DENTRO LA SCATOLA

Rubrica a cura di

Fabio A. Schreiber

Dopo aver affrontato negli scorsi anni due argomenti fondanti dell'Informatica – il modo di codificare l'informazione digitale e la concreta possibilità di risolvere problemi mediante gli elaboratori elettronici – con questa terza serie andiamo ad esplorare "*Come parlano i calcolatori*". La teoria dei linguaggi e la creazione di linguaggi di programmazione hanno accompagnato di pari passo l'evolversi delle architetture di calcolo e di gestione dei dati, permettendo lo sviluppo di applicazioni sempre più complesse, svincolando il programmatore dall'architettura dei sistemi e consentendogli quindi di concentrarsi sull'essenza del problema da risolvere.
Lo sviluppo dell'Informatica distribuita ha comportato la nascita, accanto ai linguaggi per l'interazione tra programmatore e calcolatore, anche di linguaggi per far parlare i calcolatori tra di loro – i protocolli di comunicazione. Inoltre, la necessità di garantire la sicurezza e la privatezza delle comunicazioni, ha spinto allo sviluppo di tecniche per "non farsi capire" da terzi, di quì l'applicazione diffusa della crittografia.
Di questo e di altro parleranno le monografie quest'anno, come sempre affidate alla penna (dovrei dire tastiera!) di autori che uniscono una grande autorevolezza scientifica e professionale ad una notevole capacità divulgativa.

La compilazione: concetti e sviluppi tecnologici

Stefano Crespi Reghizzi

1. La compilazione

Quali sono i programmi più eseguiti e meno conosciuti dall'utente? Con buona probabilità la risposta è: i compilatori. Introdotto all'alba dell'informatica, il termine "compilazione" sta per la traduzione d'un testo *sorgente*, scritto in linguaggio artificiale, in un testo *pozzo*[1], scritto in un altro linguaggio.
Cercheremo di descrivere i compilatori o traduttori nei loro aspetti concettuali, tecnici e applicativi, e nell'evoluzione concomitante a quella delle architetture di calcolo e della Rete.
I linguaggi progettati nei cinquant'anni passati sono migliaia, quelli tuttora utilizzati poche decine. I linguaggi, a seconda della destinazione, si classificano in: programmazione degli elaboratori (per esempio, C++), interrogazione di basi dati (per esempio, SQL), descrizione dei documenti (per esempio, XML), progetto dei sistemi digitali (per esempio, VHDL), controllo di robot ecc..
Poiché sono i linguaggi programmativi quelli che stanno al centro della ricerca sulle tecniche di compilazione, di cui beneficiano anche le altre categorie di linguaggi, non è limitativo parlare della compilazione dei soli linguaggi di programmazione.
Con un balzo indietro di 50 anni, immaginiamo la pena dei programmatori che scrivevano le applicazioni nel linguaggio assemblatore, scegliendo i codici operativi, i registri e gli indirizzi dei dati. All'ingrandirsi degli applicativi, le difficoltà crebbero, e, ad aggravare la situazione, per ogni nuovo elaboratore, il lavoro era completamente da rifare: mancava totalmente la *portabilità* dei programmi.
È storia nota l'invenzione del FORTRAN, il primo progetto industriale d'un linguaggio e del suo compilatore. Altrettanto nota, credo, è la straordinaria sinergia che si ebbe negli anni 1950-60 tra la linguistica, la nascente teoria degli automi e dei linguaggi formali, e l'ingegneria dei compilatori.
Il buon esito del compilatore FORTRAN della

1 La denominazione "testo pozzo" qui usata sembra più appropriata di quella tradizionale "testo oggetto": infatti il flusso della traduzione corre, come un corso d'acqua, dalla sorgente al pozzo.

IBM, fece fare un balzo alle applicazioni scientifiche, che poterono essere scritte in un linguaggio di alto livello, e mostrò la via a altri linguaggi importanti, come Algol 60, COBOL e Lisp.

