LuajitTEX

Luigi Scarso

Sommario

LuajitTEX è una variante di LUATEX che utilizza LuaJIT, un compilatore just-in-time per Lua. Vengono illustrati alcuni aspetti degli interpreti TEX, Lua e LuaJIT e vengono brevemente riferiti i primi risultati che evidenziano un guadagno medio del 25% del tempo di esecuzione.

Abstract

LuajitTEX is a LUATEX version that uses LuaJIT, a just-in-time Lua compiler. This paper shows some aspects of TEX, Lua and LuaJIT interpreters and shortly reports early results that highlight how execution time speeds up by 25% on average.

1 Introduzione

Quest'anno ricorre il ventesimo anno di Lua, la cui origine è descritta in IERUSALIMSCHY *et al.* (2007) (disponibile anche presso `http://www.lua.org/doc/hopl.pdf`) e che a partire dal 2005 è parte integrante del motore di impaginazione LUATEX, giunto oggi alla versione `0.77` — cioè, in modo un po' impreciso, al 77% della versione finale. Quasi a voler celebrare questa data, il team di sviluppo ha pubblicato nel Gennaio 2013 una nuova edizione del testo *Programming in Lua* (IERUSALIMSCHY, 2013) che fa da riferimento per la versione `5.2` del linguaggio pubblicata nel Dicembre 2011. Dato che il team di LUATEX tende ad usare l'ultima versione standard di Lua, nel Novembre 2012 è cominciato l'adattamento del codice base per integrare la versione `5.2` (lavoro terminato con la versione `0.75`), e questo è sembrato il momento opportuno per sperimentare alcune modifiche. Una di queste era verificare la possibilità di utilizzare un interprete Lua alternativo, LuaJIT di Mike Pall (`http://luajit.org`), giunto alla versione `2.0.0` proprio nel Novembre 2012 dopo un periodo di beta release di quasi tre anni. L'utilizzazione di LuaJIT porterebbe un considerevole guadagno di velocità (speed up) rispetto al tradizionale interprete: almeno due volte più veloce (speed up 2×) ed in particolari situazioni addirittura 110× (cfr. `http://luajit.org/ext_ffi.html`). Tuttavia bisogna considerare che LUATEX non usa *di per sé* programmi Lua: solo formati come LUALATEX o ConTEXt-MKIV hanno una consistente base di script Lua e quindi beneficierebbero in teoria dello speed up. Inoltre, sebbene LuaJIT e Lua siano API-compatibili (ovvero offrano la stessa interfaccia di programmazione) ed entrambi implementino un interprete Lua, internamente sono completamente differenti e non è chiaro a priori se anche LUATEX possa effettivamente trarre vantaggio o meno dalla diversa implementazione. Il lavoro su LuajitTEX è stata anche l'occasione per riconsiderare TEX e Lua nella loro specificità, ossia quella di essere interpreti: nelle prossime sezioni descriveremo TEX, Lua e LuaJIT considerandoli come programmi che processano un linguaggio. Vedremo la differenza tra un compilatore ed un interpretere e come questa differenza sfumi negli interpreti altamente efficienti come LuaJIT.

2 TEX, Lua, LuaJIT

2.1 Compilatori ed interpreti

Un compilatore è un programma che traduce un programma *sorgente* in un secondo programma *oggetto*. Questa definizione è piuttosto generica: nella pratica il sorgente è scritto in un linguaggio ad alto livello che permette cioè di esprimere concetti come cicli, condizioni, assegnazioni, operazioni aritmetiche senza far riferimento ad un preciso agente di calcolo utilizzando un linguaggio simile a quello naturale, mentre il codice oggetto è eseguibile da un agente di calcolo meccanico, come ad esempio i moderni microprocessori. Nel seguito con sorgente intenderemo un programma in un linguaggio ad alto livello, mentre con codice oggetto un programma adatto per microprocessori. Codice nativo e microcodice sono talvolta usati come sinonimi di codice oggetto.

La costruzione di un compilatore dipende sia dalle specifiche del linguaggio del sorgente che dalle specifiche del linguaggio oggetto: le prime cambiano lentamente, ma sono complicate da implementare, le seconde invece sono più facili da implementare ma cambiamo piuttosto frequentemente, in quanto cambiano i microprocessori disponibili sul mercato. Consideriamo ad esempio il compilatore GCC: accetta come sorgente programmi scritti in Ada, C, C++, Fortran, Go, Java, Objective-C, Objective-C++ e può produrre codice oggetto per svariate decine di famiglie di microprocessori — quindi circa un centinaio di microprocessori. Implementando separatamente un compilatore per ciascuna coppia (sorgente, oggetto) si avrebbero 10 linguaggi × 100 microprocessori = 1000 compilatori: chiaramente la gestione diventa un compito praticamente impossibile.

Una soluzione consiste nel suddividere il compilatore in uno stadio iniziale (*front end*) che si

occupa di un solo linguaggio sorgente ed opera come traduttore verso una forma intermedia IR (*Intermediate Representation*). Un secondo stadio (*middle end*) si occupa della ottimizzazione del formato IR ed un terzo stadio (*back end*) traduce la forma IR ottimizzata in codice assembly il quale viene a sua volta tradotto in codice oggetto da un assemblatore. Seguendo l'esempio precedente, in questo modo si hanno 10 linguaggi $\rightarrow$ 10 traduttori verso IR, 1 ottimizzatore di IR ed 1 backend $\times$ 100 microprocessori $\rightarrow$ 100 assemblatori per un totale di 111 componenti del compilatore — quindi rispetto a prima la complessità è ridotta di un ordine di grandezza.

Con riferimento alla fig. 1, il front end è suddiviso in tre fasi:

analizzatore lessicale (*Lexical Analyzer — lexer* per brevità — o *scanner*) suddivide il sorgente in *token*;

analizzatore sintattico (*Syntax Analyzer* o *parser*) raggruppa i token in unità sintattiche in modo da rispettare la grammatica del linguaggio. Questa fase produce una struttura dati chiamata albero della sintassi (*syntax tree*) del sorgente;

analizzatore semantico (*Semantic Analyzer*) controlla la correttezza semantica del syntax tree secondo le regole della grammatica — per esempio sommare un numero ed una stringa è un errore se il linguaggio non prevedere una conversione automatica tra i due tipi `numero` e `stringa`; non lo è se, per esempio, `numero` può essere trasformato in `stringa` e la somma di due stringhe (unione) è lecita.

Possiamo vedere il middle end come una sorta di minicompilatore suddiviso in due stadi:

generatore di codice intermedio (*Intermediate Code Generation*) ha in ingresso il syntax tree (il "sorgente" del minicompilatore) e produce il corrispondente IR, il "codice oggetto" di un microprocessore virtuale (ad esempio, ha un numero illimitato di registri);

ottimizzatore codice intermedio (*Code optimization*), probabilmente è lo stadio più complesso di tutto il compilatore.

