Il nome di Alexander Stepanov, insieme a quello di Meng Lee e di David Musser, è legato a STL, la libreria standard di template del C++

INTERVISTA A ALEXANDER STEPANOV

di Graziano Lo Russo

Alex Stepanov è il principale ispiratore di STL, la libreria standard di template del C++. Più entro nei dettagli di STL, per cercare di emularla, e più mi rendo conto di quanto sia uno stile di progetto e di programmazione del tutto diverso da qualunque altra cosa abbia mai visto in altri linguaggi: STL non assomiglia al C/C++ classico, non più di quanto assomigli a Lisp, Ada o SmallTalk. La mia maggiore curiosità nell'intervistare Stepanov era di capire da dove fossero nate queste idee. La mia maggiore sorpresa è stata scoprire che la programmazione generica ha una solidissima base di logica matematica, sia pure non "a oggetti" e che il C++, nelle parole di Stepanov, è il linguaggio di programmazione che meglio permette di esprimere la programmazione generica: altro che "assembler a oggetti", come lo chiama qualcuno! Ma lasciamo che Stepanov si presenti da solo.

Alexander, vuoi presentarti ai nostri lettori?

Sono nato a Mosca, in USSR, ed ho studiato matematica all'Università di Stato di Mosca. Ma non sono mai diventato un matematico. Non riuscivo a entusiasmarmi veramente per i numeri di Tamagawa, i gruppi di Coxeter e le altre cose in cui mi sarei dovuto specializzare. Il pensiero di Hardy, la cui matematica non sarebbe mai stata applicata, non faceva per me. Volevo fare qualcosa di più reale. Sono stato fortunato, tuttavia, perché ho visto alcuni grandi matematici al lavoro e sono diventato totalmente immune a quello pseudo rigore matematico che sfortunatamente è tanto di moda nell'Informatica.

Così approdai alla programmazione. Nel 1972 divenni membro di un gruppo che stava sviluppando un nuovo minicomputer per il controllo delle grandi centrali idroelettriche. Partecipai a tutte le fasi del progetto, dal disegno architetturale al testing dell'hardware, dal sistema operativo (la mia prima pubblicazione fu sui sistemi operativi real time) fino ai tool di programmazione. Mi feci un'esperienza sull'affidabilità del software, non è facile fare il reboot di una centrale elettrica, e sull'efficienza, l'acqua scorre in tempo reale.

A quel tempo scoprii i libri di due grandi scienziati dell'informatica dai quali ho imparato i fondamenti scientifici del mio lavoro: Donald Knuth e Dijkstra. Knuth mi ha insegnato le risposte. Dijkstra mi ha insegnato le domande.

Più volte nel corso del tempo sono ritornato sulle loro opere per nuove ispirazioni. Un'altra tappa importante della mia carriera fu alla Computer Science Branch of General Electric Research Center in Schenectady, NY. Lavorai su un linguaggio di alto livello chiamato Tecton e lessi molto: da una pletora di articoli sul progetto di un linguaggio di programmazione alla *Logica Summa* di Guglielmo da Occam - Aristotele e i logici medioevali sapevano molte cose sui differenti tipi di strutture che compaiono nei linguaggi naturali e le loro proprietà formali. A quel tempo iniziai una fruttuosa collaborazione con David Musser che dura ancora adesso.

Nel 1984 diventai assistente alla Polytechnic University di Brooklyn, NY. Insegnare informatica mi fece un gran bene, imparai a insegnare a ogni genere di corsi universitari, e nel farlo imparai molte cose. Sviluppai inoltre un'ampia libreria di algoritmi e strutture dati in Scheme. Questo lavoro portò allo sviluppo (insieme a David Musser) di una Libreria Generica per Ada.

Dopo una breve sosta ai laboratori Bell, dove lavorai su una libreria di algoritmi in C++, mi spostai ai laboratori HP di Palo Alto, CA (1988). Passai i 4 anni seguenti a lavorare sui sistemi di storage: dovetti imparare a programmare i controllori disco. Nel 1993 mi venne data una breve opportunità di tornare alla mie ricerche sulla programmazione generica. STL è un risultato di questo lavoro. Nel 1995 passai alla Silicon Graphics, dove sto cercando di formare un gruppo per lavorare su ulteriori sviluppi di STL.

Uhm ... Occam diceva "Entia non sunt multiplicanda". Gli oggetti (virtuali) non sono necessari. Più avanti affermi che ti senti un po' francescano, e Occam era francescano. Sembra che tu abbia applicato il rasoio di Occam all'OOP. Partire dagli algoritmi piuttosto che dagli oggetti mi ricorda un po'

la disputa medioevale sugli universali.

L'analogia è simpatica, ma non credo sia giusta. Non ho mai pensato all'OO in relazione alla filosofia nominalista e io non sono per niente un nominalista. In effetti la scuola francescana, Alessandro di Hales, Bonaventura e Duns Scoto, erano molto più vicini alla tradizione di S.Agostino e di Platone. Comunque Occam era un tipo strano. Come diceva l'editore della sua opera omnia, Gideon Gal: "Quello è proprio matto!".