2. Struttura del compilatore

La struttura di un compilatore degli anni 1960 è ancora attuale, al livello delle parti principali. Vi sono più passate, ognuna elabora un testo (o meglio una *rappresentazione intermedia* IR) e produce un'altro testo per la passata successiva. La prima passata legge il *testo sorgente*, scritto nel linguaggio da tradurre; l'ultima produce il *testo pozzo*, nel linguaggio della macchina.
Ogni passata, tranne l'ultima, produce dunque una IR, che è l'ingresso della passata successiva. La granularità delle IR cambia da una passata all'altra, dai costrutti del linguaggio sorgente fino alle istruzioni macchina.
In questo schema, l'ordinamento tipico delle passate è il seguente:

- il *parsificatore* (o *analizzatore lessicale-sintattico*) verifica che il testo sorgente sia lessicalmente e sintatticamente corretto, e lo trasforma in una IR, che agevola le passate successive. La IR tipica è l'*albero sintattico astratto*, AST, che mostra le inclusioni testuali tra i costrutti del programma sorgente. Gli elementi atomici della AST sono gli identificatori, le funzioni, gli operatori, le costanti ecc., ossia i *lessemi* del linguaggio.
- L'*analizzatore semantico* controlla sulla IR del programma che le regole semantiche del linguaggio sorgente siano rispettate: per esempio, le regole di conformità di tipo tra gli operandi delle espressioni aritmetiche.
- Il *generatore del codice* sceglie le istruzioni macchina.
- *Ottimizzatori*. Altre passate, poste a monte o a valle del generatore di codice, sono necessarie per migliorare l'efficienza del codice prodotto. Le ottimizzazioni sono la parte più complessa e qualificante del compilatore.

Tale schema viene denominato *traduttore guidato dalla sintassi* (SDT[2]), poiché la prima attività è l'analisi sintattica, che si basa sulla grammatica formale (o sintassi) del linguaggio sorgente. Per meglio comprendere, conviene guardare da vicino l'organizzazione dei primi due stadi.

2.1. Teorie formali nel progetto del compilatore

La decomposizione in passate permette di dominare la complessità del progetto e di sfruttare in ciascuna passata certi metodi specializzati, che ora saranno tratteggiati.
Dall'invenzione del FORTRAN, tutti i linguaggi sono stati definiti mediante una *grammatica* formale, cioè mediante delle regole sintattiche quali per esempio:

frase_condizionale → **if** espressione_booleana **then** istruzione **else** istruzione

Le regole sono dette libere dal contesto (*context-free o type 2*) dal linguista Noam Chomsky, e BNF (*Backus Naur Form*) dagli informatici del progetto Algol 60. I termini come frase_condizionale sono le classi sintattiche; la classe programma comprende tutti i testi sorgente che obbediscono alla grammatica.
Sotto la sintassi sta il lessico. Come in italiano un periodo è una serie di parole, così un programma è una sequenza di lessemi (costanti, identificatori, operatori, commenti, parole chiave come **if** ecc.).
La grammatica comprende al livello inferiore le regole lessicali, esse stesse scritte sotto forma di regole libere dal contesto o più spesso di espressioni regolari. Per esempio una costante intera è definita dall'espressione regolare 0 | (1 ... 9)(0 ... 9) *.
La prima operazione del parsificatore è l'analisi lessicale (*scanning*), che segmenta il testo sorgente nei lessemi. Essa opera tramite dei semplici automi a stati finiti che scandiscono il file sorgente e riconoscono le classi lessicali.
Successivamente il parsificatore esamina il programma, ora rappresentato come stringa di lessemi, per verificare le regole sintattiche. L'algoritmo è un automa che, oltre agli stati interni, possiede una memoria ausiliaria con modalità di accesso LIFO (ossia una pila).
Gli algoritmi lessicali e sintattici, studiati dagli anni 1960, sono efficienti: il tempo di calcolo è lineare rispetto alla lunghezza del programma, per gli algoritmi più diffusi, gli analizzatori ascendenti a spostamento e riduzione e quelli