Il middle end è una sorta di "camera di compensazione" tra due entità molto differenti: il linguaggio ad alto livello, comprensibile ma astratto, ed una miriade di microcodici (uno per ogni microprocessore) concreti sì, ma ciascuno con le proprie caratteristiche. Il formato IR deve essere facile da produrre e da tradurre in assembly, quindi la CPU virtuale deve astrarre le caratteristiche di ogni CPU concreta, ed è anche il formato utilizzato per la maggior parte delle ottimizzazioni. Alcune di queste sono semplici: per esempio nel seguente codice `C`

```
static void relax(){;}
int main(){return 0;}
```

la funzione `relax()` è inutile in quanto non sarà mai invocata e quindi può essere eliminata (*dead code*). È evidente che è meglio affrontare questa ottimizzazione (*Dead Code Elimination*) nel middle end, dato che nel front end dipende dalla grammatica del linguaggio sorgente e nel contempo si riduce la dimensione del codice assembly. Ma altre ottimizzazioni sono decisamente più complesse: ad esempio la maggior parte dei linguaggi high level non mette limiti al numero di variabili utilizzabile ed il compilatore assegna ad una variabile una distinta area di memoria. I registri di una CPU sono le unità di memoria più veloci, quindi è naturale cercare di usarli il più possibile: essendo il loro numero limitato sorge il problema della allocazione dei registri disponibili (*register allocation*). Purtroppo questo è un problema NP-completo: non si conosce una soluzione "buona" (per esempio in tempo polinomiale rispetto alla dimensione dei dati in ingresso), per cui si devono adottare tecniche euristiche che impattano in modo negativo sia sulla velocità del compilatore sia sulle sue dimensioni.

Infine nel back end abbiamo:

generatore codice assembly (*Target Code Generator*) che traduce il codice IR in assembly;

ottimizzatore codice assembly (*Target Code Optimizer*) in questo stadio è possibile effettuare una ulteriore riduzione del codice assembly sostituendo, quando possibile, un gruppo di istruzioni con uno più breve (*peephole optimization*). Non sempre presente nel back end, alcuni compilatori spostano l'ottimizzatore nel middle end;

assemblatore (*Assembler*) che traduce il codice assembly in codice oggetto. Tecnicamente è un programma distinto dal compilatore vero e proprio, ma è pratica consolidata che il compilatore di default esegua l'assemblatore al termine della fase di ottimizzazione.

Una ulteriore complicazione nasce dalla fondamentale necessità di eseguire passo dopo passo un programma compilato, tipicamente per verificare il motivo di un malfunzionamento e correggere il codice sorgente nel punto corretto (*debugging*). In questo caso è necessario mappare il codice oggetto verso il codice sorgente e questa "mappa" è immersa nel codice oggetto stesso e non è raro che questo comporti una aumento delle dimensioni anche di 4 volte; inoltre le ottimizzazioni come la Dead Code Elimination possono interferire con il flusso di esecuzione descritto dal sorgente e possono quindi confondere il programmatore, per cui vanno disabilitate — in sostanza il debugging richiede di progettare il compilatore *anche* per poter produrre un codice altamente inefficiente sia per le dimensioni che per la velocità.

Un compilatore *produce* codice oggetto, ma non lo esegue. Da questo punto di vista è differente

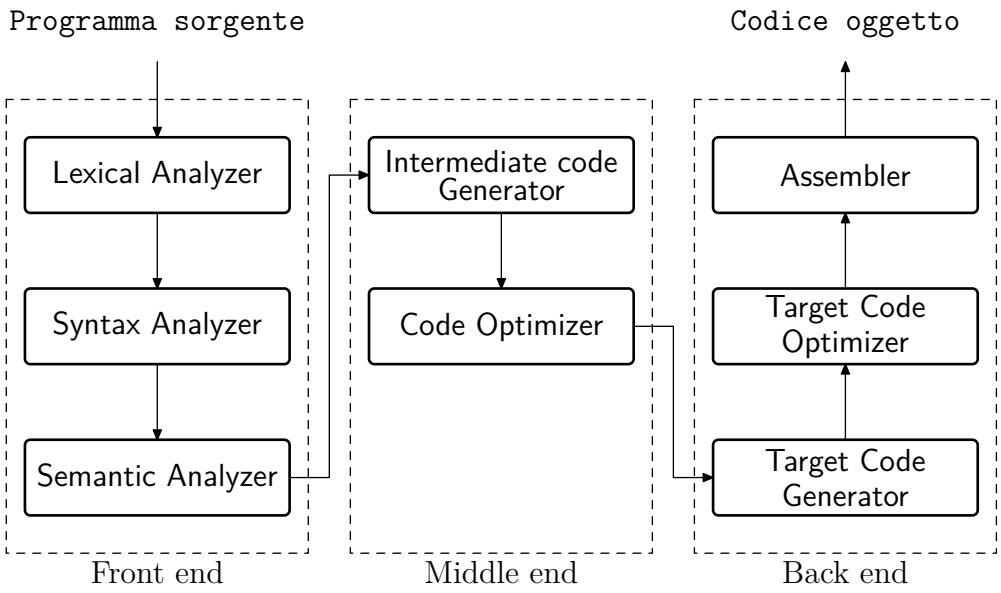

Figura 1: Stadi di un compilatore.

da un *interprete* di un linguaggio, il quale *esegue* un programma sorgente *una istruzione alla volta* fino a quando non vi sono più istruzioni da eseguire (termine dell'input, oppure `halt` dell'esecuzione per errore irreversibile, oppure uscita anticipata). Questo processo può anche essere interattivo: si legge una istruzione da terminale, la si valuta, si stampa l'eventuale risultato e si ripete il ciclo (*read-eval-print-loop* o REPL). Solitamente i linguaggi interpretati prevedono un programma REPL che viene distribuito come parte integrante dell'interprete.

