ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

II facoltà di Ingegneria - Cesena
Ingegneria Informatica

speed-2p: analisi prestazionale e profiling dell'engine tuProlog

Linguaggi e modelli computazionali L-S

Michael Gattavecchia
0000362269

prof. Mirko Viroli

ANNO ACCADEMICO 2009/10

1 Introduzione

Il progetto ha l'obiettivo di valutare le principali caratteristiche prestazionali dell'attuale implementazione di tuProlog e di proporre eventuali interventi mirati all'aumento delle stesse, sia in modo diretto, ovvero relativamente al carico computazionale, sia in modo indiretto, andando ad ottimizzare la quantità di heap allocata in modo da ridurre al minimo il fisiologico intervendo del garbage collector presente nell'environment Java.

Per effettuare una valutazione quantitativa delle caratteristiche prestazionali del sistema, oltre all'usuale utilizzo di benchmark, si è ricorsi ad uno strumento di profiling attraverso il quale è stato possibile valutare l'impatto delle varie attività svolte durante la risoluzione di un goal in modo autonomo e puntuale.

Basi di partenza per l'analisi svolta sono state la tesi di laurea *"Analisi di prestazione dell'interprete tuprolog su piattaforma Java"*[Dam08] di Michele Damian, ed il materiale[Con10] (sorgenti e relazione tecnica) prodotto da Paolo Contessi nell'ambito dell'assignment *"fast-match"* svolto per il corso di Linguaggi e Modelli Computazionali L-M. Ulteriore supporto è stato tratto dal codice sorgente e dalla documentazione allegata al progetto tuProlog[1] [2p-d].

2 Profilng tool

2.1 NetBeans Profiler

Lo strumento utilizzato per l'analisi del progetto è NetBeans Profiler[2], si tratta di un profiler integrato nell'IDE NetBeans in grado di presentare resoconti particolarmente strutturati sull'efficienza del codice, senza comportare alcun onere aggiuntivo da parte dell'analista.

Oltre all'utilizzo di quest'ultimo sono state sviluppate alcune semplici entità a supporto della valutazione.

Nello specifico si tratta di un contatore statico a più indici (`StaticCounter`) attraverso il quale è stato particolarmente agevole, tra le altre cose, valutare la ripartizione delle iterazioni all'interno di costrutti if-then, in modo da poter comprendere a quali parti di codice dedicare maggiore attenzione. Altro utile strumento è stato un cronometro (`StaticStopwatch`), anch'esso realizzato attraverso metodi statici, grazie al quale si è misurato il tempo necessario all'esecuzione di specifiche porzioni di codice.

Unitamente a ciò si è implementato un main program all'interno della classe `RunEngine` in grado di svolgere la risoluzione di un goal (dopo aver settato

[1] http://tuprolog.sourceforge.net
[2] http://profiler.netbeans.org

una specifica teoria) per un numero statisticamente rilevante di volte, in modo da poter formulare considerazioni più attendibili riguardo i gap prestazionali osservati.

Sia durante l'esecuzione, sia al suo termine, il tool di profiling permette di apprezzare la ripartizione del carico tra i vari componenti del progetto.

Sono poi disponibili due ulteriori tipologie di analisi: la prima orientata alla valutazione del consumo della memoria, la seconda allo studio delle dinamiche derivanti da applicazioni multi-threaded. La natura del progetto ha portato a trascurare quest'ultima vista in favore delle prime due.

3 Analisi

Verranno di seguito poste a confrono tre implementazioni di tuProlog al fine di poter poi formulare alcune considerazioni relativamente alle prestazioni ed ai possibili interventi orientati a migliorarle. Le versioni considerate saranno:

- tuProlog 2.1 [3]
- tuProlog 2.3 alpha
- tuProlog 2.1 (fast-match)

3.1 Benchmark

Per valutare l'efficienza delle varie implementazioni sono stati svolti alcuni benchmark classici nell'ambito Prolog. Per ognuna delle prove sono state svolte più ripetizioni dell'invocazione del metodo `solve()` sull'istanza della classe `Prolog`, in modo da rendere il più trascurabili possibile le piccole variazioni all'interno delle singole esecuzioni dovute allo stato puntuale del calcolatore.

La macchina sulla quale sono stati svolti i test è dotata di processore dual core AMD Athlon 64 X2 5400+ e di 4 GB RAM DDR2 800 MHz (OS: Ubuntu GNU/Linux 64bit v10.10).

Il test riferito come "Asserting(25)" in tabella 1 rappresenta l'esecuzione di un test all'interno del quale viene fatto un uso intensivo dell'asserzione: tale test è stato costruito ex-novo per poter valutare le caratteristiche delle varie versioni relativamente all'aggiunta di nuovi fatti alla teoria. Il sorgente del test si presenta come segue:

```
asserter(X):-asserter(X,X,X).
asserter(X,Y,N):-X>0, Y>0, !,
```

[3] Sulla versione 2.1 utilizzata sono stati apportati i cambiamenti suggeriti in [Dam08], al fine di evitare la saturazione di heap e/o stack.

```
                asrt(p(X,Y)),
                Y1 is Y-1,
                asserter(X,Y1,N).
asserter(X,0,N):-X>0,
                asrt(p(X,0)),
                X1 is X-1, !,
                asserter(X1,N,N).
asserter(0,Y,N):-Y>0,
                asrt(p(0,Y)), !,
                Y1 is Y-1,
                asserter(0,Y1,N).
asserter(0,0,_):- asrt(p(0,0)).

% asserta/assertz
asrt(X):-asserta(X).
```

mentre il goal risolto è:

```
?-asserter(25).
```

che asserisce nel complesso 676 fatti del tipo `p(X,Y)`, dove `X` e `Y` sono appunto valori numerici compresi tra 0 e 25, estremi inclusi. La sostituzione di `asserta(...)` con `assertz(...)` ha prodotto valori del tutto analoghi. Per l'esecuzione di questo test si è provveduto a rigenerare nuove instanze di `Prolog` e `Theory` ad ogni iterazione in modo da mantenere il controllo sulla dimensione della teoria complessivamente asserita.