[2] Syntax directed translator.

discendenti del tipo predittivo, che realizzano un automa deterministico. I due casi si differenziano rispetto all'ordine in cui l'albero sintattico è costruito: rispettivamente in ordine ascendente dai lessemi alla classe sintattica programma, o nell'ordine inverso, dalla radice alle foglie dell'albero, detto discendente.
In pratica il parsificatore deve anche costruire l'albero sintattico astratto AST del programma e dare indicazioni diagnostiche in caso di errore.
Certo l'esito della parsificazione è soltanto il primo esame, perché un programma, pur se sintatticamente corretto, può violare la semantica del linguaggio. Anche la distinzione tra correttezza sintattica e semantica, come quella tra lessico e sintassi, è un retaggio della prima teoria di Chomsky.
Più precisamente gli errori presenti in un programma sono classificabili come lessicali, sintattici e semantici. Un'altra classificazione ortogonale distingue gli errori *statici* da quelli *dinamici*, dove l'aggettivo "statico" indica un'attività svolta durante la traduzione, mentre "dinamico" si riferisce a quanto sarà fatto durante l'esecuzione del programma. Per esempio, l'assegnamento d'una costante del tipo stringa a una variabile del tipo integer è un errore semantico statico; l'uso di una variabile, come indice d'un vettore, la quale fuoriesca dalle dimensioni del vettore stesso, è un errore semantico, ma dinamico.
La seconda passata verifica la correttezza semantica statica del programma. Le regole di tipizzazione e di visibilità ("scope") sono quelle proprie del linguaggio sorgente. L'analizzatore o valutatore semantico è un algoritmo ricorsivo che visita lo AST, controllando, per ogni costrutto (per esempio, un assegnamento) l'osservanza delle regole semantiche.
A differenza di quelle sintattiche, le regole semantiche non sono generalmente coperte da un modello formale, anche se non sono mancati i tentativi di formalizzarle.
Nei compilatori dei linguaggi correnti, la semantica è talvolta specificata in qualche notazione semiformale, come le grammatiche con attributi (proposte da D. Knuth nel 1968).
Superato il controllo semantico, il testo è tradotto in una IR, che quasi sempre include la tabella dei simboli, concettualmente simile a una base dati che descrive le proprietà di tutti gli enti (variabili, costanti, procedure o metodi, classi ecc.) presenti nel programma. Ma non esiste uno standard, e ogni compilatore adotta la propria IR.

3. Fronte e retro

Gli analizzatori sintattico e semantico non dipendono dall'architettura del processore, ma soltanto dalle specifiche del linguaggio sorgente. Tale parte del compilatore è detta *fronte* o *tronco* comune. I metodi di progetto del tronco sono assestati e descritti in molti testi sui compilatori (per esempio, [1, 2]).
Sul tronco si innestano, come le branche d'un albero, altre passate dipendenti dalla macchina, note come *retro* (back-end), tra le quali spicca il generatore di codice.
La divisione tra fronte e retro facilita il progetto del compilatore, sia separando gli ambiti di competenza dei progettisti, sia rendendo possibile il riuso di una o dell'altra parte. Così il tronco d'un compilatore per il C potrà essere riutilizzato, sostituendo il generatore per la macchina A con quello per la macchina B. Viceversa due tronchi per il linguaggio C e C++ possono stare a monte dello stesso generatore di codice. Naturalmente, affinché il riuso sia possibile, le interfacce IR devono essere unificate. Ciò di norma avviene all'interno d'una piattaforma di compilazione, progettata da un'industria o organizzazione, per una gamma di linguaggi sorgente e di architetture; mentre l'interoperatività di compilatori di diversa origine è ostacolata dall'assenza di standard.