Ogni CPU è un interprete del proprio microcodice: infatti ad intervalli di tempo scanditi dal clock del sistema, la CPU preleva una istruzione dalla memoria, la decodifica per trovare l'operazione da eseguire ed i relativi operandi, la esegue e ripete il ciclo (*fetch-decode-execute cycle*). Scrivere in assembly un interprete di un linguaggio high level (o very high level) non è un compito precisamente semplice, ma la teoria dei compilatori ci soccorre anche in questo caso, e con riferimento alla fig. 2 e seguendo (Parr, 2009), distinguiamo diversi tipi di interpreti identificandoli con le diverse fasi del compilatore (sebbene non vi sia una corrispondenza uno a uno):

Syntax-Directed Interpreter: appena il lexer produce un token, viene eseguita l'azione corrispondente e si ripete quindi il ciclo tornando al lexer. Semplice da costruire se il linguaggio è elementare, un SDI diventa rapidamente complicato non appena si introducono le istruzioni condizionali, i cicli, le funzioni: nel caso di una funzione, ad esempio, il codice sorgente viene memorizzato integralmente e non è possibile invocare subito l'azione corrispondente, ma solo al momento della sua chiamata (*forward reference*); inoltre ad ogni chiamata lo scanner usa sempre lo stesso codice sorgente con un impatto negativo sul tempo di esecuzione;

Tree Interpreter: costruisce il syntax tree dell'input e lo utilizza come input per l'interprete. Il vantaggio principale è la separazione tra dichiarazione di un simbolo (variabile, funzione) e risoluzione del suo attuale valore, permettendo così le forward references ed evitando il reparsing del codice sorgente. Un syntax tree inoltre può essere ottimizzato per l'esecuzione, ma per attivare l'azione l'interprete deve prima visitare un sottoalbero completo, operazione più complessa rispetto ad un SDI. In ogni caso sia un SDI che un Tree Interpreter sono indicati per linguaggi specifici per area di applicazione (*Domain Specific Language* o DSL) dove spesso la grammatica non è complicata ed i programmi si riducono a flussi di comandi; programmi come lex o yacc permettono anche di automatizzare la costruzione del lexer e del parser partendo da una specifica formale, tipicamente in formato EBNF (cfr. Levine *et al.* (1992) e Levine (2009));

Bytecode Interpreter: il codice sorgente viene tradotto in un linguaggio macchina (detto *bytecode*, in quanto vi sono al massimo 256 tipi — *opcodes* — di istruzioni e quindi un opcode è codificato in un byte) di una CPU virtuale e l'interprete (chiamato, di conseguenza, *Virtual Machine* o *VM*) emula questa CPU. È simile ad un formato IR, ma mentre quest'ultimo è pensato per l'ottimizzazione, il bytecode separa il problema della complessità del parsing (astratto e "lontano" dall'hardware) da quello dell'efficienza (ottimizzazione della memoria e della CPU reale). Ciò agevola l'implementazione della VM su diverse architetture mantenendo inalterata la grammatica del linguaggio. Per que-

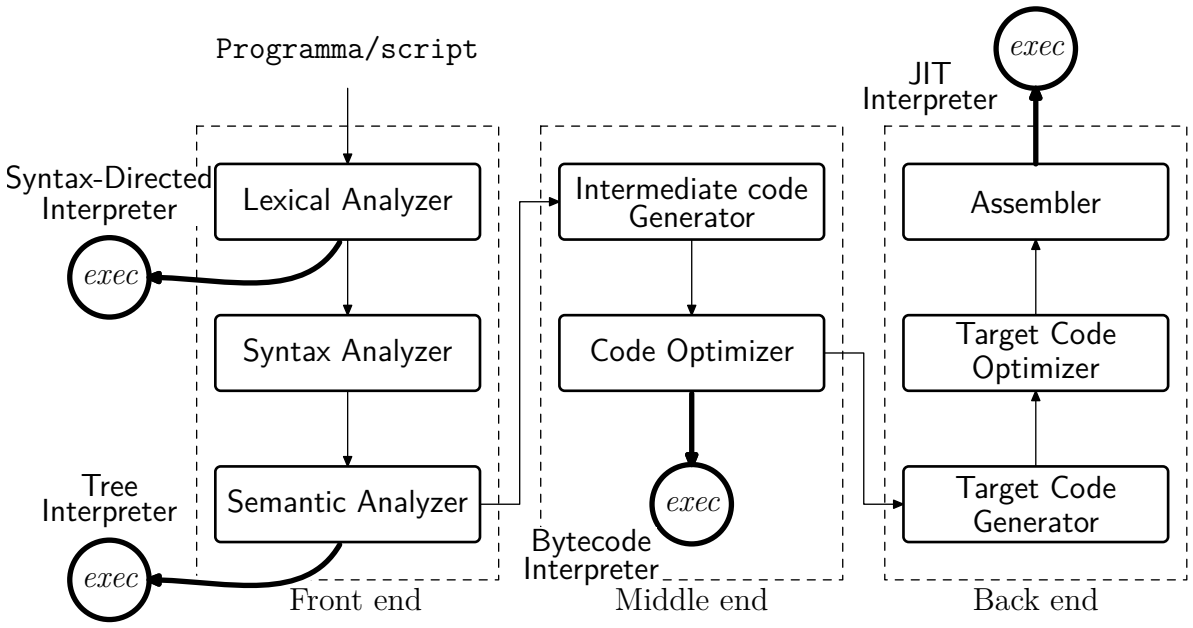

FIGURA 2: I diversi tipi di interpreti vs gli stadi di un compilatore.

sto motivo è il tipo di interprete più usato per i linguaggi ad uso generale (*general purpose*).

Analogamente ad una CPU reale, anche una VM ha tre fasi (SHI *et al.*, 2008): prelievo del bytecode dalla memoria e salto alla corrispondente azione (*dispatching*), individuazione degli operandi ed esecuzione. Normalmente il dispatching è realizzato in C/C++ come un ciclo `while` che contiene uno `switch` con tanti `case` quanti sono le opcodes del linguaggio, mentre, a seconda della struttura dati per gestire le variabili temporanee, vi sono due tipi di VM: 1) *stack based*, dove viene usato una pila (*stack*), di semplice realizzazione ma che tende a produrre bytecode più articolato; 2) *register based* dove vengono usati i registri in modo del tutto analogo ad una CPU reale, anche se in questo caso non vi sono limiti al numero di registri possibili. Il Bytecode Interpreter produce bytecode più compatto, anche se è di implementazione più complessa.

Un Bytecode Interpreter ha la possibilità di salvare il sorgente in bytecode, eliminando così la necessità di parsing ed aumentando la velocità di esecuzione dell'interprete — naturalmente se il sorgente cambia bisogna ricompilarlo, in modo analogo ad un compilatore. Gli interpreti efficienti sono in grado di ricompilare solo i sorgenti modificati e questo è di enorme aiuto nel caso di progetti con molti moduli interdipendenti, perché evita di ricompilare tutti i moduli indistintamente.