[Quest'ultima battuta mi ha confermato l'idea che Stepanov si fosse lasciato veramente ispirare da Guglielmo da Occam, ma non sapevo fin dove arrivava il suo humor e non ho osato dirglielo]

Per la maggior parte dei lettori italiani, "Stepanov" va in coppia con "STL". STL significa "Standard Template Library" o "Stepanov & Lee"?

Beh, significa davvero Standard Template Library. Ho fatto un gioco di parole nella mia intervista con Dr. Dobb's Journal dicendo che STL stava per "Stepanov e Lee", ma era solo uno scherzo.

Quale è stato il ruolo di D. Musser e di A. Koenig nella storia di STL?

Ho collaborato con Dave Musser per quasi 20 anni. La nostra collaborazione è stata così stretta che è difficile dire chi ha contribuito a cosa.

Egli non è nella lista degli autori ufficiali di STL solo perché nel ridotto lasso di tempo in cui scrivemmo la proposta per lo standard, era impegnato in altri lavori.

Andy Koenig mi ha spiegato la struttura della macchina astratta del C. STL, in un certo senso, è l'applicazione delle tecniche di programmazione generica che Dave Musser e io avevamo sviluppato a un modello di macchina C. Se non fosse stato per Andy, sarei ancora lì a scervellarmi con gli oggetti allocati nell'heap e a rimpiangere la garbage collection.

Meng Lee è stata una perfetta collaboratrice nello stadio in cui fu necessario passare dalle nostre bellissime idee a una implementazione completa. Lei non mi ha fatto distogliere l'attenzione dal problema, tendo a perdere interesse sulle cose quando il problema è risolto, se conosco la soluzione, che cosa mi importa farla conoscere al resto del mondo?

È stata lei che ha speso ore estenuanti sul codice e sulla documentazione. In un certo senso, è stata l'unica persona a credere che da tutta questa roba potesse venire fuori qualcosa di utile; credo che a quel punto sia Dave Musser che io avessimo perso la speranza di potere spiegare a qualcuno su che cosa avevamo lavorato per tutto quel tempo.

Quale è l'origine di STL? È stata concepita per quello che è ora, "la" Libreria Standard del C++, o viene da qualche altro progetto? Puoi raccontarci la storia di STL?

Nel 1976, ancora in USSR, fui colpito da un grave caso di avvelenamento alimentare per avere mangiato pesce crudo. Mentre ero in stato di delirio all'ospedale, realizzai di colpo che la possibilità di addizionare numeri in parallelo dipende dal fatto che l'addizione è associativa (così, per farla breve, STL è il risultato di un'infezione batterica). In altre parole, capii che un algoritmo di riduzione parallela è associato con il tipo di struttura di un semigruppo.

Questo è il punto fondamentale: gli algoritmi sono definiti su strutture algebriche. Mi occorsero ancora un paio di anni per poter estendere la nozione di struttura aggiungendo i requisiti di complessità agli assiomi regolari. Ci vollero poi 15 anni per farlo funzionare (non sono affatto sicuro di avere avuto successo nel diffondere il concetto al di fuori del ristretto gruppo dei miei amici). Credo che la teoria degli iteratori sia centrale per l'Informatica come la teoria degli anelli o spazi di Banach è centrale per la Matematica.

Ogni volta che guardo un algoritmo cerco di trovare la struttura sulla quale è definito. Così quello che volevo era descrivere gli algoritmi in modo generico. Questo è quello che mi piace fare. Sono capace di spendere un mese a lavorare su un algoritmo ben noto cercando di trovare la sua rappresentazione generica. Finora, ho avuto stranamente poco successo nello spiegare alla gente quanto questa sia una cosa importante. Ma, in qualche modo, il risultato dell'attività, STL, ha avuto veramente successo.

Pensavo che i requisiti di complessità di STL fossero solo dei vincoli sull'efficienza dell'implementazione. Tu sembri voler dire che la complesità è pari a un vero requisito funzionale. È così?

Si seleziona un algoritmo a seconda della complessità delle operazioni fondamentali offerte da una struttura dati. Dischi e nastri magnetici sono funzionalmente equivalenti all'interfaccia SCSI, ma nessun progettista userebbe il nastro come se fosse un disco. In termini di STL: puoi sempre implementare p+n sugli iteratori forward, ma ciò non vuol dire che li puoi liberamente scambiare con iteratori random.

Cosa è la Programmazione Generica? Ho trovato alcuni articoli di A.Koenig sul JOOP, ma la "programmazione generica" è completamente assente dalla maggior parte dei libri di programmazione, inclusi Coplien, Meyer, Stroustrup, Lippman e così via. Si può definire STL come "Programmare in C++ come non l'avreste mai creduto possibile"?