4. Meta-compilazione

Ben presto infastidì i progettisti la ripetizione del progetto del parsificatore, al mutare del linguaggio sorgente: di qui l'idea di produrre il parsificatore con un programma generatore, parametrizzato con le regole lessicali e sintattiche del linguaggio sorgente. Il generatore legge le regole (i meta-dati) e produce il programma del parsificatore specifico. Molti compilatori sono stati così prodotti, grazie alla disponibilità di buoni strumenti, dal classico *lex/yacc* (*flex/bison* nella versione libera di GNU) che produce un parsificatore ascendente in C, al più recente ANTLR, che genera un parsificatore a discesa ricorsiva in Java.
Perché non generare automaticamente anche l'analizzatore semantico, che è assai più complesso del parsificatore? L'idea ebbe comprensibilmente molti fautori, e alcuni strumenti (parame-

trizzati da una specifica semiformale delle azioni semantiche nota come *grammatiche con attributi*) ottennero qualche diffusione, per poi finire in disarmo. Due fattori giocano a sfavore dei metacompilatori semantici: la mancanza di una teoria formalizzata, semplice e condivisa; e il fatto che le tecniche di programmazione orientate agli oggetti già permettono un buon riuso di quella parte del compilatore che tratta le proprietà delle entità dichiarate nel programma sorgente.
L'uso di librerie di classi parametriche rispetto al linguaggio sorgente o all'architettura pozzo è frequente nelle piattaforme di compilazione sulle quali ritorneremo più avanti.

5. Generazione del codice

La più ovvia dipendenza dalla macchina viene dall'architettura dell'insieme di istruzioni (ISA). Fino agli anni 1980 il generatore di codice era scritto a mano da uno specialista del processore, abile nello sfruttare ogni istruzione. Ma la crescente complessità delle istruzioni delle macchine CISC[3] e la difficile manutenzione del generatore, stimolarono le ricerche sulla produzione automatica dei generatori. Ebbero successo i *generatori di generatori di codice*, impostati mediante la ricerca di forme (vedasi per esempio, [3]) sull'albero della IR. In breve, ogni istruzione macchina (per esempio, un'addizione tra un registro e una cella di memoria) è descritta formalmente come un *pattern*, ossia un frammento. La IR del programma deve essere ricoperta (tiling) con le forme corrispondenti ai pattern della macchina. Poiché molte ricoperture sono possibili, l'algoritmo deve orientare la scelta con opportune euristiche verso la copertura di costo minimo, dove il costo è il tempo di esecuzione del codice macchina.
Occorre dire che la ricerca della ricopertura ottimale dell'albero IR risulta molto più facile se l'architettura è del tipo RISC[4], nel qual caso è agevole scrivere a mano il generatore di codice.

6. Analisi statica di flusso

Per rendere eseguibile il codice, manca ancora un aspetto essenziale: la scelta dei registri del processore; infatti il passo precedente, per non affrontare in un colpo solo tutte le difficoltà, ha generato delle istruzioni che usano un numero illimitato di registri. Ma un cattivo uso dei registri provoca inutili copiature di dati, nonché rallentamenti dovuti agli accessi alla memoria. I processori moderni dispongono di tanti registri, che se ben sfruttati permettono di limitare gli accessi alla memoria.
L'assegnazione dei registri alle variabili è un problema di conflitto nell'uso di risorse scarse; poiché la ricerca della soluzione ottima è complessa, si deve ricorrere a metodi euristici.
Due variabili possono stare nello stesso registro se, in nessun punto del programma, entrambi i valori sono necessari: si dice che gli *intervalli di vita* delle due variabili sono disgiunti. Per assegnare i registri, è necessario calcolare gli intervalli di vita delle variabili. Il programma, ormai in codice macchina, è rappresentato dal suo *grafo di controllo* (CFG[5]), un'astrazione in cui ci si limita a guardare quali variabili siano lette o calcolate da ogni istruzione. Gli insiemi delle variabili vive in ogni punto del CFG sono calcolati risolvendo con metodo iterativo certe *equazioni di flusso*, facilmente ricavabili, istruzione per istruzione. La teoria delle equazioni di flusso (presente nei riferimenti [1, 2, 3, 4]) si fonda sull'algebra dei semi-anelli commutativi e trova applicazioni, non solo nella compilazione, ma anche nell'analisi e nella verifica dei programmi.
Risolte le equazioni, il compilatore, impiegando un metodo euristico, assegna registri diversi alle variabili che hanno intervalli di vita sovrapposti, senza superare per altro il numero di registri disponibili. In caso d'impossibilità, esso genera a malincuore del codice (spillatura) per copiare in memoria e poi riprendere i valori di certe variabili.