Il benchmark "LMC-sept-2010" è invece relativo al tema d'esame del corso di LMC svoltosi nel settembre 2010. In questo caso la teoria

```
program(P):-P=[dec(10,i2,20,quit),
               dec(20,i1,21,30),
               inc(21,o1,22),
               inc(22,t1,20),
               dec(30,t1,31,10),
               inc(31,i1,30)].
engine(_,quit,I,I).
engine(P,ID,I,O):-member(inc(ID,R,Next),P),!,
                  member(val(R,V),I,I2),!,
                  V2 is V+1,
                  engine(P,Next,[val(R,V2)|I2],O).

engine(P,ID,I,O):-member(dec(ID,R,Next1,Next2),P),!,
                  member(val(R,N),I,I2),!,(N=0,!,
```

Benchmark	iterations	2.1 [ms]	2.3(alpha) [ms]	2.1 (fast-match) [ms]
crypt	1000	96.3	95.3	76.1
qsort	10000	9.6	9.3	8.0
queens	100	8010	7836	7421
query	1000	59.3	62.8	21.7
tak	100	1588	1600	1521
asserter(25)	1000	45	46	42
LMC-sept-2010	100	137	138	133

Tabella 1: Tempi di esecuzione delle tre versioni a confronto

```
                engine(P,Next2,I,O);N2 is N-1,
                engine(P,Next1,[val(R,N2)|I2],O)).
member(H,[H|T],T).
member(H,[H2|T],[H2|T2]):-member(H,T,T2).
```

ed il goal

```
?-engine(P, 10, [val(i1,10),val(i2,10),val(t1,0),val(o1,0)], O).
```

comportano il calcolo del prodotto $10 * 10$ attraverso l'utilizzo di un interprete del modello *Abacus* (si rimanda al testo del'appello[4] per maggiori dettagli). Come possibile notare analizzando la teoria in questo caso viene fatto frequente ricorso all'utilizzo di liste (member/2) e ricorsione.

Analizzando i dati riportati in tabella 1 appare chiaro come il meccanismo di indicizzazione introdotto da Contessi comporti nella totalità dei casi un apprezzabile miglioramento delle prestazioni del motore Prolog. Da notare come anche nel caso in cui si risolvano goal che richiedano un numero notevole di asserzioni, l'indicizzazione non ha mai alcuna rilevanza, se paragonata alle implementazioni classiche di `ClauseDatabase`.

Volendo forzare l'analisi dell'overhead dovuto all'indicizzazione si è realizzata un'ulteriore teoria che a differenza della teoria "asserter" mostrata in precedenza si limita ad iterare istruzioni `asserta(...)` senza ulteriori computazioni. La teoria realizzata si presenta nel seguente modo:

```
asserter2(0).
asserter2(X):- X>0,
               asserta(p(X)),
               X1 is X-1,
               asserter2(X1).
```

[4] http://campus.cib.unibo.it/28535/

Benchmark	iterations	2.1 [ms]	2.3(alpha) [ms]	2.1 (fast-match) [ms]
asserter2(1000)	1000	36.9	37.5	57.2
asserter2(2500)	1000	80.0	81.8	251.3
asserter2(5000)	1000	159.8	150.1	884
asserter2(7500)	1000	230.6	233.4	2033

Tabella 2: Tempi di esecuzione in caso di pura asserzione

ed il goal imposto è stato:

```
?-asserter2(N).
```

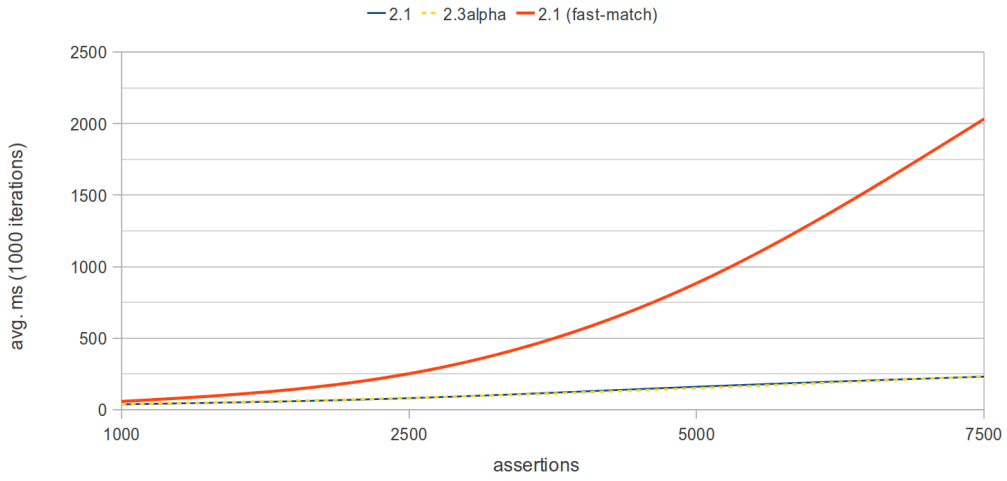

Figura 1: Tempi medi di calcolo (su 1000 iterazioni) in funzione del numero di asserzioni

L'esito di tale esecuzione ha mostrato come, nella pura asserzione, l'indicizzazione comporti un notevole peggioramento complessivo dovuto all'overhead causato dalla gestione degli indici, ancora peggiore (seppur prevedibile) è lo scaling in funzione del numero di asserzioni.

3.1.1 Considerazioni

Anche tenendo conto del fatto che il tempo di consultazione iniziale della teoria cresce di circa il 9% con l'introduzione dell'indicizzazione [Con10] e della maggiorazione computazionale conseguente ad asserzioni estremamente numerose, è comunque necessario tenere conto del fatto che i tempi di risoluzione di un

goal sono nella totalità dei casi analizzati inferiori rispetto alle implementazioni prive di indexing.