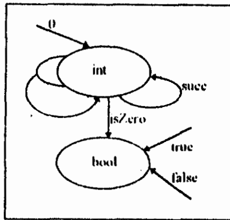
STL, almeno per me, rappresenta l'unico modo possibile di programmare. È, in effetti, assai differente dal modo comune in cui era ed è ancora presentata la programmazione in C++ nella maggior parte dei libri di testo. Ma, vedi, non stavo cercando di programmare in C++, stavo cercando il modo giusto di scrivere il software. Stavo cercando un linguaggio in cui poter esprimere ciò che volevo dire da tanto tempo. In altre parole, sapendo cosa voglio dire, lo posso dire in C++, lo posso dire in Ada, o Scheme. Mi adatto al linguaggio, ma l'essenza di ciò che voglio dire è indipendente dal linguaggio. Finora, il C++ è il migliore linguaggio che ho

Algebre Multisorted

di Carlo Pescio

Alexander Stepanov usa con una certa disinvoltura termini come algebra multisorted, teorie, categorie, ecc, che potrebbero risultare poco familiari ad alcuni lettori. Abbiamo quindi pensato di dedicare un piccolo spazio per tentare di chiarire, almeno in minima parte, di cosa si tratti. Ipotizziamo di voler *definire* i numeri naturali: intuitivamente potremmo dire che zero è un numero naturale, e che ogni numero che "segue" zero è un naturale. Come possiamo rendere questa definizione formale, ma allo stesso tempo indipendente da una rappresentazione concreta dei numeri naturali? Lo stesso problema, naturalmente, si pone per qualunque tipo di dato: una stringa, un albero binario, uno stack. Di norma questi vengono definiti facendo riferimento ad una rappresentazione del tipo, quando non si passi direttamente ad una implementazione. In informatica teorica questi vengono chiamati tipi di dato concreti (anche se molte volte, gli autori parlano di tipi di dato astratti anche quando fanno riferimento ad una rappresentazione). Esiste però la possibilità di definire dei veri e propri tipi di dato astratti, come modelli di (famiglie di) algebre su una segnatura. È necessario introdurre alcune definizioni:

FIGURA 1
Esempio di segnatura come multigrafo



Dato un insieme S di simboli (che chiamiamo *sort* o *tipi*) una segnatura Σ su S è una famiglia di insiemi $\Sigma=\{\Sigma_{w,s}\}_{w\in S^*, s\in S}$ dove ciascun $\Sigma_{w,s}$ è un insieme di simboli detti *operatori*.

Una segnatura si può rappresentare come un multigrafo orientato (**Figura 1**), dove i nodi corrispondono alle sort ed i multiarchi agli operatori. Ad esempio, il multigrafo in **Figura 1** rappresenta la segnatura su S={int, bool} dove Σ è definito da:

$\Sigma_{\Lambda,int}=\{0\}$, $\Sigma_{int,int}=\{succ\}$, $\Sigma_{intint,int}=\{+\}$, $\Sigma_{int,bool}=\{isZero\}$, $\Sigma_{\Lambda,bool}=\{true,false\}$ ed ogni altro $\Sigma_{w,s}$ è vuoto.

Possiamo pensare alla segnatura come alla sintassi; occorre ora definire il significato (semantica) degli operatori. Data una segnatura, possiamo naturalmente assegnare molti diversi significati ai suoi operatori; ciò si ottiene definendo un'algebra sulla segnatura, o Σ-algebra. Se la segnatura ha più di una sort (come in **Figura 1**), l'algebra sarà multisorted. Data una segnatura Σ, una Σ-algebra A è una coppia $A=(\{A_s\}_{s\in S}, \{\sigma_A\}_{s\in S})$, dove per ogni sort s, A_s è detto *carrier* della sort s, e per ogni $\sigma \in \Sigma_{s1...sn,s}$ σ_A è una funzione totale da $A_{s1}\times...\times A_{sn} \rightarrow A_s$ ed è detta interpretazione di σ. Informalmente, il carrier rappresenta i valori (semantici) associati ai diversi elementi sintattici, mentre le interpretazioni delle operazioni rappresentano, come ovvio, il significato dei diversi operatori introdotti con la segnatura. È immediato vedere che ogni algebra su una segnatura è anche un'algebra su una segnatura isomorfa alla prima, dove in pratica ogni sort ed operatore venga "sostituito" da uno equivalente con altro nome. Quindi l'algebra definità è in pratica indipendente dalla rappresentazione sintattica dei termini. Esiste un'algebra particolarmente interessante, che è la cosiddetta *algebra dei termini ground*; pur mancando lo spazio per una definizione formale, possiamo pensare che in quel caso la semantica equivalga alla sintassi, quindi il significato del simbolo '0' è il simbolo '0', il significato di succ(0) è il simbolo 'succ(0)', e così via. Partendo da queste basi, possiamo introdurre un *modello* dei tipi di dato, che deve rispettare degli *assiomi* all'interno di un sistema deduttivo. L'importanza di questi concetti, a prima vista un po' ostici, sta proprio nella possibilità di definire formalmente la semantica dei tipi di dato astratti, anziché limitarsi ad una descrizione in linguaggio naturale o basarsi su una rappresentazione dei tipi. Esiste un apparato formale molto ampio, che permette di arrivare a dimostrazioni formali circa le proprietà dei tipi di dato definiti come algebre su una segnatura: anche in questo caso, l'alternativa comunemente usata nei testi di strutture dati ed algoritmi è un ragionamento semi-formale basato su una rappresentazione del tipo. Riprendendo l'esempio di cui sopra, potremmo ad esempio dimostrare che in un modello dove l'operatore + rispetta gli assiomi:

+(0, y) = y
+(succ(x), y) = +(x, succ(y))

l'operatore ha la proprietà associativa e commutativa. Esiste infine la possibilità di rendere *eseguibili* le specifiche, usando linguaggi basati su sistemi di riscrittura come Obj3, in modo da poter prototipare più rapidamente modelli alternativi.

Carlo Pescio svolge attività di consulenza in ambito internazionale sulle metodologie di sviluppo Object Oriented e di programmazione in ambiente Windows. È laureato in Scienze dell'Informazione, membro dell'Institute of Electrical and Electronics Engineers Computer Society, dei Comitati Tecnici IEEE su Linguaggi Formali, Software Engineering e Medicina Computazionale, dell'ACM e dell'Academy of Sciences di New York. Può essere contattato tramite la redazione o su Internet come pescio@programmers.net

scoperto per dire ciò che voglio dire. Non è ideale, ma posso fare di più in C++ che in qualunque altro linguaggio io abbia provato. È mia speranza che un giorno ci sarà un linguaggio progettato specificamente con la programmazione generica.

Puoi spiegare a un comune programmatore C++ cosa è la programmazione generica e in che relazione sta con il C++ e con STL?

La programmazione generica è un modo di programmare basato sul cercare la rappresentazione più astratta possibile di algoritmi efficienti. Cioè, si parte da un algoritmo e si cerca l'insieme più generale di requisiti che gli permettono di lavorare in modo efficiente. La cosa interessante è che molti algoritmi diversi richiedono lo stesso insieme di requisiti e ci sono molte implementazioni di questi requisiti. L'analogo in matematica è che molti teoremi differenti dipendono dallo stesso insieme di assiomi e che ci sono diversi modelli di questi assiomi.
Le astrazioni funzionano!

La programmazione generica assume che ci siano alcune leggi fondamentali che governano il comportamento delle componenti software e che è possibile progettare moduli interoperabili basandosi su queste leggi. È anche possibile usare le leggi per guidare il disegno del software.

STL è un esempio di programmazione generica. C++ è un linguaggio nel quale sono stato capace di produrre un esempio convincente.

Credo che STL segni una netta dipartita dal comune stile di programmazione C++, che a sua volta credo sia derivato fondamentalmente da SmallTalk. Sei d'accordo?

Sì. STL non è object oriented. Io credo che la "object orientation" sia un imbroglio quasi quanto l'Intelligenza Artificiale. Devo ancora vedere un pezzo di codice interessante che venga da quella gente lì. A dire il vero, sono stato ingiusto con l'AI: ho imparato molte cose dal lavoro del Laboratorio di AI del MIT, hanno fatto cose veramente fondamentali;

Hakmem [MIT AI Memo 239, February 1972] di Bill Gosper è uno dei libri più interessanti che un programmatore possa leggere. L'AI magari non aveva un fondamento serio, ma ha prodotto Gosper e Stallman, Moses e Stallman [Stallman: il principale autore di Emacs, il ben noto editor programmabile in Lisp; Moses: autore di Macsyma, il primo sistema matematico simbolico completo, scritto in Lisp; Stallman: autore, insieme a Guy Steele, di Scheme: un dialetto del Lisp]. Trovo che l'OOP sia tecnicamente scorretta. Cerca di decomporre il mondo in termini di interfacce che variano su di un singolo tipo. Per affrontare i problemi reali servono le algebre multisorted - famiglie di interfacce su tipi multipli. Trovo che l'OOP sia filosoficamente scorretta. Dice che tutto è un oggetto. Anche se fosse vero non è molto interessante: dire che tutto è un oggetto è come non dire niente.

Trovo che l'OOP sia metodologicamente sbagliata. Inizia con le classi, è come se un matematico iniziasse con gli assiomi. Non si inizia con gli assiomi: si inizia con le prove. Solo quando si sono scoperte un po' di prove correlate, si arriva agli assiomi. Lo stesso nella programmazione: devi cominciare con degli algoritmi interessanti. Solo quando li capisci bene, puoi ricavare un'interfaccia che li faccia funzionare.

Posso riassumere il tuo pensiero dicendo "cerca le strutture dati [generiche] dentro a un algoritmo" invece di "cerca gli algoritmi [virtuali] dentro a un oggetto"?

Sì, inizia sempre dagli algoritmi.

Questo è un cambiamento di paradigma radicale rispetto sia alla programmazione imperativa che a quella OO. Quali sono i benefici, e i difetti, di questo paradigma confrontato con l'OOP tradizionale, come in Smalltalk o in Java?