7. Ottimizzazioni: un catalogo senza fine

Ora il codice macchina è eseguibile, ma le sue prestazioni sarebbero inaccettabili, se non venisse migliorato con tante astute trasformazioni. L'ottimizzazione è la parte più impegnativa del compilatore, la cui complessità cresce con quella dei processori.
Conviene distinguere le trasformazioni indipen-

[3] Complex instruction set computer.

[4] Reduced instruction set computer.

[5] Control flow graph.

denti dalla macchina dalle altre. Le prime sono efficaci su ogni architettura. Si pensi al calcolo d'una espressione X + Y · Z che compare due volte nel programma, senza che le variabili cambino valore. Il compilatore salva in un registro il risultato del primo calcolo, e sostituisce il secondo con la copiatura del registro.
Un catalogo ragionato delle numerosissime trasformazioni esistenti è in Muchnick [4]. Una questione delicata è l'ordine in cui eseguire le trasformazioni; infatti una modifica, apparentemente non migliorativa, può abilitare altri miglioramenti.
Per esemplificare le ottimizzazioni dipendenti dalla macchina, possiamo soffermarci su due importanti aspetti architetturali.

8. Gerarchia di memoria

Per ridurre le attese (*latenze*) dovute alle operazioni sulla memoria RAM, le macchine usano memorie *cache*, più veloci ma molto più piccole. Un programma che con un ciclo iterativo spaziasse su troppe celle di memoria vanificherebbe l'efficacia della cache, che dovrebbe essere continuamente caricata e scaricata dalla memoria.
Concentrando l'attenzione sulle parti del programma eseguite più intensamente (i cosiddetti *nuclei* o *punti caldi*), l'ottimizzatore riorganizza i cicli in modo da operare su uno spazio di memoria contenibile nella cache. Inoltre esso inserisce anticipatamente nel codice le istruzioni di precaricamento (*prefetch*), che caricano la cache con i dati, in anticipo rispetto al momento in cui saranno richiesti.

9. Parallelismo schedulazione e predicazione

Le macchine offrono tante forme di parallelismo: pipeline, disponibilità di più unità funzionali, istruzioni composte da più codici operativi (architettura VLIW[6]), fino alle architetture multiprocessore.
Per sfruttare il pipeline, un buon codice deve rarefare le istruzioni di salto. Una trasformazione migliorativa consiste nell'ingrandire le dimensioni dei *blocchi basici* (una serie di istruzioni non interrotte da salti né da etichette), magari srotolando il corpo di un ciclo iterativo due o più volte.
Le architetture VLIW bene illustrano il problema della *schedulazione delle istruzioni*, ossia del loro riordino rispetto all'ordine originale, allo scopo di utilizzare al meglio in ogni istante di clock tutte le unità funzionali della macchina.
La schedulazione è semplice per un blocco basico, ma diventa intrattabile per l'intero programma. Il compilatore impiega una gamma di schedulatori, appropriati per diverse parti del programma: l'algoritmo di list scheduling (simile al job shop scheduling della ricerca operativa) si addice alla parti acicliche; l'elegante algoritmo di software pipelining e modulo scheduling si applica ai cicli più caldi del programma, l'algoritmo di trace scheduling privilegia la *traccia* del programma eseguita più spesso, anche al costo di duplicare del codice nelle altre parti.
Un altro meccanismo, spesso combinato con il parallelismo, è l'esecuzione *speculativa e predicativa* del codice. L'idea è che nessuna unità funzionale deve essere lasciata inerte. Se per esempio, un moltiplicatore fosse disponibile, ma la o le istruzioni in fase di esecuzione non lo richiedessero, sarebbe un peccato. Il compilatore, analizzando il codice, sceglie una futura e distante istruzione di moltiplicazione da far eseguire anticipatamente, pur senza la certezza che il flusso di controllo la raggiungerà.
Per confermare l'effetto dell'istruzione, quando si avrà la certezza della sua utilità, un bit o predicato, riceve il valore da un'apposita istruzione.
L'esecuzione predicativa non è indicata per dispositivi alimentati a batteria, perché consuma energia per eseguire operazioni talvolta inutili.