Un esempio particolarmente utile alla comprensione delle forze in gioco è il benchmark "asserting(25)" mostrato in tabella 1, relativamente al quale è possibile notare come anche asserendo discrete quantità di nuovi fatti, l'indicizzazione compensi più che sufficientemente il fisiologico overhead, arrivando ad offrire prestazioni addirittura migliori.

Ovviamente con esecuzioni massivamente assertive l'overhead emerge in modo netto, determinando un preoccupante divergenza dei tempi medi di esecuzione: proprio su tale aspetto si è focalizzata l'attenzione, come esposto nella sezione successiva.

4 Proposte d'intervento

4.1 Indexing

A seguito dei dati presentati in figura 1 si è svolta una (lunga) sessione di profiling relativa all'esecuzione di 7500 fatti attraverso la risoluzione del goal `?-asserter2(7500).`

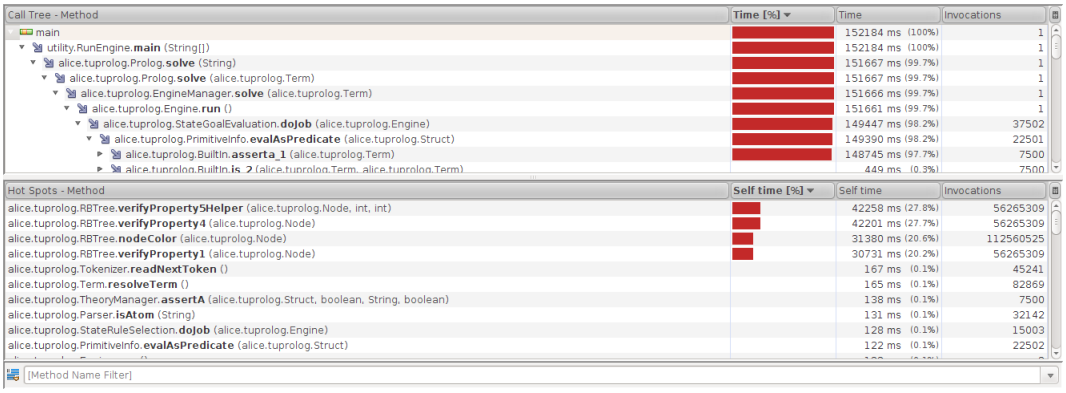

Figura 2: Esiti del profiling della sessione di puro asserting

come è possibile notare in figura 2, i principali responsabili del decadimento prestazionale sono alcuni metodi della classe `RBTree` : in particolare tre metodi di verifica delle proprietà dell'albero rappresentato dalla classe, oltre ad un quarto ad essi collegato.

Benchmark	iterations	2.1 [ms]	2.3(alpha) [ms]	2.1fm (no_check) [ms]
asserter2(1000)	1000	36.9	37.5	35.1
asserter2(2500)	1000	80.0	81.8	75.2
asserter2(5000)	1000	159.8	150.1	148.0
asserter2(7500)	1000	230.6	233.4	217.5
asserter2(10000)	1000	306.9	312.4	299.5

Tabella 3: Tempi di esecuzione in caso di pura asserzione

Analizzando meglio il codice introdotto [5] si è potuto notare come gli sviluppatori abbiano introdotto un meccanismo di valutazione del bilanciamento dell'albero all'interno della classe `RBTree`.

Come da essi specificato tale meccanismo, introdotto a fini di debug e basato sull'asserting Java, risulta particolarmente oneroso in termini computazionali: tuttavia è possibile bypassarlo attraverso il setting a `false` del flag `VERIFY_RBTREE`, producendo così un significativo incremento prestazionale.

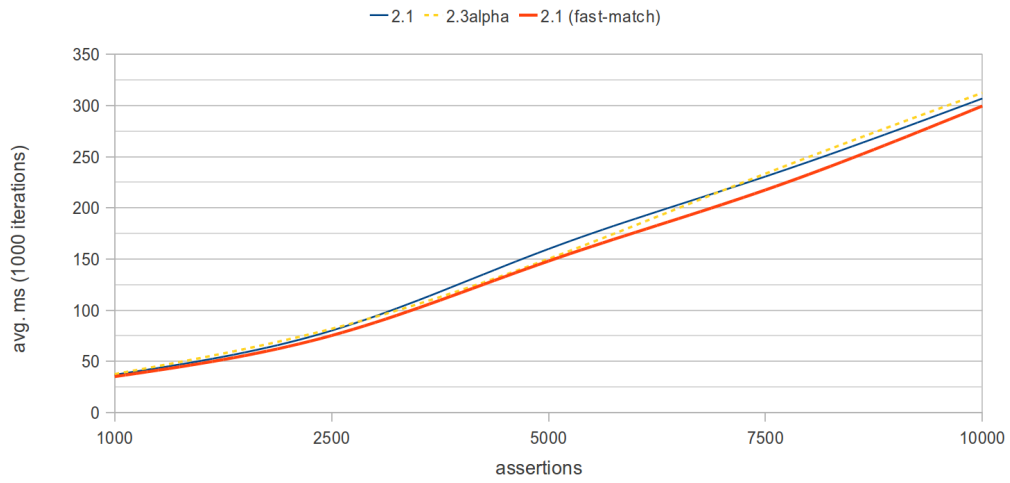

Figura 3: Tempi medi di calcolo (su 1000 iterazioni) in funzione del numero di asserzioni

Per valutare tale incremento sono stati nuovamente eseguiti i test presentati in tabella 2, ma solo dopo essersi assicurati del buon esito del testing JUnit del progetto così modificato. A tal proposito è stato necessario apportare alcune modifiche al codice prodotto da Paolo Contessi [Con10], che a causa di

[5] da Paolo Contessi, e reperibile all'indirizzo
http://en.literateprograms.org/Red-black_tree_(Java)?oldid=16622

un errato utilizzo di un iteratore non permetteva la corretta esecuzione della totalità dei test.