JIT Interpreter: è una specializzazione del Bytecode Interpreter e nasce dalla constatazione che alcune parti di bytecode generato non cambiano e quindi possono essere tradotte a tempo di esecuzione (*run time*) in codice macchina della CPU reale "al momento" (*just in time* or *JIT*). Un JIT Interpreter deve quindi saper riconoscere quali parti di bytecode sono *hot* (e quindi mantere una statistica del loro uso), valutare se è possibile tradurle in codice macchina e, nel caso, produrre il relativo codice, eventualmente ottimizzarlo ed infine eseguirlo. È evidente che si tratta di un interprete decisamente più complesso dei precedenti ma le prestazioni possono essere simili ad un codice compilato o, in alcuni casi dove l'ottimizzazione dinamica è più efficiente di quella statica del compilatore, addirittura superiori — ed è questo che giustifica l'investimento di risorse per la costruzione di un JIT Interpreter. La presenza di codice macchina implica ovviamente che questo interprete è specifico per una coppia CPU/Sistema Operativo ma apre anche un problema di sicurezza. Molte CPU infatti hanno un flag che impedisce ad un processo di eseguire codice macchina salvato nel proprio segmento dati e settare questo flag *può* richiedere di privilegi di amministratore. Questo significa che un JIT interpreter potrebbe non essere eseguito se privo di un certificato di idoneità, magari rilasciato da un ente indipendente dall'amministratore del sistema.

Sia un Bytecode che un JIT Interpreter prevedono di ottimizzare il bytecode, ma non bisogna comunque dimenticare l'influenza del run time: ad esempio nel programma Lua `function relax() end return 0` simile al codice C visto precedentemente, è possibile eliminare `relax()` solo quando si esegue `return 0` — ossia quanto è troppo tardi — dato che solo a quel momento l'interprete sa che `relax()` non verrà più chiamata.

È interessante osservare come Java riassume quanto visto fino ad ora. Java è un linguaggio com-

pilato high level e general purpose, ispirato al C++ ed ObjectiveC e quindi Object Oriented, creato da SUN Microsystems (successivamente acquisita da Oracle) agli inizi degli anni 90. Il suo compilatore `javac` produce esclusivamente bytecode (i *class files*) per la macchina virtuale `java` (la Java VM o JVM), originariamente un Bytecode Interpreter stack based successivamente evolutosi in un JIT Interpreter (*HotSpot*^{TM}, attualmente mantenuto da Oracle). HotSpot^{TM} è probabilmente meglio descrivibile come un trans-compilatore: il bytecode in ingresso viene tradotto in una IR ad alto livello (*HIR*) la quale viene ottimizzata e tradotta in una IR a basso livello (*LIR*) ulteriormente ottimizzata con una tecnica peephole e resa disponibile per la traduzione in codice macchina. Il termine Hotspot[1], come visto precedentemente, indica che le ottimizzazioni riguardano le parti maggiormente eseguite del flusso bytecode e quindi la VM mantiene una statistica del flusso.

Una fondamentale caratteristica di Java è che un class file è eseguibile ovunque sia presente una JVM, indipendentemente dal CPU/Sistema Operativo. `Dalvik`, sviluppata da Google per il proprio SO Android pensato per dispositivi mobili, è un altro JIT Interpreter register based per Java, ma il bytecode non è compatibile con la JVM. Sebbene in Shi *et al.* (2008) sia presentata una JVM register based più performante rispetto ad una stack based, presso `https://blogs.oracle.com/javaseembedded/entry/how_does_android_22s_performance_stack_up_against_java_se_embedded` sono disponibili dei test che puntano in direzione contraria, per cui la questione rimane ancora controversa.

2.2 TEX

Per la maggior parte degli utenti TEX, `pdflatex`, `luatex` o `xetex` sono programmi che trasformano un testo in un file `dvi` o `ps` o `pdf` (possibilmente mediante programmi come `dvips` o `dvipdf`). Superficialmente (ma non troppo, visto che gli sviluppatori parlano di back end dvi o back end pdf) la somiglianza con un compilatore è evidente: il testo è un programma in un linguaggio ad alto livello e il file di uscita ha un formato binario (almeno fino a quando ci si si limita a `dvi` o `pdf`). Visto che il formato `pdf` probabilmente è il più usato, non è improprio vedere `tex` come un *compilatore di documenti*: in questo senso è simile a `javac`, con il `pdf` al posto del bytecode. Sebbene questo punto di vista sia accettabile (e nella pratica consolidato) tuttavia si focalizza sulla relazione input-output e nasconde importanti parti che rimangono comunque interessanti da vedere.

Innanzitutto TEX è un linguaggio di programmazione ad alto livello dedicato alla tipografia digitale (quindi un DSL); è un linguaggio a marcatori di tipo procedurale, ed i marcatori sono macro definibili dall'utente. Esistono diversi programmi che sono in grado di leggere sorgenti TEX e la maggior parte di questi (o perlomeno quelli noti all'autore) sono interpreti con REPL. Analizzando il codice sorgente di `pdftex` e `luatex` si nota che la funzione `main_control` implementa il dispatching delle istruzioni: il codice sorgente viene letto sequenzialmente in spezzoni, che vengono a loro volta decomposti in coppie `(character-code, category-code)` — i `token`. Ad esempio `(ch('a'),11)` e `(ch('b'),11)` sono due token diversi perché il character code (indicato dalla funzione `ch()`) è differente, ma entrambe sono della stessa categoria sintattica `11` (lettere) e generano quindi lo stesso tipo di token. La coppia `(ch('\'),0)` invece è carattere di tipo escape per cui il resto dell'input va trattato in modo particolare: ad esempio se l'input contiene `\relax␣foo` il carattere \ segnala a TEX di produrre un token di tipo macro identificata dai caratteri `'r' 'e' 'l' 'a' 'x'` (e delimitata da ␣) che vengono quindi consumati e non trattati come lettere.
Qui sotto viene riportata la funzione relativa a `luatex`, ma quella di `pdftex` è simile:

```
void main_control(void){
  main_control_state = goto_next;
  init_main_control();
  if (equiv(every_job_loc) != null)
     begin_token_list(equiv(every_job_loc),
                      every_job_text);
  while (1) {
     if (main_control_state==goto_skip_token)
       main_control_state = goto_next;
     else
       get_x_token();
     if (interrupt != 0 && OK_to_interrupt) {
       back_input();
       check_interrupt();
       continue;
     }
     if (int_par(tracing_commands_code)> 0)
       show_cur_cmd_chr();

     (jump_table[(abs(mode)+cur_cmd)])();
     if (main_control_state == goto_return) {
       return;
     }
  }
  return;
}
```

Il punto chiave è
`jump_table[(abs(mode)+cur_cmd)])()`:
`jump_table` è un array che memorizza le funzioni (o più precisamente i puntatori alle funzioni) da eseguire per ciascun tipo di token in relazione allo stato corrente. È una variante del classico `while` con `switch-case`, il quale infatti è usato nel codice di `pdftex` e di cui si vede