10. Elaborazione di flussi multi mediali

La diffusione dei dispositivi multi-mediali ha creato una nuova classe di coprocessori di flusso (stream), realizzati con architetture molto più veloci dei microprocessori, per tali elaborazioni. L'elaborazione consiste in un ciclo di istruzioni ripetute indefinitivamente sul flusso d'ingresso, per produrre quello d'uscita. La presenza di numerose unità funzionali inter-

[6] Very long instruction word.

connesse da una rete rende difficile la programmazione manuale e motiva lo sviluppo di compilatori capaci di estrarre da un programma generico le parti destinate al coprocessore di flusso e di tradurle nelle istruzioni del coprocessore stesso. La traduzione presenta difficili problemi di assegnazione ottimale delle istruzioni alle unità funzionali.
L'esempio dei processori di flusso è emblematico della compilazione per architetture speciali, un campo in sviluppo grazie alla flessibilità offerta dalle moderne tecniche di progetto dello hardware.
Ogni trasformazione ottimizzante dà un piccolo contributo (si spera positivo ma non sempre è così!) all'efficienza del codice macchina. Aggiungendo con abilità e tenacia qualche punto percentuale qua e là, si ottengono le buone prestazioni che qualificano i compilatori.

11. Interpretazione e compilazione dinamica

La Rete ha fatto emergere nuove esigenze, perché gli applicativi possono essere trasferiti da una macchina remota fornitrice a quella dell'utente, proprio al momento dell'esecuzione. Non essendo praticamente possibile trasferire i codici binari, perché la macchina utente è in generale ignota, la soluzione, affermatasi con il linguaggio Java, è l'*interpretazione*. L'applicativo, tradotto dal fornitore in un linguaggio intermedio o virtuale (ByteCode) indipendente dalla macchina, è spedito alla macchina destinataria, dove è eseguito da una *macchina virtuale* VM. In questo modo l'onere di adattare il linguaggio Java all'architettura è spostato dal mittente al destinatario dell'applicativo.
Ma la velocità dell'applicativo interpretato è almeno un ordine di grandezza inferiore, rispetto al codice compilato. Come conciliare l'indipendenza dalla macchina con l'efficienza? La risposta sta nella *compilazione dinamica* (anche detta JIT *just in time*). Accanto all'interprete, vi è un traduttore che converte le istruzioni virtuali in quelle del processore. Poiché, quando gira il traduttore, l'applicativo è fermo, e l'utente si spazientisce, occorre ridurre il tempo di compilazione. Tipicamente, l'esecuzione inizia mediante la macchina virtuale, poi parte il compilatore dinamico che traduce le parti calde, limitando le ottimizzazioni a quelle meno faticose. Per esempio l'assegnazione dei registri fisici è fatta con algoritmi più sbrigativi di quelli utilizzati da un compilatore statico. Occorre preliminarmente identificare le parti calde, per mezzo di tecniche di analisi statica e di conteggio dinamico (*profiling*) delle istruzioni eseguite.
Il compilatore dinamico deve essere programmato con estrema cura, per portare frutto. Una rassegna della compilazione dinamica è in [5].
Non sempre però la compilazione dinamica è possibile. I piccoli dispostivi, come i telefonini, non hanno sufficiente memoria per il compilatore dinamico. Per essi, la macchina virtuale resta l'unica possibilità, che deve sfruttare ogni possibile risparmio di memoria, senza per altro sacrificare la velocità.