La reiterazione dei test di asserzione ha mostrato un incremento prestazionale particolarmente rilevante, portanto la versione caratterizzata dall'indexing a presentare prestazioni addirittura migliori rispetto alle implementazioni classiche (figura 3).

4.2 Classe `Struct`

Continuando nel profiling della versione "fast-match" è emerso un frequente ricorso al metodo `getTerm()`, che nella quasi totalità dei casi viene invocato su istanze della classe `Term`. Analizzando meglio la distribuzione dei carichi si è poi notato come anche il metodo `Term getTerm(int)` della classe `Struct` fosse molto spesso invocato. Andando ad analizzare il codice di quest'ultimo:

```
public Term getTerm(int index) {
    return arg[index].getTerm();
}
```

dove la variabile private `arg` rappresenta una array di `Term`, si è avuta conferma del perchè si avesse un così frequente ricorso al metodo `getTerm()` della classe `Term`. Tale metodo è però definito come astratto, e pertanto è stato necessario andare ad analizzare la gerarchia all'interno della quale esso era posto (figura 4).

Una variabile di tipo `Term` può quindi in realtà essere istanza delle classi `Struct`, `Var` o `Number` [6] , all'interno delle quali il metodo `getTerm()` è rispettivamente ridefinito.

Ciò che ha incuriosito in fase di analisi è stato il fatto che, sia all'interno della classe `Number`, sia all'interno della classe `Struct`, tale metodo fosse implementato nel seguente modo:

```
public Term getTerm() {
    return this;
}
```

mentre all'interno della classe `Var` vengono svolti alcuni controlli specifici prima di ritornare il `Term` richiesto.

A questo punto, visto il consistente ricorso al metodo `getTerm(int)` della classe `Struct`, si è pensato di sfruttare l'equivalenta dell'implementazione

[6] la classe `Number` è a sua volta specificizzata nelle classi `Double`, `Float`, `Int` e `Long`, nelle quali però il metodo `getTerm()` non è ulteriormente ridefinito.

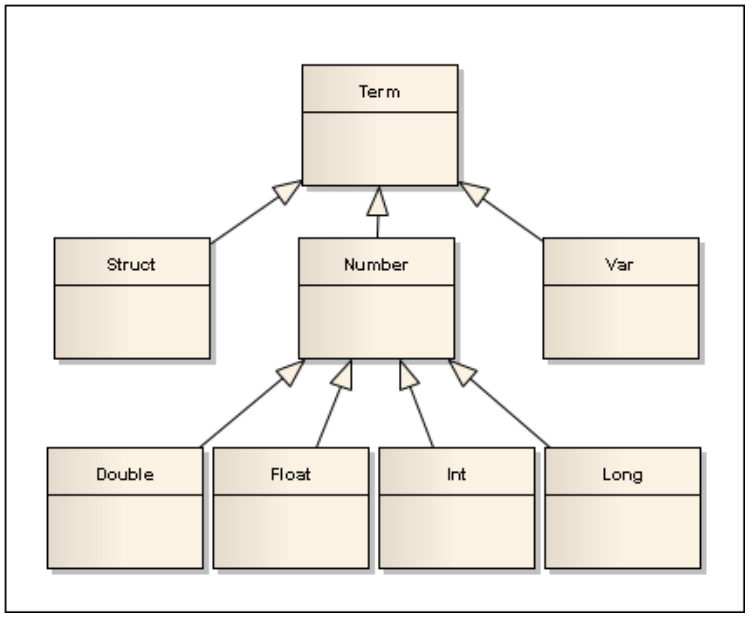

Figura 4: Class diagram relativo alla gerarchia al di sotto della classe `Term`

di `getTerm()` nelle prime due classi per evitare che ad ogni invocazione di quest'ultimo si innescasse il meccanismo del polimorfismo per andare a determinare la tipologia di istanza riferita dalla variabile di tipo `Term` per definire quale porzione di codice eseguire.

Si è quindi inserito un controllo interno al metodo `getTerm(int)` (classe `Struct`) a monte della restituzione del valore di ritorno, discriminando quì instanze della classe `Var` rispetto alle altre possibili:

```
public Term getTerm(int index) {
    if (!(arg[index] instanceof Var))
        return arg[index];
    return arg[index].getTerm();
}
```

in questo modo in tutti i casi in cui all'interno dell'elemento riferito del vettore `arg` sono presenti instanze delle classi `Struct` o `Number` (o di una delle sue estensioni) si evita di andare a effettuare una chiamata che, in sostanza, non farebbe altro che restituire esattamente l'elemento del vettore, dopo un casting a `Term`.

Benchmark	iterations	avg time (old) [ms]	avg time (new) [ms]	improvement
crypt	1000	76.11	74.15	2.6 %
qsort	10000	10.33	9.31	9.9 %
queens	100	7369	7276	1.3 %
query	1000	21.12	21.00	0.57 %
tak	100	1624	1610	0.87 %

Tabella 4: Tempi di esecuzione delle due implementazioni a confronto

4.2.1 Confronto tra le due implementazioni

Il semplice accorgimento adottato ha prodotto risultati apprezzabili dal punto di vista prestazionale. Al fine di valutarne quantitativamente l'impatto sono stati svolti una serie di benchmark [Sics] particolarmente noti.

Per ogni prova sono state svolte più ripetizioni dell'invocazione del metodo `solve()` sull'istanza della classe `Prolog`, in modo da rendere il più trascurabili possibile le piccole variazioni all'interno delle varie esecuzioni dovute allo stato puntuale del calcolatore. Anche in questo caso la macchina sulla quale sono stati svolti i test è dotata di processore dual core AMD Athlon 64 X2 5400+ e di 4 GB RAM DDR2 800 MHz (OS: Ubuntu GNU/Linux 64bit v10.10).