1. In `http://www.ntg.nl/maps/16/14.pdf` D. Knuth lo indica con "jump trace"

ancora traccia nei tre `if-then` presenti all'interno del ciclo `while(1)`. Usare una jump table è una soluzione vantaggiosa perché mantiene il codice più corto e quindi gestibile. Un interprete TEX è quindi un SDI anche se, da questo punto di vista, è un caso limite: è inusuale per un interprete DSL un così fine controllo sull'input. Ad esempio è possibile, usando `catcode`, ridefinire la categoria dei caratteri in modo tale da trasformare TEX in un altro interprete per i records di bibTEX:

```
@ARTICLE{Ertl03thestructure,
    author = "M. Anton Ertl and David Gregg",
    title = "The Structure and Performance of
             Efficient Interpreters",
    journal = "Journal of Instruction-Level
               Parallelism",
    year = "2003",
    volume = "5",
    pages = "2003"
}
```

L'utente ha quindi il controllo del lexer e questo, unito al fatto che TEX è Turing-complete (cfr. `http://it.wikipedia.org/wiki/Turing_equivalenza`) significa che in realtà TEX è un meta-DSL — permette cioè di definire DSL. Non solo: è possibile salvare un set di macro in formato compresso per essere rapidamente caricato in un secondo momento e questo formato è portabile (entro certi limiti: cfr. `http://www.tug.org/texinfohtml/web2c.html#Memory-dumps`) — una sorta di bytecode, quindi. Tuttavia in questo caso entrano in gioco considerazioni di efficienza, e linguaggi come il C sono a tutt'oggi le scelte preferibili, anche considerando che i compilatori ANSI C (quelli conformi allo standard ANSI X3.159-1989, successivamente ratificato in ISO/IEC 9899:1990) sono molto diffusi.

Esiste però l'altra faccia della medaglia: la complessità del linguaggio TEX può essere controproducente. Consideriamo ad esempio ancora il C, la cui grammatica è fissata e non può essere cambiata in corso di esecuzione. È quindi possibile costruire un lexer per C, integrarlo in un editor ed usarlo come evidenziatore di sintassi, ad esempio per evidenziare i commenti dal resto del codice. Praticamente tutti gli editor per sviluppatori hanno un evidenziatore di sintassi per i più comuni linguaggi di programmazione, ma per il TEX non esiste a tutt'oggi una soluzione definitiva. Ad esempio, dato che è possibile ridefinire il significato di caratteri è anche possible ridefinire % come un carattere *non* di commento, ma un lexer tradizionale non cambierebbe la sua assegnazione originale e continuerebbe a vedere % come un commento, con il risultato che il codice verrebbe evidenziato in modo sbagliato. Un evidenziatore di sintassi per il TEX in pratica coinciderebbe con il programma `etex`, back end a parte.

Un secondo aspetto critico negli SDI, e quindi in TEX, è costituito proprio dallo `switch-case`: in breve, la CPU non riesce a fare una buona previsione di dove sarà trasferito il controllo dell'esecuzione e questo con le moderne architetture pipeline ha un impatto negativo sulla performance.

La soluzione ideale sarebbe eliminare del tutto lo `switch` ed utilizzare un `goto` per saltare direttamente alla subroutine che esegue l'azione, come nel seguente pseudocodice in cui la label del `goto` è una variabile (*label as value*) e quindi può essere calcolata a tempo di esecuzione (*computed goto*), e dove gli indirizzi delle subroutine sono memorizzati nell'array `dispatch_table[]` ed il flusso dei token in `instruction[]`. Il token da eseguire è indicizzato in `instruction` dalla variabile `pc`, una sorta di program counter che può essere modificato dalle subroutine `goiftrue`, `goiffalse`...:

```
static void* dispatch_table[] = {
        ...
        &&OPR_AND,
        &&OPR_OR,
        &&OPR_CONCAT,
        &&OPR_ADD,
        ...};
#define DISPATCH() \
  goto *dispatch_table[instruction[pc++]]
int pc = 0;
while (1) {
  ...
  OPR_AND:
    goiftrue(fs, v);
    DISPATCH();
  }
  OPR_OR: {
    goiffalse(fs, v);
    DISPATCH();
  }
  OPR_CONCAT: {
    exp2nextreg(fs, v);
    DISPATCH();
  }
  OPR_ADD: {
    if (!isnumeral(v)) exp2RK(fs, v);
    DISPATCH();
  }
  ...
}/* end while */
```

Notiamo che sebbene assomigli alla `jump_table` vista precedentemente, di fatto incide sulle prestazioni perché a run time la chiamata di funzione tramite puntatore è più onerosa rispetto alla invocazione della stessa subroutine definita a tempo di compilazione tramite un computed goto. Inoltre, come detto precedentemente, le moderne CPU hanno una architettura a pipeline dove (semplificando) un modulo si occupa del fetch, un secondo del decode ed un terzo di execute dell'istruzione assembly ed è concettualmente identica ad una catena di montaggio, in quanto il primo modulo dopo aver terminato il fetch dell'istruzione corrente passa il

compito al modulo successivo ed inizia il fetch della prossima istruzione. A regime, supponendo che ogni modulo impieghi lo stesso numero di cicli di clock u, ad ogni u "esce" una istruzione completa *se la pipeline è sempre "piena"* (l'approccio naïve, quello di attendere che ogni istruzione completi il ciclo richiederebbe $3u$). L'effetto di un jump si vede solo al modulo execute, dove la CPU scopre che le istruzioni nei moduli fetch e decode non sono valide perché bisogna prelevare un'istruzione diversa e quindi bisogna svuotare la pipeline e ricominciare, operazione onerosa con conseguente perdita di performance. Dato che non è possibile evitare i jump, un modulo hardware (*branch predictor*) cerca di prevedere, partendo dall'indirizzo dell'istruzione corrente, l'indirizzo della prossima istruzione supponendo valido il *principio di località spaziale*. Questo afferma che quando si accede ad un indirizzo i è molto probabile che i successivi accessi siano vicini ad i (Hennessy e Patterson, 2012, pag. 45). Nel caso di uno `switch/case` l'indirizzo corrente è l'indirizzo della variabile dello `switch`, indirizzo *che non cambia a run time*, mentre quello di destinazione è dato dal valore della suddetta variabile *che cambia a run time* e può essere una delle n label del `case`: assumendo equiprobabilità, la probabilità di caricare l'indirizzo giusto è $1/n$ e quindi non si sfrutta il principio di località spaziale. Nel caso di un computed goto l'indirizzo corrente è l'istruzione della subroutine appena eseguita *e cambia a run time*: il successivo indirizzo, relativo alla prossima subroutine, ha maggior probabilità di essere caricato dal branch predictor proprio per il principio di località spaziale (Ertl e Gregg, 2003, pag. 14). Questo comporta anche che le istruzioni da prelevare probabilmente risiederanno nella cache, il cui tempo di accesso è dalle 20 alle 160 volte minore di quello della memoria del sistema. Purtroppo labels-as-values è una estensione C non standard: alcuni compilatori come GCC la implementano, altri no, come il compilatore di Microsoft. Al contrario i puntatori a funzioni e `switch/case` sono standard, quindi è naturale che un interprete TEX che mira alla portabilità lo preferisca ad altri costrutti anche a prezzo di una minore performance (è anche possibile riscrivere `main_control` in assembly, dove le label-as-first-class-value sono ammesse, ma è evidente che si hanno problemi di portabilità ancora maggiori).