Il mio approccio funziona, il loro no. Cerca di implementare a oggetti qualcosa di semplice, come, diciamo, max. Non so come lo si possa fare. Usando la programmazione generica posso scrivere:

```
template <class StrictWeakOrdered>
inline StrictWeakOrdered&
max(StrictWeakOrdered& x,StrictWeakOrdered& y) {
  return x < y ? y : x;
}
```

e:

```
template <class StrictWeakOrdered>
inline const StrictWeakOrdered& max
(const StrictWeakOrdered& x,
     const StrictWeakOrdered& y)
{
  return x < y ? y : x;
}
```

(Servono sia il & che il const).

Cerca di farlo in Java. Non si può scrivere in Java una funzione generica max() che prende due argomenti dello stesso tipo e ha un tipo di ritorno dello stesso tipo. L'inheritance e le interfacce non aiutano. E se non possono implementare max o swap o una ricerca lineare, che possibilità hanno di implementare qualcosa di veramente complesso? Questa è la mia cartina di tornasole: se un linguaggio mi lascia implementare max e swap e le ricerche lineari in modo generico, allora ha qualche potenzialità.

Java è un linguaggio completamente nuovo, però manca dei template, perciò impedisce l'uso della programmazione generica. Ogni cosa deve essere una classe. Cosa pensi di Java?

Ho passato alcuni mesi a programmare in Java. Contrariamente alle promesse dei suoi autori, non mi ha ispirato. Per la prima volta nella mia vita, programmare in un nuovo linguaggio non mi ha dato nuove idee. Mantiene tutta quella roba che in C++ non uso mai, ereditarietà, funzioni virtuali, astrusità OO e toglie tutto quello che trovo utile. Può darsi che abbia successo - dopo tutto, anche il DOS ebbe successo - e può essere un affare per tutti voi lettori imparare Java, ma non ha nessun valore intellettuale. Guarda la loro implementazione delle hash table. Guarda alla loro routine di sort che si trova nello loro *cool* sorting applet. Cerca di usare AWT. Il miglior modo di giudicare un linguaggio è di guardare il codice scritto da chi lo propone. "Radix enim omnium malorum est cupiditas"- e Java è chiaramente un esempio di Money Oriented Programming (MOP).

Come il principale propositore di Java in SGI mi disse: "Alex, devi andare dove sta il denaro". Ma io non voglio particolarmente andare dove sta il denaro, di solito non profuma bene da quelle parti.

Pensi che la programmazione per template e la programmazione generica saranno adottati dalla maggioranza dei programmatori C++, o che resteranno confinati a STL, un po' come i manipolatori, che non sono mai stati usati fuori dalla libreria iostream?

Non so. Ci vorrà molto tempo prima che le idee dietro STL entrino nel mainstream. Lo sapremo fra 10-15 anni se qualcosa sarà venuto fuori da tutto ciò.

Una cosa che mi ha stupito è quanto rapidamente STL sia stata adottata dal comitato di standardizzazione C++. Voglio dire, i comitati di standardizzazione sono ben noti per esser cauti e conservativi. Come si spiega?

Il supporto di Bjarne Stroustrup è stato fondamentale. Bjarne voleva davvero STL nello standard e quando Bjarne vuole qualcosa, lo ottiene. È testardo come un mulo. Mi ha persino convinto a fare dei cambiamenti in STL che io non avrei fatto per nessun altro, anch'io sono testardo, ma è la persona più determinata che io conosca. Ottiene quello che vuole. Gli ci è voluto un po' per capire cosa fosse STL, ma quando ci è arrivato, ha saputo farla passare.

Ha contribuito a STL anche sostenendo l'idea che esiste più di un modo valido di programmare, nonostante critiche e propagande senza fine da più di una decade, per arrivare a quella combinazione di flessibilità, efficienza, overloading e sicurezza dei tipi nei template che hanno reso STL

possibile. Voglio dire chiaramente che Bjarne è il più eminente progettista di linguaggi della mia generazione.

STL è piena di usi "creativi" dei template, come esportare tipi simbolici dalle classi, o il pattern matching di un set di algoritmi in overload sui tag di iteratori. Di sicuro nessun libro di C++ insegna queste cose. Come sei arrivato a questi idiomi di C++?

Sapevo esattamente cosa cercavo di ottenere. Così ho giocato con il linguaggio fino a quando non ci sono riuscito. Ma mi ci sono voluti molti anni per scoprire tutte le tecniche. E molte volte ho seguito piste false. Per esempio, ho sprecato anni cercando un uso per l'ereditarietà e le funzioni virtuali. Prima di capire che sono meccanismi fondamentalmente scorretti e che non dovrebbero esser usati. Sono felice che nessuno possa vedere i passi intermedi - la maggior parte erano veramente sciocchi. Mi ci sono voluti anni per uscire con qualcosa di appena decente. Mi ha anche aiutato il fatto che Bjarne abbia voluto mettere alcune nuove feature nel linguaggio giusto per supportare i miei idiomi. L'ha chiamato una volta "just in time language design".