12. Altri usi della compilazione dinamica

La compilazione dinamica ha altre applicazioni: la *traduzione da binario a binario* trasforma, al momento dell'esecuzione, il codice macchina della macchina A in quello della macchina B, curando anche la conversione delle chiamate di sistema operativo. Questa tecnica è molto utilizzata per emulare un'architettura ISA su di un'altra, per esempio, l'architettura Intel x86 sull'architettura TransMeta.
La *ottimizzazione continua* si applica sulle macchine, per esempio, i serventi delle basi dati, in ciclo continuo, con programmi in cui il profilo del carico da eseguire cambia, mettiamo, tra un giorno e il successivo.
Il codice macchina potrà essere ottimizzato in funzione del profilo attuale del carico. Per esempio, se una certa procedura in un certo giorno è invocata con un parametro fissato su un valore costante, il compilatore può *specializzare* il corpo della procedura, propagando il valore costante e eliminando eventuali test sul valore del parametro.
In un campo diverso, con dispositivi a batteria, si usa la compilazione dinamica per ridurre la potenza elettrica assorbita. È noto che la potenza cresce con il quadrato della frequenza del clock e che i nuovi processori dispongono di comandi per variare la frequenza di lavoro. Se l'applicativo in esecuzione ha sufficienti margini rispetto alle scadenze di tempo reale

da rispettare, la frequenza può essere rallentata, riducendo il consumo. È un monitore dinamico che valuta le opportunità e produce le istruzioni necessarie.

13. Infrastrutture aperte per la compilazione

Oggi chi deve sviluppare un compilatore ha la possibilità di partire dall'esistente e di modificarlo, e solo raramente si impostano da zero dei nuovi traduttori. Molti compilatori di proprietà di aziende sono l'evoluzione di progetti, anche molto vecchi, che via via si adeguano per le nuove architetture. Inoltre varie comunità hanno progettato e mantengono aggiornate le cosiddette infrastrutture aperte (o piattaforme) per la compilazione, veri progetti collettivi di componenti software per la costruzione dei compilatori.
La maggior parte delle piattaforme opera sui linguaggi più classici e diffusi: C, C++, Java e C#. Per inciso, nonostante la fioritura di tante proposte linguistiche, C e C++ restano la scelta preferita di quanti devono scrivere i compilatori e il software di sistema.
Altre comunità lavorano sui linguaggi funzionali e sui linguaggi interpretati di alto livello, come Python.
Limitandoci alle infrastrutture più diffuse, la più veneranda è *SUIF* della Stanford University, "una piattaforma libera progettata per aiutare la ricerca collaborativa sui compilatori ottimizzanti e parallelizzanti". È facile scrivere in C++ le proprie passate di compilazione appoggiandosi alle pratiche rappresentazioni IR di SUIF. Sono disponibili i fronti per C, C++, Fortran, e Java, e i retro per vari microprocessori. Una funzionalità, forse sorprendente ma utilissima, ritraduce in C un programma, dal linguaggio intermedio IR di SUIF. Ciò permette di analizzare l'effetto delle ottimizzazioni prodotte e di confrontarle con quelle di compilatori rivali.
Il grande progetto *GCC* (GNU Compiler Collection) fa capo alla organizzazione GNU e mira a fornire una linea di compilatori liberi, della stessa qualità di quelli industriali. Vi sono fronti per C, C++, Fortran, Java e Ada e retro per x86 e altre macchine.
Una gemmazione del progetto, DotGNU, si è rivolta ai linguaggi programmativi e per i servizi di Rete della Microsoft. Si ricorda che l'insieme Dot Net di Microsoft è principalmente un linguaggio intermedio, ispirato dal ByteCode di Java, di cui allarga le potenzialità e l'idoneità per diversi linguaggi sorgente.
L'affermarsi del progetto GCC ha l'effetto positivo di consolidare certi standard di fatto per le rappresentazioni intermedie, e di sottrarre lo sviluppo dei linguaggi al controllo esclusivo di una o poche aziende.
Sempre per le piattaforme Dot Net, ha avuto buona diffusione l'infrastruttura MONO, libera ma sponsorizzata da Novell.
Per il linguaggio Java esistono varie disponibilità di macchine virtuali e di compilatori dinamici; Kaffe è una delle più note. Sono questi però dei progetti di minor respiro e di sopravvivenza talvolta incerta.