2.3 Lua

In Ierusalimschy *et al.* (2005) il team di sviluppo dichiara gli obiettivi di Lua: linguaggio general purpose, semplicità di sintassi e di realizzazione, efficienza dell'interprete in termini di memoria e velocità di esecuzione, portabilità su diverse architetture, facilità di integrazione in progetti software esistenti.

Con riferimento alle sezioni precedenti, possiamo riassumere così le sue caratteristiche:

- Lua è un linguaggio high level general purpose (a differenza di TEX). Ha la gestione automatica delle variabili non più indicizzate (*garbage collector*), un meccanismo di coroutine per esprimere la concorrenza, e le funzioni sono assegnabili come le variabili (*function as first class value*);
- l'interprete Lua è un Bytecode Interpreter, ma il bytecode non è portabile come nel caso di Java (il suo principale obiettivo infatti è solo velocizzare il caricamento di codice precompilato e non di renderlo multipiattaforma);
- la VM è register based e per design (per esempio per non limitare la portabilità del codice C) il dispatching utilizza lo `switch/case` e non i computed goto;

Lua è progettato sia per essere estensibile sia per estendere una applicazione. Nel primo caso questo significa sia che è possibile caricare moduli scritti in Lua (caratteristica comune a tutti gli interpreti general purpose) ma soprattutto che per un programmatore C è relativamente semplice scrivere moduli in C, con l'evidente vantaggio di poter usare un linguaggio compilato. Facciamo un esempio: consideriamo la funzione "esterna" `func_sin_I` che viene invocata 10^5 volte e supponiamo che sia implementata in Lua:

```
local function Lua_func_sin_I(a)
 local y=0;
 local sin = math.sin
 for i=1,1000 do
     y=sin(y+a);
 end
 return y;
end
func_sin_I = Lua_func_sin_I
--Test code
local n=1e5
local a,jfloat=0.0,0.0
local t =os.clock()
for j=1, n  do
 jfloat=j*1.0
 a=func_sin_I(jfloat)+a
end
print("done",a,n,os.clock()-t)
```

In Linux 64 bit, con Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz, il codice impiega circa 7.9 secondi:

```
$ luatex --luaonly lua-native.lua
done 2.644855993055 100000 7.93
```

Ora consideriamo la sua equivalente in C

```
double func_sin_I(double a){
 double y=0;
 int i;
 for(i=0;i<1000;i++)
     y=sin(y+a);
 return y;
}
```

ed il relativo codice, supponendo per il momento che la funzione sia disponibile nel modulo Lua `shlib`:

```
local _c, error = package.loadlib(
          "./swiglibsh.so",
          "luaopen_shlib")
if not(_c) then
  print("Errore:",error) return 1 end
local shlib = _c()
local func_sin_I = shlib.func_sin_I
-- The test code follows
```

Il tempo impiegato è ora di circa 3 secondi:

```
$ luatex --luaonly lua-swig.lua
done 2.644855993055 100000 3.01
```

con uno speed up di $7.9/3 \approx 2.6\times$. Per caricare il codice C compilato in Lua bisogna però preparare e compilare un *wrapper* (`swiglibsh.so` nell'esempio), ossia un programma in C che segua la Application Program Interface (*API*) di Lua — in breve, le API (Ierusalimschy, 2013) sono le regole da seguire affinché l'interprete Lua sia in grado di usare una funzione che risiede in una libreria esterna non necessariamente scritta in C (e per questo chiamate *foreign functions*). È interessante notare che Lua utilizza per questo scopo un meccanismo stack based: tutte le comunicazioni tra la libreria e l'interprete avvengono tramite uno stack. Le API sono progettate proprio per facilitare questa comunicazione e non sorprende che esistano programmi in grado di automatizzare il compito di creare wrappers; uno di questi è SWIG (`http://www.swig.org`). Con SWIG si descrivono in un file interfaccia i simboli della libreria da esportare e come importarli in Lua: nel nostro caso l'interfaccia è semplicemente

```
%module shlib
%{
/* C functions exported */
extern double func_sin_I(double a);
%}
/* Lua functions */
double func_sin_I(double a);
```

Successivamente si lancia il programma che produce il codice del wrapper, lo si compila e quindi lo si collega con codice della funzione in C. Il seguente script mostra (per Linux) l'intero processo:

```
swig -lua shlib.i
O=-O2
gcc="gcc -msse4.2 "
## libreria
$gcc  $O -fPIC -I./ -o shlib.o -c shlib.c
$gcc  $O -shared -I./ shlib.o -lm -o libsh.so
## wrapper: swiglibsh.so è il modulo Lua
$gcc $O -I./ -I/usr/include/lua5.2 -fPIC \
   -o shlib_wrap.o -c shlib_wrap.c
$gcc $O  -I./ -shared shlib_wrap.o \
   -l:./libsh.so -llua5.2 -lm -o swiglibsh.so
```

È da notare come script, file interfaccia e codice C siano in tutto meno di 500 bytes. SWIG è così affidabile che raramente è necessario modificare il wrapper C prodotto, lavorando normalmente solo sul file di interfaccia ad alto livello[2].

LuaTEX è invece un esempio di una applicazione, pdfTEX in questo caso, estesa con Lua. L'idea è semplice: scelta una funzione target (come `line_break` che in pdfTEX si occupa di calcolare i breakpoints della linea) si aggiunge nel codice sorgente una funzione (*callback*, come `lua_linebreak_callback` in questo caso) che chiama l'interprete Lua e riceve il risultato:

```
void line_break(boolean d,
                int line_break_context){
...
callback_id =
 callback_defined(linebreak_filter_callback);
if (callback_id > 0) {
 callback_id =
  lua_linebreak_callback(d, temp_head,
             addressof(cur_list.tail_field));
...
}
```

il callback viene invocato con opportuni parametri ed è il punto in cui si interagisce con lo stack:

```
int
lua_linebreak_callback(int is_broken,
    halfword head_node, halfword * new_head){
int a;
register halfword *p;
int ret = 0;                /* failure */
lua_State *L = Luas;
int s_top = lua_gettop(L);
int callback_id =
  callback_defined(linebreak_filter_callback);
if (head_node == null ||
    vlink(head_node) == null ||
     callback_id <= 0) {
    lua_settop(L, s_top);
    return ret;
}
if (!get_callback(L, callback_id)) {
   lua_settop(L, s_top);
    return ret;
}
nodelist_to_lua(L, vlink(head_node));
lua_pushboolean(L, is_broken);
if (lua_pcall(L, 2, 1, 0) != 0) {
    fprintf(stdout, "error: %s\n",
             lua_tostring(L, -1));
    lua_settop(L, s_top);
    error();
    return ret;
```