Come si può notare in figura 5 il numero di invocazioni del metodo `getTerm()` (classe `Struct`) per la medesima risoluzione si riduce sensibilmente (nell'esempio di circa il 35%), a tutto vantaggio dei tempi di esecuzione.

In tabella 4 vengono presentati gli esiti dell'esecuzione di alcuni benchmark notevoli per il linguaggio prolog: nella colonna 'iterations' viene riportato il numero di invocazioni consecutive del metodo `solve()`, tale numero è stato tarato in funzione della durata media dei benchmark per evitare di avviare simulazioni eccessivamente prolungate nel tempo.

Come si può notare si sono ottenuti risultati apprezzabili, con una prestazione particolarmente rilevante nel benchmark 'qsort' (-10%). Ovviamente i benefici variano in funzione del codice specifico, ma è importante notare come sia stato possibile ottenere un miglioramento medio dei tempi di esecuzione dell'ordine del 3%.

È da notare come l'accorgimento adottato all'interno della classe `Struct` non fosse presente anche nelle versioni 2.1 e 2.3alpha: tralasciando per un attimo l'indexing, la modifica del metodo `getTerm(int)` potrebbe sicuramente portare migioramenti prestazionali dello stesso ordine all'interno delle varie versioni preesistenti.

Hot Spots - Method	Self time [%]	Self time	Invocations
alice.tuprolog.Var.**occurCheck** (java.util.List...		71.6 ms (2.4%)	24918
alice.tuprolog.Struct.**unify** (java.util.List, jav...		62.4 ms (2.1%)	5841
alice.tuprolog.Struct.**getTerm** (int)		51.1 ms (1.7%)	46102
alice.tuprolog.Struct.**isList** ()		32.3 ms (1.1%)	10308
alice.tuprolog.Struct.**toString** ()		27.9 ms (0.9%)	1815
alice.tuprolog.Struct.**<init>** (String, java.util...		26.4 ms (0.9%)	1120
alice.tuprolog.Struct.**getTerm** ()		23.8 ms (0.8%)	22849
alice.tuprolog.Struct.**resolveTerm** (java.util...		16.6 ms (0.6%)	2158
alice.tuprolog.Struct.**copy** (java.util.Abstrac...		14.0 ms (0.5%)	3050
alice.tuprolog.Struct.**resolveTerm** (long)		11.1 ms (0.4%)	3714
alice.tuprolog.Struct.**<init>** (String, int)		8.69 ms (0.3%)	1090
alice.tuprolog.TheoryManager.**assertZ** (alic...		7.14 ms (0.2%)	150
alice.tuprolog.Struct.**copy** (java.util.Abstrac...		6.70 ms (0.2%)	156
alice.tuprolog.Struct.**getArg** (int)		5.10 ms (0.2%)	3788
alice.tuprolog.Struct.**<init>** (int)		3.34 ms (0.1%)	3206

Hot Spots - Method	Self time [%]	Self time	Invocations
alice.tuprolog.Var.**occurCheck** (java.util.List...		92.3 ms (3.3%)	24918
alice.tuprolog.Struct.**toString** ()		44.5 ms (1.6%)	1815
alice.tuprolog.Struct.**unify** (java.util.List, jav...		41.8 ms (1.5%)	5841
alice.tuprolog.Struct.**getTerm** (int)		32.8 ms (1.2%)	46102
alice.tuprolog.Struct.**resolveTerm** (long)		17.5 ms (0.6%)	3714
alice.tuprolog.Struct.**<init>** (int)		15.1 ms (0.5%)	3206
alice.tuprolog.Struct.**resolveTerm** (java.util...		14.6 ms (0.5%)	2158
alice.tuprolog.Struct.**isPrimitive** ()		12.9 ms (0.5%)	1155
alice.tuprolog.Struct.**<init>** (String, java.util...		11.4 ms (0.4%)	1120
alice.tuprolog.Struct.**<init>** (String, alice.tu...		8.66 ms (0.3%)	438
alice.tuprolog.TheoryManager.**assertZ** (alic...		8.32 ms (0.3%)	150
alice.tuprolog.Struct.**getTerm** ()		6.65 ms (0.2%)	14857
alice.tuprolog.Struct.**isEmptyList** ()		6.5 ms (0.2%)	3013
alice.tuprolog.Struct.**copy** (java.util.Abstrac...		5.87 ms (0.2%)	3050
alice.tuprolog.Struct.**isList** ()		5.38 ms (0.2%)	10308

Figura 5: Hot Spot view relativa all'esecuzione del benchmark 'qsort' con `Struct` originale (sopra) e nuovo (sotto)

4.3 Profiling dopo l'applicazione delle modifiche

Dopo aver messo in pratica le modifiche proposte all'interno del fork "fast-match" di tuProlog si è voluta compiere un'ulteriore fase di profiling orientata all'individuazione di ulteriori aspetti migliorabili dal punto di vista prestazionale.