14. Il futuro

L'innovazione nello sviluppo dei linguaggi di programmazione segna il passo, e i migliori tra i nuovi linguaggi che si affacciano mi sembrano delle abili rivisitazioni dei concetti classici, per adattarli a aspettative e motivazioni alquanto mutate. Il progetto dei loro compilatori (ossia dei fronti) sarà un lavoro routinario per i professionisti della compilazione.
Le direzioni di principale innovazione per la compilazione sono quelle attinenti allo sviluppo di nuove architetture di calcolo.
Per i processori, la diffusione a breve e medio termine delle architetture (esempio, Intel multi-core) con più processori interconnessi in rete, impone una complessa sinergia con il compilatore, al fine di parallelizzare e ripartire il codice, prima sui processori e poi sulle loro unità funzionali.
In un'altra direzione, per portare i linguaggi sui più piccoli dispositivi, sarà sempre necessario un artigianato di grande abilità, con attenzione alle tecniche di programmazione più compatte e efficienti.
Più a lungo termine c'è chi prevede una drastica caduta di affidabilità dei processori, causata da guasti transienti, provocati sui piccolissimi transistori delle fluttuazioni ambientali. In tale prospettiva, il compilatore sarà chiamato a generare codici eseguibili ridondanti, capaci di scoprire, e mascherare gli errori di basso livello.

Per quanto riguarda le Reti, il supporto necessario per far funzionare in modo sicuro i servizi e i programmi mobili sarà necessariamente dinamico, e la compilazione dinamica dovrà perfezionare i propri metodi di progetto, di convalida e di manutenzione.

Bibliografia

[1] Crespi Reghizzi S.: *Linguaggi formali e compilazione*. Pitagora, Bologna 2006.

[2] Aho A., Sethi R., Ullman J., Lam M.: *Compilers: Principles*. Techniques, and Tools, Addison Wesley, 2006.

[3] Appel A.: *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[4] Muchnick S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[5] Duesterwald E.: *Dynamic Compilation*. In Srikant Y. N., Shankar Priti (a cura di), *The Compiler Design Handbook*. CRC Press, 2002.

[6] Ujval Kapasi et al.: Programmable Stream Processors. *Computer*, Vol. 36, n. 8, p. 54-62., 2003.

Stefano Crespi Reghizzi lavora nel Dipartimento di Elettronica e Informazione del Politecnico di Milano. Il gruppo di ricerca da lui guidato studia la teoria dei linguaggi artificiali e progetta compilatori e macchine virtuali per i moderni processori.
Coordina il dottorato di ricerca in Ingegneria dell'Informazione: automatica, elettronica, informatica e telecomunicazioni. Insegna i corsi di Linguaggi formali e compilatori e di Analisi e ottimizzazione dei programmi, per la laurea in ingegneria informatica.
Ingegnere elettronico, ha conseguito il dottorato in computer science alla University della California di Los Angeles. Ha insegnato e collaborato scientificamente nelle università di Berkeley, Lugano, Pisa, Santiago de Chile, Stanford e Parigi. È uno dei fondatori del programma scientifico internazionale "Automata theory from mathematics to applications" della ESF, la fondazione europea per la scienza. Tra le sue opere sulla compilazione si ricorda il libro "Linguaggi formali nelle scienze della comunicazione", rivolto a lettori di matrice umanistica e il testo universitario "Linguaggi formali e compilazione" per gli ingegneri informatici.
E-mail: crespi@elet.polimi.it