2. Il progetto SwigLib (`http://www.luatex.org/swiglib.html`) intende esplorare proprio l'area di sviluppo di moduli C per LuaTEX con SWIG. Alcuni sono già disponibili presso `https://foundry.supelec.fr/scm/viewvc.php/trunk/?root=swiglib`. SwigLib è in parte sponsorizzato dal TEX User Group, cfr. `http://tug.org/tc/devfund/grants.html`

```
}
p = lua_touserdata(L, -1);
if (p != NULL) {
    a = nodelist_from_lua(L);
    try_couple_nodes(*new_head,a);
    ret = 1;
}
lua_settop(L, s_top);
return ret;
}
```

Con `get_callback(L, callback_id)` si pone nello stack la funzione Lua identificata da `callback_id` (`linebreak_filter` in questo caso); con `nodelist_to_lua(L, vlink(head_node))` e `lua_pushboolean(L, is_broken)` si aggiungono due argomenti, sempre nello stack, ed infine con `lua_pcall` si chiama la funzione; `p = lua_touserdata(L, -1)` è il risultato, e se è non nullo viene convertito in una lista di nodi e passato all'interprete TEX tramite `try_couple_nodes(*new_head,a)`. In questo caso è possibile quindi rimpiazzare l'algoritmo `line_break` con uno proprio, senza dover ricompilare il codice. LUATEX ha qualche centinaio di callbacks (oltre ad una versione embedded di METAPOST) — TEX è un programma piuttosto complicato.

Infine un ultimo cenno ad un'altra caratteristica di Lua che è molto importante per il TEX: l'uso di *self-describing data format* che sono costrutti Lua. In questo modo è più facile adattare Lua al linguaggio che si vuole estendere: per esempio ecco come potrebbe apparire la versione Lua del record BibTEX visto precedentemente:

```
current_key=nil
current_type=nil
bibtex_db={}
function insert(...)
 local _t=bibtex_db[current_type] or {}
 _t[current_key] = (...)
 bibtex_db[current_type]=_t
end
article=function(key)
  current_type,current_key='article',key
  return insert
end
ARTICLE,Article=article,article
ARTICLE "Ertl03thestructure" {
    author = "M. Anton Ertl and David Gregg",
    title = "The Structure and Performance of
             Efficient Interpreters",
    journal = "Journal of Instruction-Level
               Parallelism",
    year = "2003",
    volume = "5",
    pages = "2003"
}
print(bibtex_db.
      article["Ertl03thestructure"].author)
print(bibtex_db.
      article["Ertl03thestructure"].title)
```

`ARTICLE "Ertl03thestructure"` è una chiamata alla funzione `ARTICLE(key)` (alias di `article(key)`) la quale ritorna a sua volta la funzione `insert(...)` che viene chiamata con la tabella `{author=...,pages="2003"}`. Come si vede la somiglianza con il record di BibTEX è notevole.

Notiamo che quanto detto fino a qui si applica direttamente a LUATEX, anche se attualmente non è presente un REPL Lua nativo: per eseguire script Lua a riga di comando bisogna usare la sintassi `$ luatex --luaonly <scritp-lua>`, mentre con `\directlua <16-bit number> <general text>` si possono eseguire script Lua da TEX.

2.4 LuaJIT

Si è visto precedentemente come dalla coppia Domain Specific Language/Direct Syntax Interpreter di TEX si sia passati alla coppia general purpose/-Bytecode Interpreter di Lua: vediamo ora come LuaJIT si inserisce in questa linea evolutiva.
LuaJIT è un interprete Lua 5.1, meno focalizzato sulla portabilità e più sulla velocità e sulla memoria. Vi sono tre importanti novità da evidenziare:

1. la VM è in assembly: il codice è minore, implementa la tecnica dei computed goto vista in precedenza per cui ha migliori performance in termini di occupazione di cache e branch prediction ed è più veloce di una VM in C;

2. LuaJIT è un JIT-Interpreter *tracing just-in-time*, simile a HotSpot™ di Oracle; diverse funzioni delle librerie standard sono compilate just in time, ma un buon numero ancora no (*NYI*, Not Yet Implemented cfr. `http://wiki.luajit.org/NYI`);

3. Il modulo `ffi` è integrato nella VM (*foreign function interface*) e offre funzionalità simili a SWIG, ma *evitando* il wrapper intermedio.

Non è difficile vedere le performance di `func_sin_I` in LuaJIT utilizzando `ffi`, la funzione `math.sin` (jit-compiled) ed infine il wrapper SWIG, il cui codice sorgente è valido ma deve essere collegato con la libreria `libjit` e non `liblua`:

```
-- ----------------------------------------
-- test.lua
-- ----------------------------------------
--
-- ffi interface
local ffi = require("ffi")
ffi.cdef[[
double func_sin_I(double a);
]]
local shlib = ffi.load("./libsh.so")
--
-- SWIG interface
local _c, error =
  package.loadlib("./swiglibsh_jit.so",
                  "luaopen_shlib")
if not(_c) then print("Errore:" ,error)
```

```
    return 1 end
local SWIG_shlib = _c()
--
-- LuaJIT native
local function Lua_func_sin_I(a)
 local y=0;
 for i=1,1000 do
     y=math.sin(y+a);
 end
 return y;
end
local f1 = shlib.func_sin_I
local Lua_f1 = Lua_func_sin_I
local SWIG_f1 = SWIG_shlib.func_sin_I
--
--Test code
local n=1e5
local a=0.0
local jfloat = ffi.new('double',0)
local t=os.clock()
for j=1, n  do
 jfloat=j*1.0
 a=shlib.func_sin_I(jfloat)+a
end
print("jit/ffi",a,n,os.clock()-t)
local t=os.clock()
a=0;jfloat=0
for j=1, n  do
 jfloat=j*1.0
 a=Lua_f1(jfloat)+a
end
print("jit/native",a,n,os.clock()-t)
local t=os.clock()
a=0;jfloat=0
for j=1, n  do
 jfloat=j*1.0
 a=SWIG_f1(jfloat)+a
end
print("jit/SWIG",a,n,os.clock()-t)
```

Eseguendo il programma con
`$>luajittex --jiton --luaonly test.lua`
abbiamo:

```
jit/ffi    2.644855993055 100000 2.98
jit/native 2.6448559930926 100000 3.69
jit/SWIG   2.644855993055 100000 2.99
```

Confrontata con la versione nativa precedente lo speed up ora è di $7.9/3.7 \approx 2.1\times$. È interessante notare come in questo caso le prestazioni si equivalgano se si delega al C la funzione `func_sin_I`: tuttavia in `http://luajit.org/ext_ffi.html` è mostrato un esempio dove la sostituzione di una `table` di Lua con un `struct` in C ha come effetto uno speed up del $110\times$ e una riduzione di memoria occupata di 64 volte (su processore `x86_64`).