Hot Spots - Method	Self time [%]	Self time	Invocations
alice.tuprolog.Var.**unify** (java.util.List, java.util.List, alic...		19034 ms (7.5%)	11209290
alice.tuprolog.Var.**getTerm** ()		17038 ms (6.7%)	35789565
alice.tuprolog.Struct.**unify** (java.util.List, java.util.List, al...		11576 ms (4.6%)	6424184
alice.tuprolog.Term.**resolveTerm** ()		11395 ms (4.5%)	4874096
alice.tuprolog.Struct.**getTerm** (int)		10970 ms (4.3%)	11137719
alice.tuprolog.Var.**occurCheck** (java.util.List, alice.tupr...		9722 ms (3.8%)	5968182
alice.tuprolog.Engine.**run** ()		7749 ms (3.1%)	363
alice.tuprolog.Term.**match** (alice.tuprolog.Term)		7264 ms (2.9%)	2261696
alice.tuprolog.Struct.**copy** (java.util.AbstractMap, int)		6690 ms (2.6%)	3146102
alice.tuprolog.SubGoalStore.**fetch** ()		6385 ms (2.5%)	4498597
alice.tuprolog.StateRuleSelection.**doJob** (alice.tuprolog...		6353 ms (2.5%)	924813
alice.tuprolog.Var.**free** (java.util.List)		5800 ms (2.3%)	4729335
alice.tuprolog.Var.**copy** (java.util.AbstractMap, int)		5757 ms (2.3%)	4344112
alice.tuprolog.Struct.**getTerm** ()		5732 ms (2.3%)	12339227
alice.tuprolog.Int.**unify** (java.util.List, java.util.List, alice....		5563 ms (2.2%)	3229768
alice.tuprolog.EngineManager.**spy** (String, alice.tuprolo...		4105 ms (1.6%)	4421119
alice.tuprolog.ClauseInfo.**bodyCopy** (alice.tuprolog.Su...		4053 ms (1.6%)	924075
alice.tuprolog.Var.**<init>** (String, int, int, long)		3589 ms (1.4%)	1985924
alice.tuprolog.SubGoalStore.**backTo** (alice.tuprolog.Su...		3411 ms (1.3%)	2376306
alice.tuprolog.Struct.**<init>** (int)		3217 ms (1.3%)	3157980
alice.tuprolog.Var.**free** ()		3089 ms (1.2%)	6868592
alice.util.OneWayList.**transform2** (java.util.List)		2913 ms (1.2%)	745377
alice.tuprolog.FamilyClausesList.**get** (alice.tuprolog.Term)		2913 ms (1.2%)	745377
alice.tuprolog.ClauseStore.**existCompatibleClause** ()		2835 ms (1.1%)	1264991
alice.tuprolog.Term.**<init>** ()		2821 ms (1.1%)	5619125
alice.tuprolog.StateBacktrack.**doJob** (alice.tuprolog.En...		2815 ms (1.1%)	179439
alice.tuprolog.StateGoalSelection.**doJob** (alice.tuprolog...		2710 ms (1.1%)	1568893
alice.tuprolog.ClauseStore.**checkCompatibility** (alice....		2568 ms (1%)	3113141

Figura 6: Hot Spot view relativa all'esecuzione dei benchmark in sequenza.

Avendo notato sensibile miglioramento nell'asserzione ed un generico miglioramento nelle prestazioni generali, si è scelto di svolgere quest'ulteriore fase di testing tenendo come riferimento gli stessi benchmark utilizzati nelle fasi precedenti. Per evitare che le singole risoluzioni potessero determinare risultati poco generali si è imposta la risoluzione sequenziale della totalità dei benchmark precedentemente considerati (crypt, qsort, queens, query, tak).

I dati forniti dal profiler evidenziano (figura 6) come nella distribuzione dei tempi di esecuzione siano assenti singole tipologie di chiamate in grado caratterizzare l'andamento dell'intera esecuzione, come accaduto relativamente alle asserzioni.

Hot Spots - Method	Self time [%]	Self time	Invocations
alice.tuprolog.Tokenizer.**readNextToken** ()		351 ms (7.9%)	147719
alice.tuprolog.Parser.**expr0** ()		186 ms (4.2%)	63242
alice.tuprolog.Tokenizer.**readToken** ()		178 ms (4%)	379162
alice.tuprolog.Struct.**toString** ()		178 ms (4%)	41803
alice.tuprolog.Token.**isOperator** (boolean)		161 ms (3.6%)	190264
alice.tuprolog.Parser.**parseLeftSide** (boolean, i...		152 ms (3.4%)	63308
alice.tuprolog.PrimitiveManager.**identify** (alice.t...		122 ms (2.8%)	52007
alice.tuprolog.Parser.**exprA** (int, boolean)		121 ms (2.7%)	63308
alice.tuprolog.Parser.**exprB** (int, boolean)		112 ms (2.5%)	63308
alice.tuprolog.Token.**isType** (int)		100 ms (2.3%)	174872
alice.tuprolog.Int.**toString** ()		96.6 ms (2.2%)	60030
alice.tuprolog.Parser.**isAtom** (String)		89.3 ms (2%)	62176
alice.tuprolog.Tokenizer.**unreadToken** (alice.tu...		87.8 ms (2%)	231443
alice.tuprolog.Parser.**expr0_arglist** ()		87.5 ms (2%)	42311
alice.tuprolog.Struct.**toStringAsArg** (alice.tupr...		76.1 ms (1.7%)	20000
alice.tuprolog.Token.**<init>** (String, int)		74.8 ms (1.7%)	147719
alice.tuprolog.Token.**getType** ()		72.7 ms (1.6%)	185086

Figura 7: Hot Spot view relativa alla consultazione di teorie particolarmente estese.

In questa esecuzione si può comunque notare come a rappresentare la principale voce di carico computazionale sia il processo di unificazione. Analizzando il codice non è stato possibile individuare particolari fonti di carico, quest'ultimo è infatti principalmente imputabile alla natura ricorsiva del processo in sè.

Imponendo una teoria particolarmente estesa, come ad esempio `th100` [Con10], si può notare come le principali voci di carico passino dall'unificazione alla consultazione della teoria, senza anche in questo caso essere contraddistinte da particolari picchi (figura 7).

4.4 tuProlog 2.3alpha

Fatte queste premesse si è pensato di apportare le modifiche descritte in precedenza all'ultima versione di tuProlog disponibile (2.3alpha) e di eseguire quindi un confronto prestazionale delle due, in modo da poter avere una stima degli effettivi progressi prestazionali ottenuti.