Quale pensi sia il modo migliore di insegnare la programmazione generica e STL ai programmatori C++? Pensi che l'ereditarietà e altre tecniche OO dovrebbero essere insegnate prima, dopo o insieme a STL?

Ho in mente di fare un corso sulla programmazione generica in SGI. Spero che porterà a un libro, ma sono notoriamente uno scrittore pigro, non finisco mai un articolo a meno che abbia un collaboratore che lo faccia.

Una ricerca su Lycos sul tuo nome mi ha dato solo due titoli: il manuale di STL e un rapporto sulla tua presentazione di STL al comitato di standardizzazione.

Beh, sono pigro, ma non così pigro. Ho pubblicato circa 20 articoli e un libro. Molti sono su differenti siti STL (il sito di Dave Musser probabilmente ne ha parecchi).

Quale libro?

Il libro si chiama ""The Ada Generic Library: Linear List Processing Packages", di David R. Musser e Alexander A. Stepanov, Compass Series, Springer-Verlag, 1989. Ma non vale proprio la pena di leggerlo.

STL spinge all'estremo i compilatori C++. I compilatori odierni sono ancora incapaci di compilare alcune parti di STL. Come sei riuscito a sviluppare e testare STL?

Mi sono venuti i capelli grigi a tentare di compilare STL. La realtà sfortunatamente è che un mucchio di codice nell'implementazione attuale di STL non è ottimizzato a causa di limitazioni e di errori dei compilatori che ho usato nello sviluppo di STL.

Fortunatamente, Bjarne mi ha aiutato a capire cosa certe feature non ancora implementate avrebbero dovuto fare. È di notevole aiuto poter chiedere al progettista di un linguaggio cosa un dato costrutto si suppone debba fare.

Da dove arrivano gli allocatori di STL?

Ho inventato gli allocatori per maneggiare l'architettura di memoria Intel. In teoria non sono poi un'idea così brutta, uno strato che incapsula tutta la gestione della memoria: puntatori, reference, ptrdiff_t, size_t. Sfortunatamente in pratica non funziona. Per esempio:

```
vector<int, alloc1> a(...);
vector<int, alloc2> b(...);
```

ora non si può scrivere:

```
find(a.begin(), a.end(), b[1]);
```

b[1] restituisce un alloc2::reference e non int&. Potrebbe essere un mismatch di tipo. Bisognerebbe cambiare il modo in cui il nucleo (*core*) del linguaggio tratta i reference per rendere gli allocatori davvero utili.

Questo non è un serio problema dei template in C++? Posso variare il comportamento di una classe usando parametri di template, ma le diverse specializzazioni non sono compatibili come tipo. Al contrario, le sottoclassi (in senso OO) di una classe sono tutte sottotipi del tipo base.

Credo che il problema sia più profondo. T* è cablato nel linguaggio. In generale, credo che sarebbe necessario scrivere un linguaggio fin dalle fondamenta per poter programmare in modo generico in modo consistente. Come vorrei che qualcuno mi assumesse per fare proprio questo!

Ho trovato due implementazioni di hash table nel sito di D. Musser, ed entrambe funzionano e anche bene, molto meglio delle hash table che si trovano comunemente nelle librerie di classi. Perché non sono state incluse in STL?

Politica. Avrebbero dovuto esserci. La nostra nuova implementazione di STL le contiene. In generale, ci occorre un meccanismo per aggiungere cose a STL. Dopo tutto, STL è un framework estensibile, implora di essere esteso.

Ci sono molte strutture dati mancanti, per esempio, liste singolarmente linkate, matrici e grafi. SGI vuol guidare l'estensione a STL.

Stai ancora lavorando su STL? Facendo che cosa?

Il mio gruppo in SGI, Matt Austern, Hans Boehm e io, abbiamo appena finito una nuova versione di STL. Include i contenitori hash, l'allocazione della memoria thread safe e una spettacolare documentazione Web. SGI l'ha rilasciata in pubblico dominio (http://www.sgi.com/Technology/STL). Speriamo di potere continuare a farla crescere. Fra le cose che vorremmo aggiungere ci sono strutture dati multidimensionali, persistenza e multithreading. Ma non è chiaro quanto a lungo potremo continuare a fare queste cose. Non c'è supporto da parte del mio management per alcuna attività su STL/programmazione generica. È molto difficile spiegare loro perché SGI dovrebbe pagare per delle estensioni. Fu altrettanto arduo spiegarlo al mana-

gement di Hewlett Packard. Tanto che cancellarono il mio progetto 5 mesi dopo che STL era stata approvata nello standard.

STL sembra avere anche qualche difetto. Per prima cosa, l'STL di HP non è thread safe. In secondo luogo, i template obbligano a mettere un mucchio di codice negli header file. Non è possibile costruire vere librerie di template: STL è quasi completamente una libreria risolta in compilazione. L'ultimo problema è che quasi nessun CASE tool e nessuna metodologia di OOD supporta la programmazione generica. Per esempio, nessun CASE tool permette di definire funzioni generiche. E solo la notazione di Booch è in qualche modo capace di esprimere i template, ma in modo non molto intuitivo (questa, almeno, è la mia impressione).

