ha il mercato o le strategie di marketing nella selezione tra LP? Quanto conta la plasticità e la flessibilità dei principi di progettazione di un LP per la sua sopravvivenza? Il profondo cambiamento che è stato richiesto per passare dalla versione 1.7 di Java alla nuova versione 1.8 rappresenta un ottimo esempio di adattamento al contesto e di lotta per la sopravvivenza.

In conclusione, applicare la teoria darwiniana a sistemi culturali o sociali, per quanto accattivante, ha il grosso rischio di essere una forzatura: le analogie fenomenali non vanno scambiate per spiegazioni e non vanno fatti trasferimenti teorici grossolani. D'altra parte, quando si guarda allo sviluppo storico di un sistema, che sia biologico, culturale o tecnologico, c'è sempre in gioco un intreccio interessante di dinamiche diverse, e il confronto (cauto) con la ricchezza esplicativa della teoria dell'evoluzione è un modo per cercare di identificare e di fare ordine tra tutte queste dinamiche. Considerare quindi la teoria dell'evoluzione come un toolbox di pattern esplicativi può offrire uno

strumento per arricchire l'analisi anche dell'attuale esplosione delle tecnologie software.

Bibliografia

[1] Wikipedia: Measuring programming language popularity. https://en.wikipedia.org/wiki/Measuring_programming_language_popularity (ultimo accesso Gennaio 2016).

[2] TIOBE Index for Objective-C http://www.tiobe.com/index.php/content/paperinfo/tpci/Objective_C.html (ultimo accesso Gennaio 2016).

[3] Arthur, B.(2009). The nature of technology. What it is and how it evolves. Free Press.

[4] Johonson, S.(2010). Where good ideas come from: a natural history of innovation. Riverhead Press.

[5] Solè, R., Valverde, S., Rosas-Casals, M., Kauffman, S., Farmer, D., Eldredge, N.(2013). ``The evolutionary ecology of technological innovation'', *Complexity*, 18, 15-27.

[6] Pagel, M.(2009). ``Human language as a culturally transmitted replicator'', *Nature Reviews Genetics*, 10, 405-415.

[7] Atkinson, Q.D., Meade, A., Venditti, C., Greenhill,S.J., Pagel,M., (2008). ``Languages Evolve in Punctuational Bursts'', *Science*, 319, 588.

[8] Laland, K., Wray, G.A., Hoekstra, H.E.,(2014). ``Does evolutionary theory need a rethink?'', *Nature*, 514, 161-164.

[9] Claidiere, N., Scott-Phillips T.C., Sperber, D. (2014). ``How Darwinian is cultural evolution?'', *Philosophical Transactions of the Royal Society B Biological Sciences*, 369: 20130368.

[10] Lane, D., Maxfield,R., Read, D., van der Leeuw, S., (2009). "From Population to Organization Thinking", *Complexity Perspectives in Innovation and Social Change, Methodos* Series 7, Springer.

[11] Loch, C.H., Huberman, B.A.,(1999). "A Punctuated-Equilibrium Model of Technology Diffusion'', *Management Science*, 45(2),160-177.

[12] Programma il Futuro. MIUR e CINI http://www.programmailfuturo.it/

(ultimo accesso Gennaio 2016).

[13] TIOBE Index for January 2016, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (ultimo accesso Gennaio 2016).

[14] Crafa, S., (2015). "Modelling the Evolution of Programming Languages".

Technical Report arXiv:1510.04440 2015.

[15] http://www.levenez.com/lang/ (ultimo accesso Gennaio 2016).

[16] Mace, R., Holden, C.J.,(2005). "A phylogenetic approach to cultural evolution". *Trends in Ecology and Evolution* vol. 20: 116-121.

[17] Valverde, S., Solè, R.,(2015). "Punctuated equilibrium in the large-scale evolution of programming languages", *Journal of The Royal Society Interface*, vol. 12(107).

[18] Sutter, H. "The free lunch is over: A fundamental turn toward concurrency in software". Dr. Dobb's (online) Journal 30-3:

http://www.gotw.ca/publications/concurrency-ddj.htm. (ultimo accesso Gennaio 2016).

[19] Crafa, S.,(2015). "The role of concurrency in an evolutionary view of programming abstractions", *Journal of Logical and Algebraic Methods in Programming* vol 84:732-741.

[20] The Reactive Manifesto, http://www.reactivemanifesto.org/ and http://typesafe.com/blog/reactive-manifesto-20 (ultimo accesso Gennaio 2016).

[21] Czajkowski, G., Dvorsky M., Zhao A.J., Conley, M.,(2011). "Sorting petabytes with MapReduce - the next episode". Available at

http://googleresearch.blogspot.com/2011/09/ (ultimo accesso Gennaio 2016).

[22] Goodrich, M.T., Sitchinava, N., Zhang, Q., (2011). "Sorting, searching, and simulation in the mapreduce framework", *Algorithms and Computation*: (7074), pp. 374-383.

[23] Bisseling, R.H., McColl, W.F. (1994). "Scientific computing on bulk synchronous parallel architectures". *Proceedings of the 13th IFIP World Computer Congress:*(1), pp. 509-514.

[24] Valiant, L.G.,(2011). "A bridging model for multi-core computing". *Journal of Computer and System Sciences*: 77(1), pp. 154-166.