Non è difficile ora capire cos'è LuajitTEX: è LuaTEX dove Lua è stato sostituito con LuaJIT, cosa possibile grazie al fatto che le API sono sostanzialmente invariate. A parte il nome `luajitex`, l'invocazione dell'interprete Lua a riga di comando o da TEX è analoga a quanto visto per LuaTEX.

3 Primi risultati

LuajitTEX è un progetto relativamente giovane: la prima release ufficiale è di fine Dicembre 2012 e per il momento l'obiettivo principale è verificare la corretta implementazione e misurare speed-up, rimanendo in sincronia con le release di LuaTEX e LuaJIT. Dato che ConTEXt-MKIV attualmente ha il più esteso codice base in Lua, alcuni test sono stati condotti da H. Hagen (Hagen, 2013) ed i risultati si possono così riassumere:

1. la VM in assembly introduce in media uno speed up $2\times$;
2. il compilatore JIT tende a degradare leggermente le prestazioni.

Questi risultati sono coerenti con quanto visto precedentemente, ma sono relativi *solo* al codice Lua mentre ConTEXt (LuaTEX) usa anche l'interprete TEX che non è influenzato da LuaJIT: i test condotti mostrano che il codice TEX occupa una quantità dal 30% al 70% del tempo totale per cui lo speed up complessivo varia da $1.15\times$ a $1.35\times$ — in media cioè si ha un guadagno del 25% del tempo, comunque significativo. È ancora in corso di studio invece il degrado dovuto al JIT, che per questo motivo è disattivato per default: la spiegazione potrebbe essere l'utilizzo delle funzioni NYI e delle chiamate C verso Lua che possono interferire con il tracing del bytecode, impedendone la compilazione.
Infine il consumo di memoria non mostra apprezzabili differenze.

4 Conclusioni

È interessante notare come LuajitTEX riassuma l'evoluzione dell'interprete, partendo dal più semplice (TEX) per arrivare all'implementazione odierna dove sfuma il confine tra compilatore ed interprete.

L'incremento di performance di LuajitTEX non è trascurabile e, da questo punto di vista, lo qualifica come una valida alternativa a LuaTEX, ma la performance non è l'unico parametro e forse neanche il più importante. Un nuovo interprete TEX deve essere disponibile per le stesse piattaforme di LuaTEX, ma in questo caso si tratta di un difficile obiettivo per LuajitTEX dato che la VM è implementata in codice assembly per piattaforme specifiche. Inoltre anche la scelta del compilatore è vincolante: è richiesto almeno `gcc 3.4` ed attualmente non è il compilatore usato per la TEX Live. Per il momento l'eseguibile è disponibile per diverse piattaforme Unix/Mac presso `http://svn.contextgarden.net/suite-bin/tex/` (grazie a Mojca Miklavec): una versione per windows 32 bit si trova presso `http://w32tex.org/` (grazie al prof. A. Kakuto).
Tuttavia il compilatore non è il solo problema: il team di sviluppo di Lua e quello di LuaJIT (essenzialmente una persona) sono separati e non

condividono una comune *roadmap*. Ad esempio la versione di luaTEX per la TEX Live 2013 si baserà su Lua 5.2.1, ma LuaJIT si basa su Lua 5.1 con una integrazione per Lua 5.2 non a livello di API, ma solo di linguaggio, e non pare ci sia l'intenzione a breve di portarsi ad una piena compatibilità con Lua 5.2. Quindi allo stato attuale Lua appare un progetto più solido se non altro perché LuaJIT è un *one-man* project, ma non si deve trascurare che la prima release è del Settembre 2005 e che con la versione 2.0 la comunità è notevolmente cresciuta. Un ambito in cui LuajitTEX può competere con luaTEX è il *data publishing* in cui vi siano requisiti sia di velocità che di interfacciamento verso altri sistemi (ad esempio DBMS o server WEB) ed in questo contesto il modulo `ffi` è una funzionalità interessante, soprattutto se è possibile sfruttare il self-describing data con il compilatore jit evitando le funzioni NYI. Anche in questo caso, però, la differenza tra la complessità di un'interfaccia `ffi.cdef` di LuaJIT e la medesima in SWIG per standard Lua non è così rilevante — e SWIG è un pre-processore C più completo e può produrre wrappers per altri linguaggi oltre a Lua. Nel caso, è eventualmente possibile scrive un modulo SWIG che emuli `ffi`, i.e. un wrapper alla libreria libffi (cfr. `https://foundry.supelec.fr/scm/viewvc.php/trunk/experimental/libffi/?root=swiglib`). Si tratta in sostanza di una caratteristica affascinante sì, ma che deve essere valutata in base al progetto.

Riferimenti bibliografici

Ertl, M. A. e Gregg, D. (2003). «The structure and performance of efficient interpreters». *Journal of Instruction-Level Parallelism*, **5**, p. 2003.

Hagen, H. (2013). «ConTEXt: Just in Time LuaTEX». *TUGboat*, **34** (1), pp. 72–78.

Hennessy, J. L. e Patterson, D. A. (2012). *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann, 5ª edizione.

Ierusalimschy, R. (2013). *Programming in Lua, Third Edition*. Lua.org.

Ierusalimschy, R., de Figueiredo, L. H. e Celes, W. (2005). «The implementation of lua 5.0». *Journal of Universal Computer Science*, **11** (7), pp. 1159–1176. `http://www.jucs.org/jucs_11_7/the_implementation_of_lua`.

— (2007). «The evolution of lua». In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, New York, NY, USA, HOPL III, pp. 2–1–2–26. URL `http://doi.acm.org/10.1145/1238844.1238846`.

Levine, J. R. (2009). *flex and bison - Unix text processing tools*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Levine, J. R., Mason, T. e Brown, D. (1992). *lex & yacc (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Parr, T. (2009). *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1ª edizione.

Shi, Y., Casey, K., Ertl, M. A. e Gregg, D. (2008). «Virtual machine showdown: Stack versus registers». *ACM Trans. Archit. Code Optim.*, **4** (4), pp. 2:1–2:36. URL `http://doi.acm.org/10.1145/1328195.1328197`.

▷ Luigi Scarso
`luigi dot scarso at gmail dot com`