STL di SGI è thread safe.

L'accettazione del modello separato di compilazione nello standard, progettato dai miei colleghi in SGI John Wilkinson, Jim Dehnert and Matt Austern risolve il tuo secondo problema. È stato votato nello standard, ma ci vorrà qualche tempo prima che i compilatori possano fare compilazioni separate. Credo che la compilazione separata porterà alla fine ad avere librerie separate, e persino shared library o DLL, di template. Questo è il principale motivo per cui ho iniziato il lavoro in SGI sulla compilazione separata.

Non è poi così difficile scrivere tool che gestiscano la programmazione generica. E sono sicuro che se c'è qualche prospettiva di mercato per questo, Grady Booch cambierà la sua notazione per trattare la programmazione generica.

Una cosa penosa per un programmatore C++ è che i comitati di standardizzazione sembrano ignorarsi gli uni gli altri. OMG ha appena definito uno standard per la programmazione distribuita (CORBA), ma il mapping di CORBA sul C++ non riconosce STL. Hanno definito il loro set di classi, come Sequence<T> e CORBA::String. Lo stesso problema ce l'hanno ODMG e il suo standard ODMG-93 per i database a oggetti. Perché succede? Cambierà qualcosa?

Sono abbastanza vecchio da ricordare tutti gli standard di networking emessi negli anni '70. Chi se li ricorda più ora?

Cosa diventa la programmazione generica in ambiente distribuito? La programmazione generica è basata sull'idea che il compilatore "conosce" tutti i tipi possibili al tempo di compilazione. Questo non è realistico in un ambiente distribuito. Dobbiamo pensare a una specie di ORB integrato con il compilatore? Oppure la programmazione generica non è adatta alla programmazione distribuita, in tal caso, Java sarebbe meglio?

La programmazione generica non ha nulla a che vedere con run-time vs. compile-time. Il problema che trovo nell'OOP è che non permette di scrivere nemmeno gli algoritmi più semplici. Di nuovo, la signature di max è:

```
max: T x T -> T
```

Questo non è esprimibile in Java, perché l'ereditarietà da qualche classe o interfaccia T la cambia così:

```
max: T' x T -> T
```

Occorre la trasformazione covariante della signature e la possibilità di ottenere tipi da altri tipi, una specie di tipi virtuali, una v-table contenente descrittori di tipi [questo è quanto si ottiene in C++ con l'idioma dei typedef embedded in classi, N.d.T.].

Cosa è un iteratore?

Un iteratore è l'unione di due teorie. La prima teoria è la teoria del nome (handle, cookie, address). Un nome è qualcosa che punta a qualcos'altro. Noi la chiamiamo Trivial Iterator nel nostro sito Web.

Oltre al dereferenziamento, ha definito l'uguaglianza secondo il seguente assioma:

```
i == j iff &*i == &*j
```

Cioè, due iteratori sono uguali se e solo se puntano allo stesso oggetto. L'uguaglianza, naturalmente, deve soddisfare tutti gli assiomi standard.

La seconda teoria è quella del successore (++i) con i suoi raffinamenti: successore (i++), predecessore (--i e i--) e addizione (++, --, + e -) con gli assiomi standard.

E, naturalmente, un puntatore è un modello di iteratore random.

Alcuni sostengono che un puntatore è un modo di fare qualunque concepibile cosa orrenda e strana nella memoria. Java e Delphi hanno fatto appena bene a rimuovere i puntatori.

Non permettere l'aritmetica dei puntatori è una buona cosa. L'aritmetica dei puntatori dovrebbe essere permessa solo dove è realmente permessa in C e C++, cioè per i puntatori ad array. Non permettere i puntatori in generale è stata una vera stupidaggine. Non si può scrivere uno swap generico senza avere nel linguaggio puntatori o reference.

"Categoria" è un termine inflazionato nella comunità C++. Come chiamare altrimenti le categorie di iteratori?

Io spesso li chiamo concetti.

Ho cercato di sviluppare una lista singolarmente linkata conforme a STL. Ma non ho implementato la funzione *size()* perché non volevo che l'overhead tenesse un contatore solo per implementare size() in un tempo costante, secondo quanto richiesto dal C++ Committe Draft. Ho fatto bene?

size() una volta era di complessità lineare nel caso delle liste STL. La tua è stata una decisione giusta perché se un utilizzatore vuole tenere un contatore è facile farlo, ma di solito non serve e rende splice di complessità lineare (una volta era costante). Ma il comitato di standardizzazione ha insistito che la cambiassi in complessità costante. Non ho avuto scelta. Forse dovremo cambiare questo requisito.

Sto cercando di esprimere un albero in STL e ho trovato qualche problema: ogni nodo ha un padre, ma due figli. Spostarsi sul padre potrebbe essere rappresentato come --, ma mi occorrerebbero due diversi operatori ++ per spostarmi sui figli. Come si può affrontare con gli iteratori strutture non lineari, come alberi o grafi?