Le modifiche apportate riguardano, rispettivamente:

- l'introduzione del meccanismo di fast match (indicizzazione) con esclusione della verifica ciclica degli alberi, che ha compreso la sostituzione/introduzione delle classi:

  `alice.util.ReadOnlyLinkedList`

Benchmark	iterations	2.3alpha [ms]	2.3 (fast match) [ms]	improvement
crypt	1000	95.0	75.9	20.1%
qsort	10000	9.3	7.9	15.1%
queens	100	7767.8	7519.9	3.2%
query	1000	62.6	22.3	64.4%
tak	100	1573.4	1564.0	0.6%

Tabella 5: Tempi di esecuzione delle due implementazioni (2.3.0alpha)

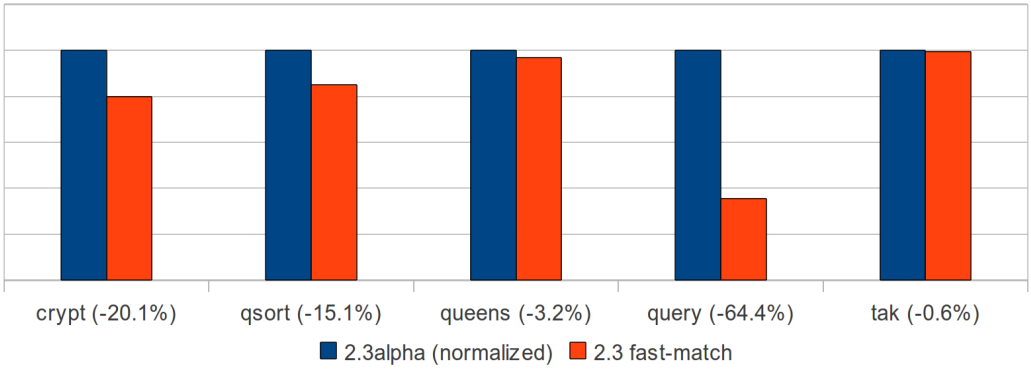

Figura 8: Benchmark tuProlog 2.3 dopo l'applicazione di fast-match e modiche proposte.

```
    "     .OneWayList
alice.tuprolog.ClauseDatabase
      "       .ClauseFilter
      "       .ClauseStore
      "       .Double
      "       .FamilyClauseIndex
      "       .FamilyClauseList
      "       .Float
      "       .Int
      "       .Long
      "       .Number
      "       .RBTree
      "       .Struct
      "       .Term
      "       .TheoryManager
      "       .Var
```

- la modifica del metodo `Struct.getTerm(int)`

Come mostrato in figura 8 l'esecuzione dei benchmark ha registrato un miglioramento medio delle prestazioni del 20.7% preservando la completa funzionalità dell'engine. Le modifiche proposte sembrano dunque rappresentare un significativo passo in avanti nell'evoluzione dell'engine tuProlog.

Nota 1 prima dell'avvio della fase di benchmarking ci si è assicurati, a valle dell'applicazione delle modifiche descritte, che la versione 2.3alpha continuasse a completare con successo la suite di test predefinita.

Nota 2 per la corretta compilazione del progetto presente all'interno dell'archivio `2p-2.2.3.0alpha.zip`[7] è necessario il download del package javassist[8].

5 Tail Recursion Optimization

L'ultima fase affrontata è stata l'analisi dell'attuale implementazione della *tail recursion optimization* di TuProlog.

L'attuale versione e tutte le precedenti presentano un approccio alla soluzione del problema che però non riesce ad ottenere, in pratica, l'ottimizzazione (e il conseguente boost prestazionale). Per poter individuare il contesto specifico nel quale intervenire è necessario, per prima cosa andare ad analizzare il meccanismo attraverso il quale l'engine arriva alla risoluzione dei goal.

Il motore è composto da un FSA (figura 9) dalla struttura piuttosto sintetica e da una serie di strutture dati ad albero, quest'ultime in fase di esecuzione possono essere ricondotte a strutture analoghe a stack.

Ogni volta in cui l'engine va a risolvere una determinata clausola viene instanziato un corrispondente `ExecutionContext`, linkato ai precedenti mediante la definizione della proprietà `fatherCtx`. All'interno della classe è presente il metodo `performTailRecursionOptimization(Engine)` che assegna `fatherCtx` discriminando in funzione dell'esistenza di soluzioni alternative all'interno del contesto (`!haveAlternatives`) unitamente all'esistenza di sottogoal attraverso la chiamata `!goalsToEval.haveSubGoals()`: qual'ora le due condizioni siano entrambe verificate il contesto verrà linkato non al contesto corrente dell'`Engine` passato in ingresso, ma al padre (`fatherCtx`) del contesto corrente di quest'ultimo.

[7] disponibile per il download all'indirizzo http://tuprolog.sourceforge.net

[8] http://sourceforge.net/projects/jboss/files/Javassist/3.11.0.GA/javassist-3.11.GA.zip/download