[25] Milner, R., (1993). "Elements of Interaction - Turing Award Lecture", *Communication of the ACM*, vol. 36(1): 78-89.

Glossario

mutazione genetica: modifica stabile ed ereditabile del meriale genetico, dovuta ad agenti esterni o al caso ma non alla ricombinazione genetica derivante dalla riproduzione. Le mutazioni genetiche rappresentano il principale meccanismo di variazione degli individui, su cui poi agisce la selezione naturale.

trasferimento genico orizzontale: processo secondo cui una cellula trasferisce materiale genetico ad un' altra cellula non discendente. Si oppone al trasferimento verticale, ovvero la riproduzione, secondo cui un organismo riceve il materiale genetico dai suoi genitori. Il trasferimento genico orizzontale è molto comune fra differenti specie di batteri, ed influenze orizzontali sono tipiche dell'evoluzione dei sistemi culturali, come nel caso dello sviluppo dei diversi linguaggi umani.

selezione naturale: "La conservazione delle differenze e variazioni individuali favorevoli e la distruzione di quelle nocive" (C.Darwin, L'origine delle specie)

contingenza: mentre un processo casuale non ha nessuna regolarità e nessun principio di causa-effetto, un processo contingente è un processo imprevedibile perche' determinato da un intreccio complesso di diversi fattori. Il processo evolutivo non è un processo puramente casuale, ma un processo storico in cui sarebbe bastato cambiare di pochissimo qualche elemento del contesto e si sarebbero ottenuti effetti radicalmente diversi.

convergenza evolutiva: fenomeno per cui specie diverse che vivono nello stesso tipo di ambiente si evolvono sviluppando indipendentemente strutture o adattamenti che li portano ad assomigliarsi moltissimo. Ad esempio gli squali e i delfini appartengono a specie (e classi) distinte (i primi sono pesci mentre i secondi sono mammiferi) ma convergenti verso una forma molto simile.

plasticità fenotipica: capacità di un organismo con un dato genotipo di cambiare il suo fenotipo in risposta ai cambiamenti nell'ambiente, inclusi i cambiamenti che si verificano durante la vita adulta dell'organismo. Ad esempio la capacità di un insetto di deporre uova di colori diversi a seconda del sito in cui vengono deposte, aumentando così le possibilità di sopravvivenza della prole in ambienti altamente variabili.

speciazione: processo evolutivo grazie al quale si formano nuove specie da quelle preesistenti. L'opposto della speciazione è l'estinzione di una specie.

filogenesi: processo di ramificazione delle linee di discendenza nell'evoluzione delle specie. Le ricostruzioni filogenetiche ricostruiscono le relazioni di parentela evolutiva tra i diversi organismi.

gradualismo filetico: teoria secondo cui vi è una differenza genetica impercettibile tra una generazione e la successiva, e le grandi modificazioni sono la somma di piccoli cambiamenti generati dalla selezione naturale.

teoria degli equilibri punteggiati: teoria secondo cui i cambiamenti evolutivi avvengono con lunghi periodi di stasi interrotti da periodi di improvvisa e rapida apparsa di numerosi elementi di novità. Questa teoria si oppone a quella del gradualismo filetico, ma in realtà l'evoluzione della vita sembra aver dato spazio ad entrambi i pattern, che quindi non si escludono.

teoria della costruzione della nicchia: teoria secondo cui l'ambiente non solo seleziona gli organismi ma, almeno in parte, viene continuamente costruito e modificato dai viventi. Tramite la modifica del loro ambiente di vita, gli organismi lasciano quindi alle generazioni successive anche un'eredità non genetica, che retroagisce sulle dinamiche evolutive della specie. Ad esempio effetti della costruzione della nicchia si ritrovano nell'evoluzione di alcuni uccelli che fabbricano nidi particolari, ma anche nell'evoluzione dell'uomo, che è capace di modificare profondamente il proprio ambiente di vita.

co-evoluzione: processo di evoluzione congiunto di due o più specie che interagiscono tra loro tanto strettamente al punto da costituire ciascuna un forte fattore selettivo per l'altra, col risultato di influenzarsi vicendevolmente. Ad esempio le forme di alcuni fiori co-evolvono insieme alla forma dei loro insetti impollinatori.

trend macro-evolutivi: cambiamenti evolutivi che avvengono in modo trasversale in diverse specie di organismi. Ad esempio l'aumento della complessità degli organismi o la crescita delle dimensioni fisiche degli animali avvenuta in determinati periodi geologici.

exaptation: processo evolutivo in cui un tratto evolutosi con una certa funzione viene successivamente riusato con una funzionalità diversa. Ad esempio le piume usate da alcuni dinosauri con una probabile funzione di isolamento termico, sono state poi un tratto indispensabile per permettere agli uccelli, che discendo dai dinosauri, di volare.

Biografia

Silvia Crafa è ricercatrice confermata presso il Dipartimento di Matematica dell'Università di Padova. La sua attività, sia di ricerca che di didattica specialistica, si focalizza sulle tecniche di analisi formale dei linguaggi di programmazione e sulla teoria dei sistemi concorrenti. È autrice di numerose pubblicazioni su prestigiose riviste internazionali e collabora con diverse istituzioni straniere. È membro dell'International Federation for Information Processing Working Group n.1.8 dedicato alla teoria della concorrenza.

email: crafa@math.unipd.it