Anche su di una sequenza ci sono diversi iteratori. Gli iteratori inversi sono un esempio. Gli iteratori stride sono molto importanti e avrebbero dovuto essere messi in STL.

Confesso la mia ignoranza. Cosa è un iteratore stride?

Andare da i a i+5 o i+10.

Che differenza c'è con gli iteratori random?

Un iteratore stride è un adattatore di iteratore che prende un iteratore random e fornisce un (iteratore random) tale che ++ causa un salto (*stride*): una sequenza di iteratori a intervalli di n passi.

Come si rapporta al problema di attraversare un albero?

Non volevo dire che gli iteratori stride o inversi abbiano niente a che fare con l'attraversamento degli alberi, ma ci potrebbero essere tipi multipli di iteratori su una struttura dati per diversi ordini di iterazioni: nel caso degli alberi attraversamenti in pre, o in post ordine.

Un mio frequente dilemma è: dovrei fare di questa funzione una funzione membro o una funzione globale? Quale è stato il razionale di questa decisione in STL?

Falla globale ogni volta che puoi. Sarebbe molto meglio se anche begin() e end() fossero globali - così le si potrebbe definire per array C. Sarebbe molto più elegante se l'operatore * fosse globale con questa definizione generica:

```
template <class T>
T& operator*(T& x) { return x;}

template <class T>
const T& operator*(const T& x) { return x;}
```

Così potremmo scrivere:

```
copy(0, 25, ostream_iterator<int>("\n"));
```

In generale, per oggetti non-iteratori l'operatore * dovrebbe restituire l'oggetto stesso, il "significato" di una cosa senza nome è la cosa stessa. Mi piacerebbe poter scrivere anche i costruttori e i distruttori come funzioni globali. Si potrebbero fare cose incredibili se il linguaggio lo permettesse.

Supponiamo che io definisca l'operatore * come nel tuo esempio, e poi definisca nel mio programma:

```
template <class T> class SmartPtr {
  T* ptr;
public:
  SmartPtr(T* _ptr = 0) : ptr(_ptr){}
  T* operator*() const {return ptr;}
  // ...
};
```

a questo punto il codice dovrebbe essere ambiguo:

```
int i=0;
SmartPtr<int> sp(&i);
int j= *i; // apply SmartPtr<int>::operator* or
// operator<SmartPtr<int>>? both are user
defined
```

No. La tua seconda definizione fa miglior match (l'unificazione è più profonda) di quella globale. Questo è il senso della specializzazione parziale [la specializzazione parziale è stata una delle ultime innovazioni introdotte nel C++].

Ora, qualche curiosità: perché STL non ha un adattatore di tipo "sorted container"?

I set sono sorted container.

Ma non sono adattatori. Perché è possibile scrivere uno heap, ma non un sorted container, da qualunque contenitore con iteratori random?

Ricordiamoci che è stato molto difficile fare accettare STL a causa delle sue dimensioni. Ho dovuto buttare via dozzine di componenti utili: pensa a cosa è accaduto alle hash table.

Perché la funzione adjust_heap di <heap> non è documentata? È indispensabile per usare gli heap nell'algoritmo di Dijkstra.

Ho fatto molta fatica a fare accettare le funzioni heap nello standard. In origine, avrei voluto tutte le funzioni ausiliarie visibili in STL, ma non fu politicamente possibile.

Mi è difficile capire cosa c'entri la politica con una funzione sullo heap.

Non è questione di quella particolare funzione, ma del loro numero totale. Bjarne è personalmente responsabile per aver dimezzato il numero di componenti in STL. Cercava di renderlo il più piccolo possibile per placare le opposizioni.

Sei mai stato in Italia?

Sì, ho passato 10 giorni a Pisa, ho visitato Firenze e Lucca. Sogno di andare ad Assisi - in fondo al cuore sono un francescano. Piango durante il secondo atto della Tosca e il terzo atto della Traviata. Tengo Dante (in italiano!) sul mio comodino. Mi piace la *pasta*, il *prosciutto* e il *Chianti* [in it. nel testo]; anche se sono un tradizionalista, mi sento di più come Giovanni XXII che come Pio XII.

Devi sapere l'italiano molto bene se riesci a leggere Dante in originale: ben pochi italiani ci riuscirebbero!

Il mio italiano è molto scarso. Il modo in cui leggo Dante in italiano è con una traduzione in inglese a fronte. Leggo una *stanza* ad alta voce in italiano e poi leggo la traduzione.

Grazie infinite, Alex.

Ti auguro di potere venire presto in Italia e di visitare Assisi, leggendo Dante nella città vecchia di Assisi: la più bella cittadella medievale in Italia.

Graziano Lo Russo è laureato in Ingegneria Elettronica; è uno specialista in analisi e sviluppo Object Oriented. Può essere contattato tramite e-mail all'indirizzo: lorusso@programmers.net