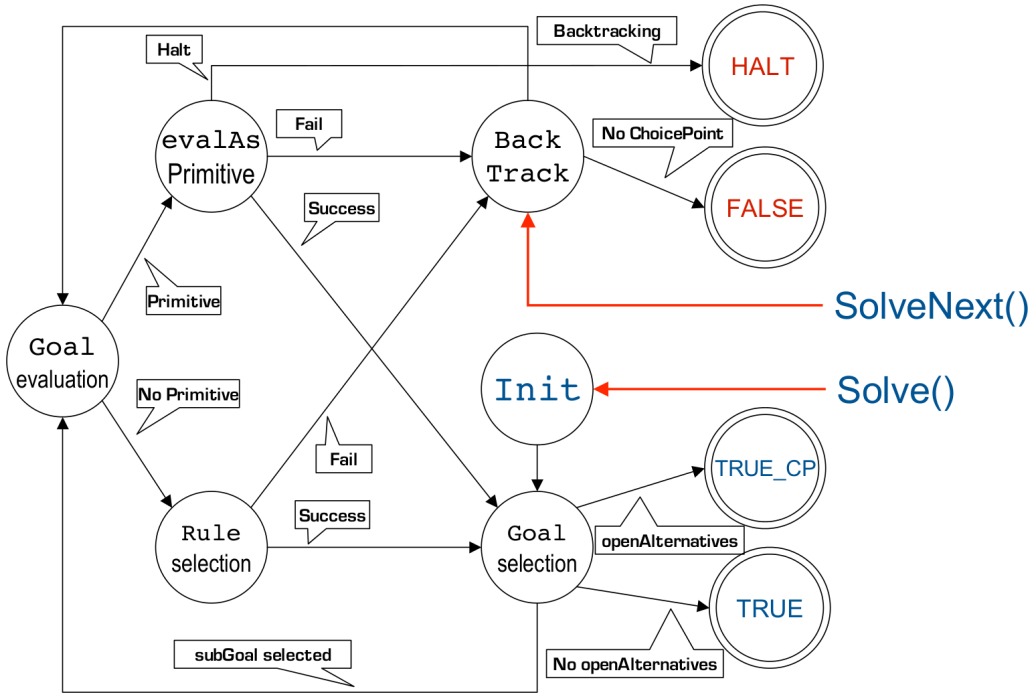

Figura 9: Macchina a stati alla base del motore TuProlog.

Questo meccanismo vorrebbe, nelle intenzioni, attuare l'ottimizzazione andando ad eliminare i nodi inutili nell'albero di risoluzione costruito durante l'esecuzione. Ciò che però non permette la corretta attuazione dell'ottimizzazione per come essa è attualmente implementata è probabilmente la serie di operazioni innescate dall'invocazione `goalsToEval.haveSubGoals()` precedentemente menzionata: tale metodo, definito all'interno della classe `SubGoalStore` va a valutare se un indice interno sia minore o uguale della dimensione di un `SubGoalTree` rappresentante i goal da risolvere all'interno del contesto di riferimento.

```
public boolean haveSubGoals() {
    return (index <= goals.size());
}
```

Il debug ha evidenziato come ogni chiamata del metodo sopra riportato avvenga in condizioni tali da impedire che il valore booleano ritornato possa essere *false*. In particolare il valore di `index` è sempre uguale a 0, mentre la dimensione della struttura `goals` ne è sempre strettamente maggiore.

Il valore di `index` viene settato a 0 in fase di instanziamento, e prevede poi un possibile incremento ad ogni invocazione del metodo `fetch()`: si è però appurato che su tutte le instanze sulle quali viene invocato il metodo `haveSubGoals()` non è in nessun caso stato precedentemente invocato il metodo `fetch()`. Tale circostanza si verifica a causa del fatto che l'operazione di fetch sulla variabile goalsToEval avvenga nello stato di *GoalSelection*, mentre l'ottimizzazione-tail è giustamente collocata nello stato *RuleSelection*.

Volendo realizzare un'implementazione funzionante di TRO sarà necessario andare ad agire sulla gestione degli `ExecutionContext`, forzando l'engine ad evitare di ripercorrere il percorso risolutivo generato ed a fornire subito in uscita l'esito della computazione: ovviamente a patto che nella teoria settata sia correttamente implementato un algoritmo che supporti la ricorsione tail.

Probabilmente sarà necessario integrare informazioni ora non presenti all'interno delle strutture dati coinvolte, sulle quali basare la discriminazione dell'engine in fase di risoluzione. Attualmente l'ottimizzazinoe è richiamata quando il FSA si trova nello stato di RuleSelection (classe `StateRuleSelection`), ed è molto probabilmente proprio in questa fase che essa dovrebbe concretizzarsi, dato che è in essa che gli `ExecutionContext` vengono gestiti.

6 Conclusioni

L'integrazione dell'indexing all'interno del motore TuProlog e la modifica della classe `Struct` hanno permesso un miglioramento medio delle prestazioni del motore pari al 20%, con picchi del 64% in benchmark specifici.

La riduzione dei tempi nella consultazione delle teorie e nel reperimento dei termini rappresentano sicuramente il principale fattore responsabile dello speedup ottenuto, ed un ulteriore contribuito al risultato finale è sicuramente da imputare anche gli accorgimenti orientati al risparmio di memoria allocata che, limitando gli interventi del garbage collector, permettono una maggiore disponibilità di risorse all'environment.

L'ultima sessione di profiling svolta (sec. 4.3) sembra evidenziare la presenza di una sorta di equilibrio tra i costi per l'accesso alle strutture dati e quelli relativi alla loro gestione: tentando di variare le percentuali secondo cui questi due aspetti sono agevolati non si sono in nessun caso ottenute variazioni prestazionali apprezzabli.

Allo stato attuale volendo incrementare ulteriormente le prestazione dell'engine sembra essere necessaria un'evoluzione della sua struttura esplicitamente orientata al miglioramento del meccanismo di unificazione, piuttosto che un'ulteriore fase di raffinamento svolto a livelli molto dettagliati.

Detto questo è però importante tenere conto del fatto che integrando un'implementazione funzionanto della *tail recursion optimization* si potrebberò ridurre sensisbilmente i tempi necessari alla produzione dell'output. Con interventi mirati nell'ambito della gestione degli stati del FSA sarà molto probabilmente possibile introdurre la TRO senza modificare la sostanza dell'attuale meccanismo di risoluzione.

Riferimenti bibliografici

[Dam08] Michele Damian
Analisi di prestazione dell'interprete tuProlog su piattaforma Java
Facoltà di ingegneria, Università di Bologna, 2008

[Con10] Paolo Contessi
Relazione del progetto *Fast-Match*
II facoltà di ingegneria, Università di Bologna, 2010

[Has99] Hassan Ait-Kaci
Warren's Abstract Machine - A Tutorial Reconstruction
MIT Press, febr. 1999

[2p-e] A. Benini, A. Omicini, G. Piancastelli, A. Ricci
The Architecture and Design of a Malleable Object-Oriented Prolog Engine
DEIS, Università di Bologna, 2008

[2p-d] *tuProlog Javadoc, source code*
http://tuprolog.sourceforge.net

[Sics] *SICStus Prolog performance benchmarks*
http://www.sics.se/isl/sicstuswww/site/performance.html